Programming Languages

http://www.cse.iitd.ac.in/~sak/courses/pl/2021-22/index.html

S. Arun-Kumar

Department of Computer Science and Engineering

I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

April 17, 2023

Contents

1. The Programming Languages Overview

What is a Programming Language?

- A (*linear*[a]) notation for the *precise, accurate and complete* description of algorithms and data-structures *implementable* on a digital computer.
- In contrast,
  - the usual *mathematical* notation is accurate and precise enough for human beings but is not necessarily implementable on a digital computer,
  - and often the usual *mathematical* notation is not linear (think of integrals or matrices).
  - *pseudo-code* for algorithms and data-structures is too *abstract*[b] to be directly executed on a digital computer or even a virtual computer.
- A *program* is a sentence in a programming language intended to describe algorithms designed for a *universal computing machine*.
- While algorithms terminate not all programs may terminate.

[a] i.e. a sequence of characters

[b] Too many implementation details are either left unspecified or implicit.

The World of PLs

- There are just too many actual programming languages and more are being designed every year. Impossible to master every new PL that is released.
- Often impossible to master every feature of even the PLs that are currently in use.
- Often not necessary to master all features of a PL.

Why Study the subject of PL? - 1

To understand the various major paradigms of programming.

- The same algorithm requires different design considerations in different paradigms.
- Different data-structures as part of the language,
- Different libraries provided along with the language implementation.
- Different styles of thought involved in the implementation.

Why Study the subject of PL? - 2

To understand the major features and their implementation common to large numbers of PLs.

- The same feature may be implemented differently in different PLs.
- The same algorithm is written differently in different PLs of the same paradigm depending upon
  - the data-structures available,
  - the control structures available,
  - the libraries available.

Why Study the subject of PL? - 3

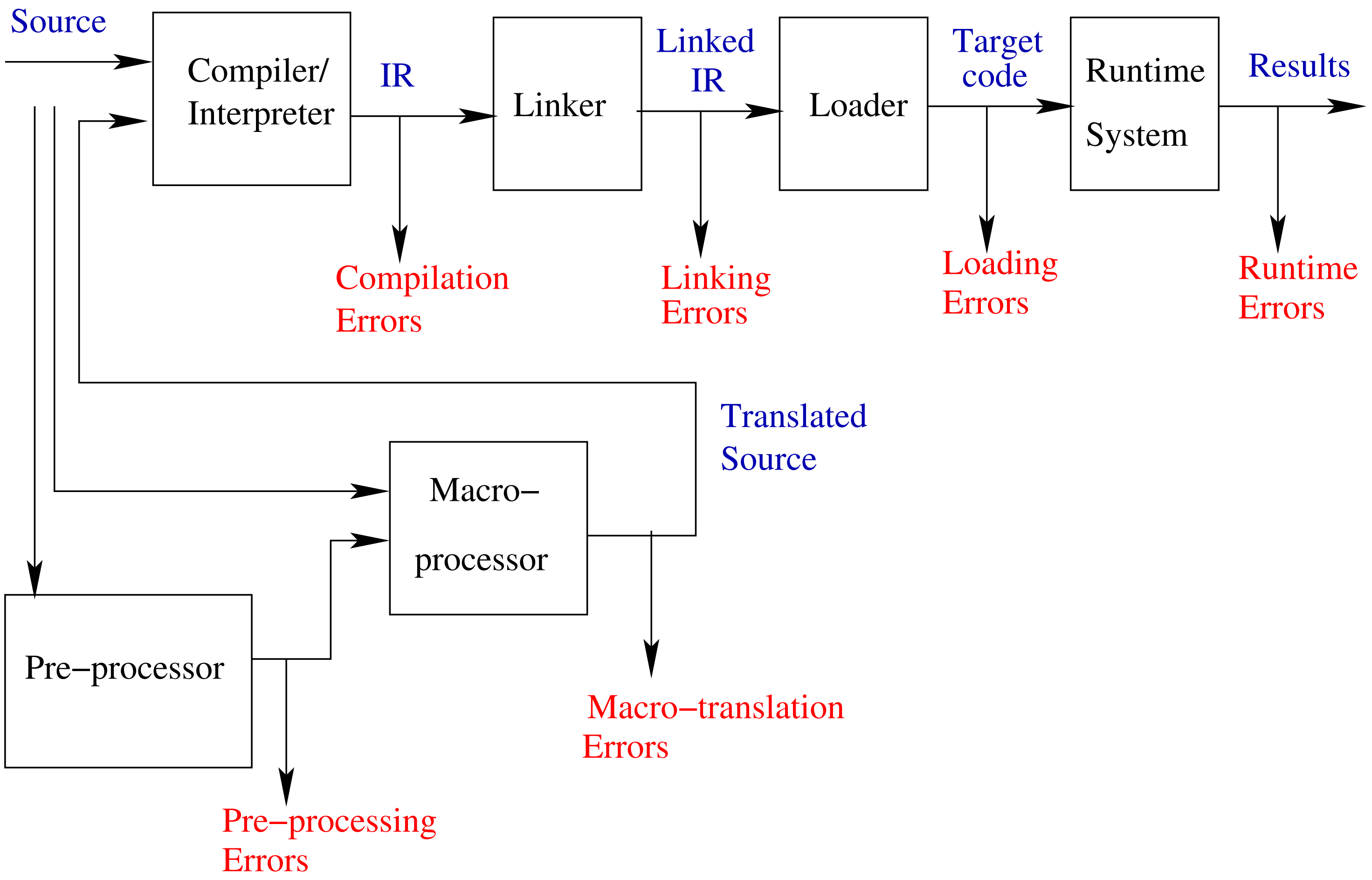
To understand the major design and architectural considerations common to most PLs.

- Whether a data- or control-structure is part of the programming language itself.
- Whether certain complex (data- and control-)structures are provided as libraries of the programming language.

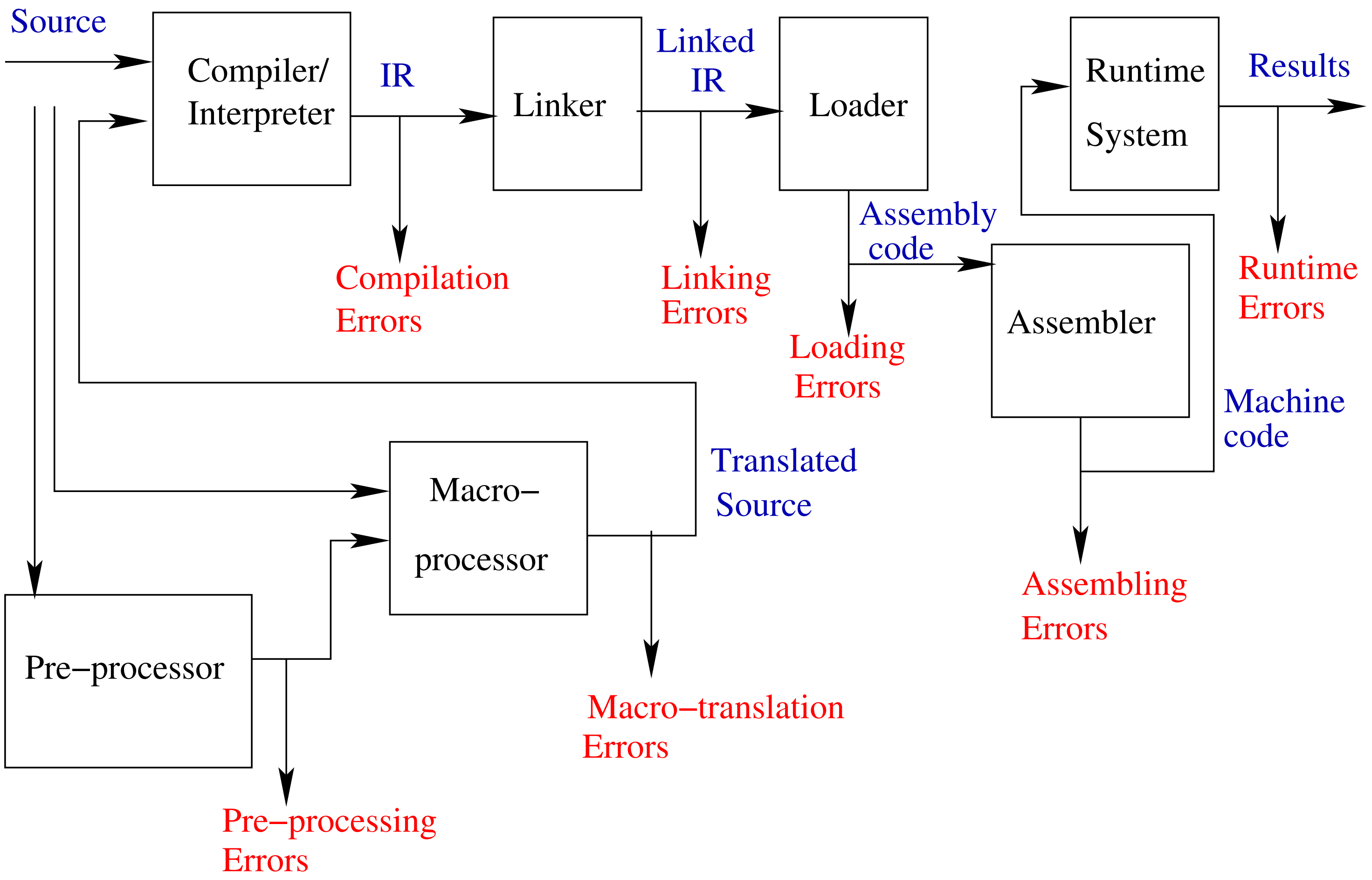
Architectural Considerations in PL

- Compilation vs Interpretation
- Portability considerations across hardware architectures
- Virtual machines or target architectures.
- Stack architecture vs. register architecture.
- Representation and typing.
- The set of intermediate languages/representation required for the implementation.
- Support for parallelism.

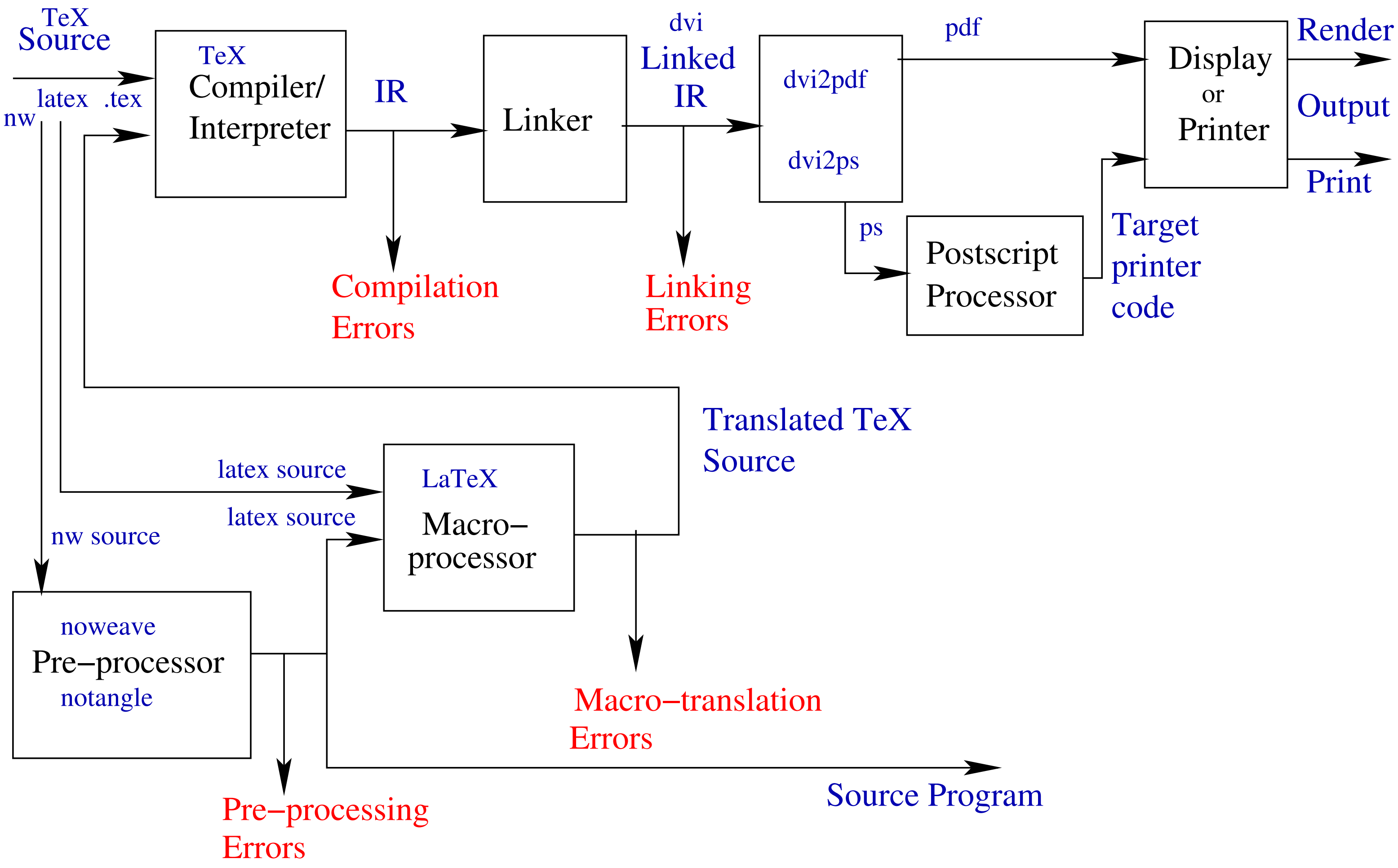
Programs: Source to Runs



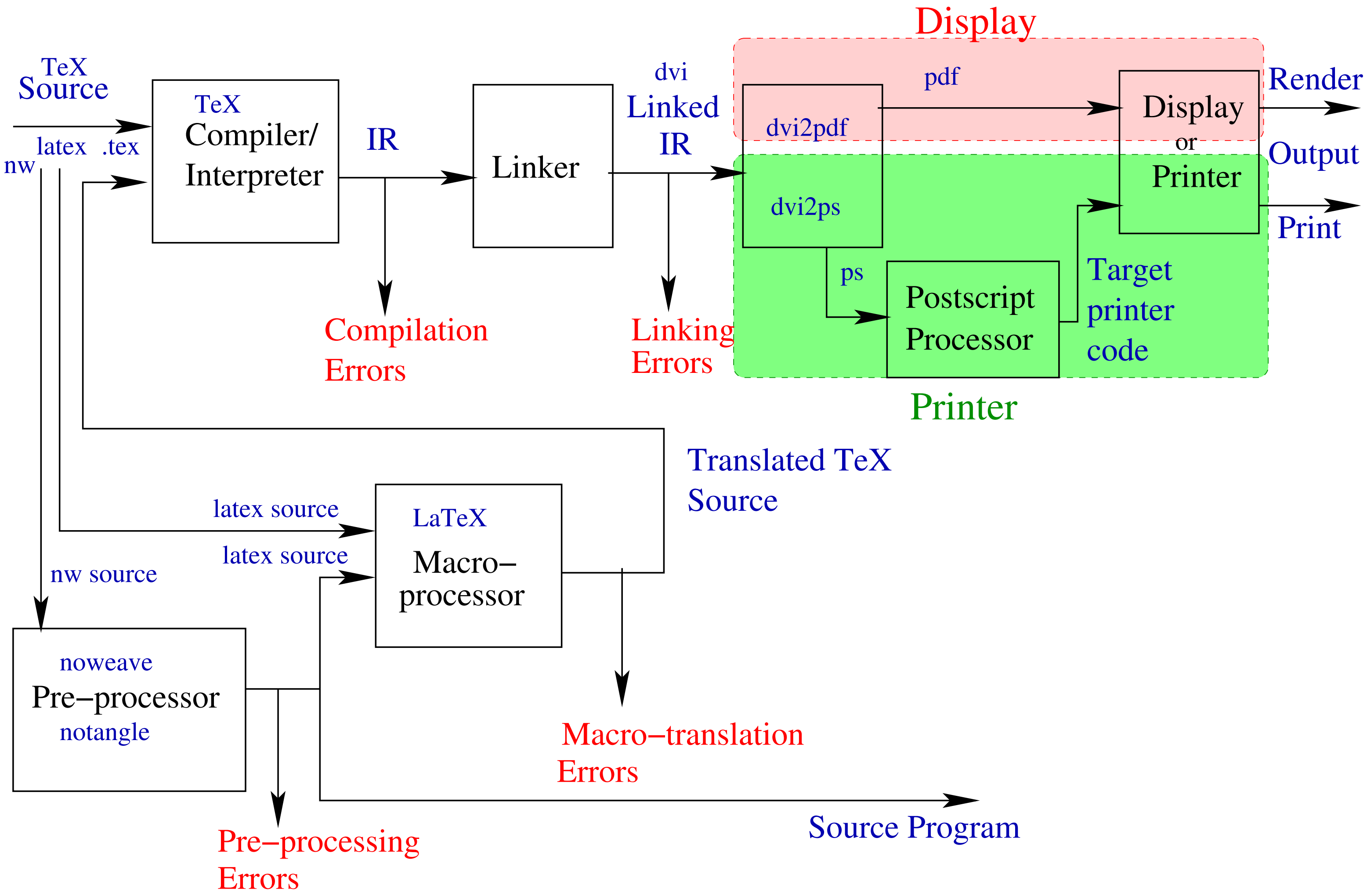
Programs: Source to Runs-2



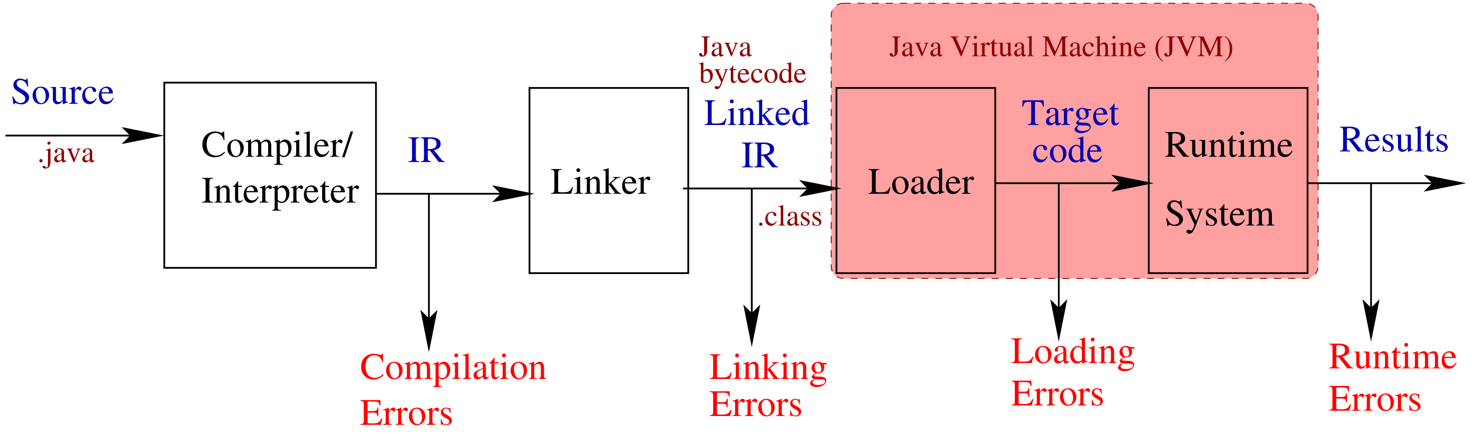
Programs: Source to Runs-1: LaTeX



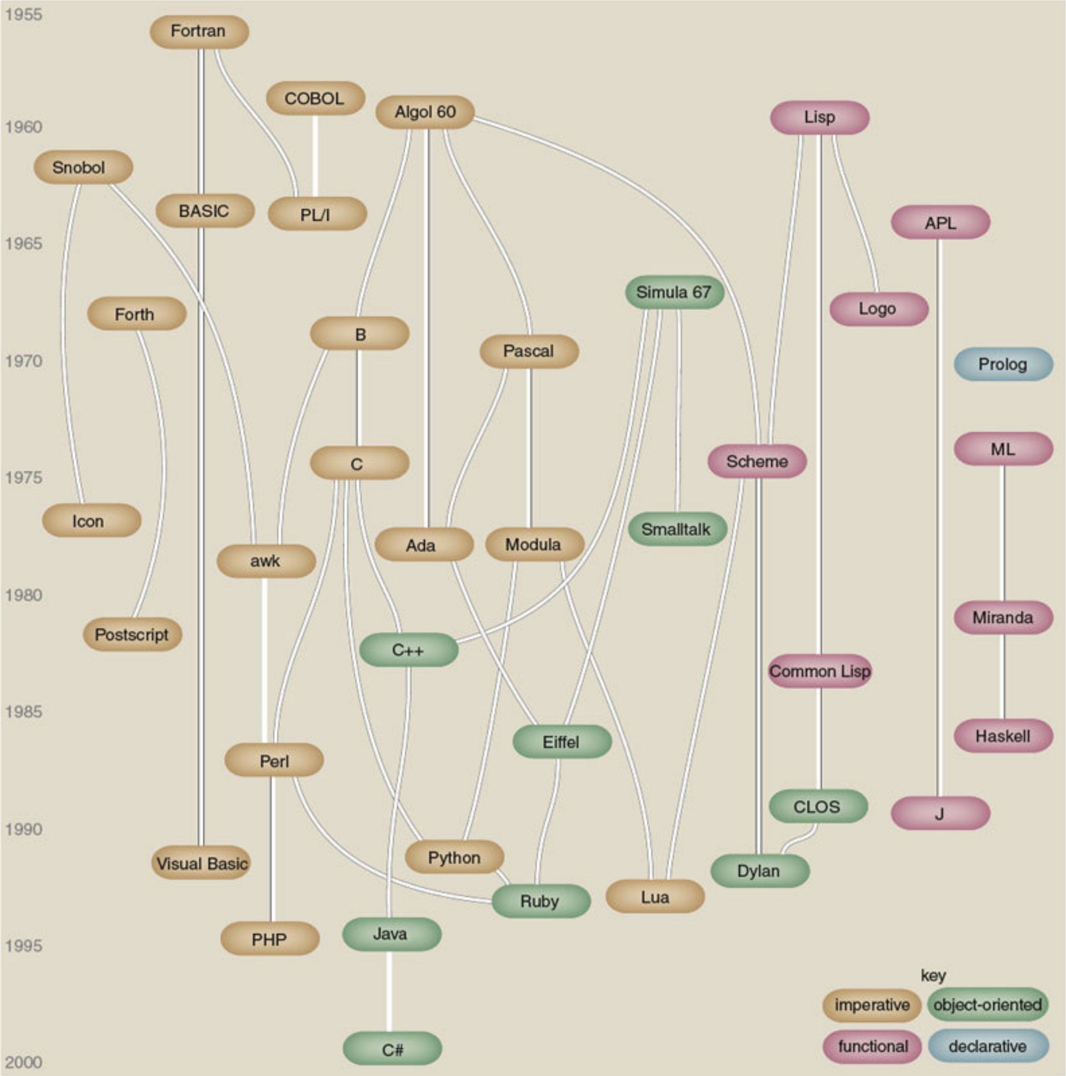
Programs: Source to Runs-2: LaTeX



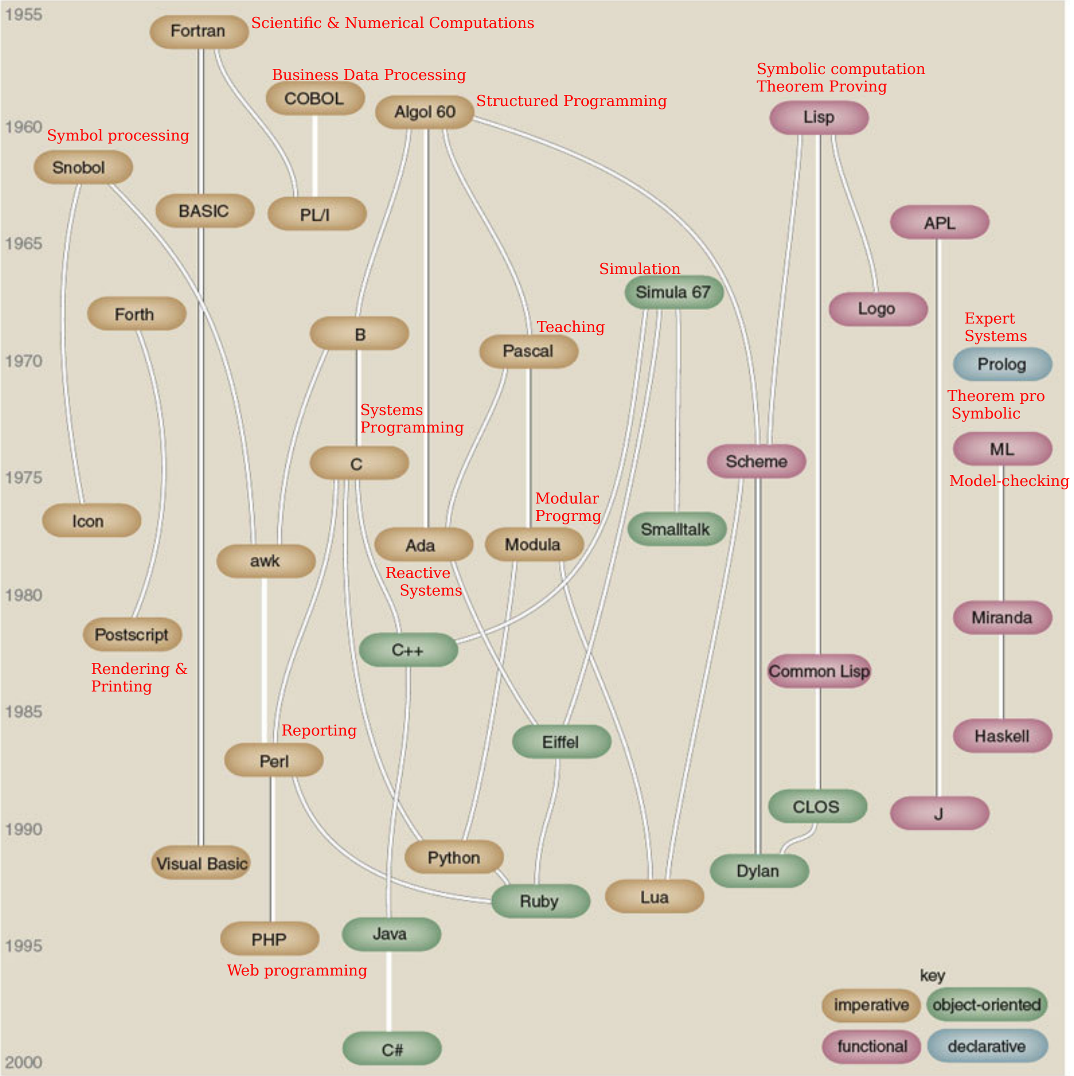
Programs: Source to Runs: Java



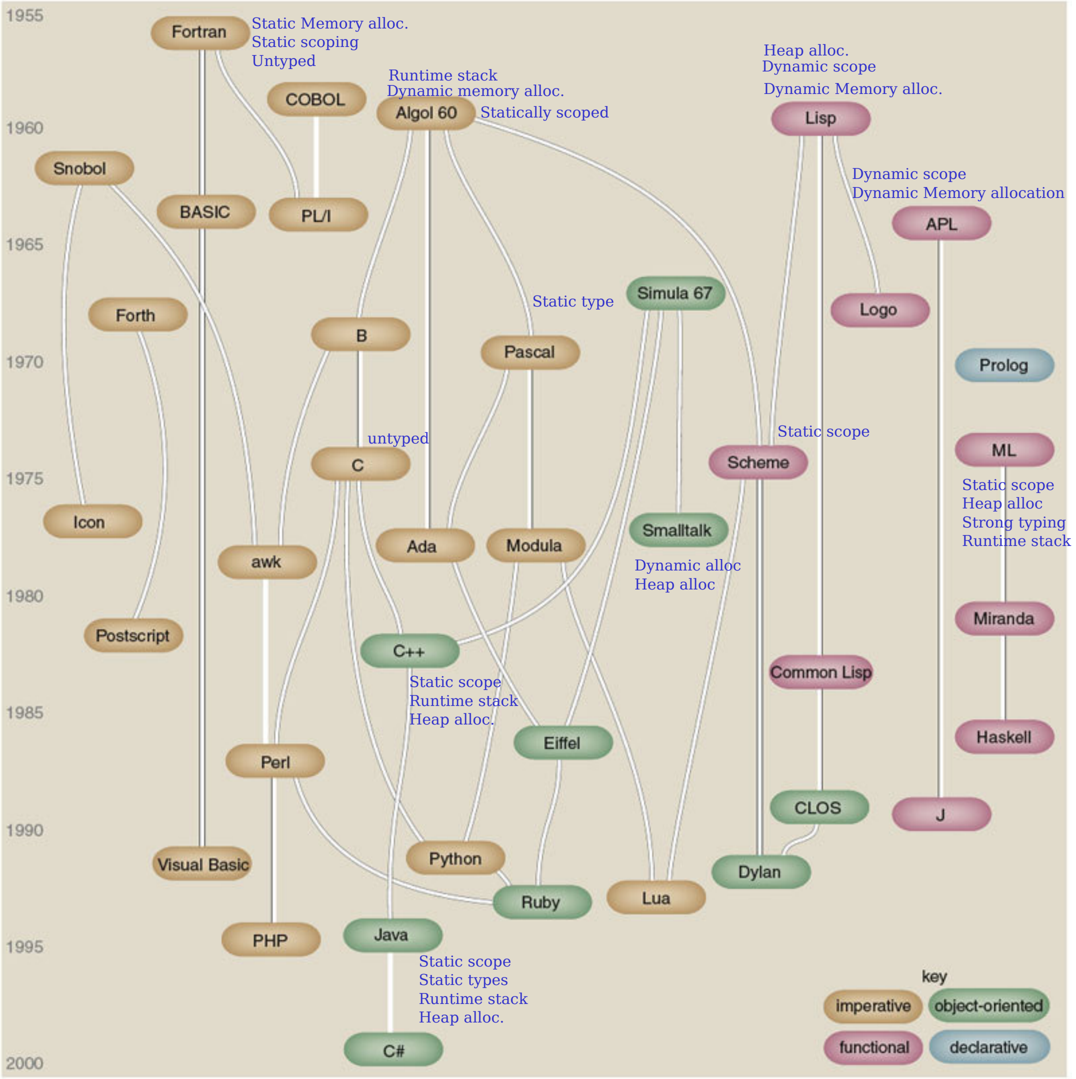
The Landscape of General PLs



The Usage of General PLs



The Major Features of General PLs



FORTRAN

- The very first high-level programming language
- Still used in scientific computation
- Did not allow recursion
- Static memory allocation (since no recursion was allowed)
- Very highly compute oriented
- Runs very fast because of static memory allocation
- Parameter passing by reference

COBOL

- A business oriented language
- Did not allow recursion
- Extremely verbose
- Very highly input-oriented
- Meant to manage large amounts of data on disks and tapes and generate reports
- Not computationally friendly

LisP

- First functional programming language
- Recursion allowed and extensively used
- Introduced lists and list-operations as the only data-structure
- Introduced symbolic computation
- Much favoured for AI and NLP programming for more than 40 years
- The first programming language whose interpreter could be written in itself.

ALGOL-60

- Introduced the Backus-Naur Form (BNF) for specifying the syntax of a programming langauge
- Formal syntax defined by BNF (an extension of context-free grammars)
- First imperative language to implement recursion
- Introduction of block-structure and nested scoping
- Dynamic memory allocation
- Introduced the call-by-name parameter passing mechanism

Pascal

- ALGOL-like language meant for teaching structured programming
- Introduction of new data structures – records, enumerated types, sub-range types, recursive data-types
- Its simplicity led to its “dialects” being adopted for expressing algorithms in pseudo-code
- First language to be ported across a variety of hardware and OS platforms – introduced the concepts of virtual machine and intermediate code (bytecode)

ML

- First strongly and statically typed functional programming language
- Created the notion of an inductively defined type to construct complex types
- Parametric polymorphism allowing code reuse and type-instantiation.
- Powerful pattern matching facilities on complex data-types.
- Introduced type-inference, thus making declarations unnecessary except in special cases
- Its module facility is inspired by the algebraic theory of abstract data types
- The first language to introduce functorial programming between algebraic structures and modules

Prolog

- First declarative programming language
- Uses the Horn clause subset of first-order logic
- Goal-oriented programming implementing a top-down methodology
- Implements backtracking as a language feature
- Powerful pattern-matching facilities like in functional programming
- Various dialects implement various other features such as constraint programming, higher-order functions etc.

2. Introduction to Compiling

Introduction to Compiling

- Translation of programming languages into executable code
- But more generally any large piece of software requires the use of compiling techniques.
- The processes and techniques of designing compilers is useful in designing most large pieces of software.
- Compiler design uses techniques from formal language theory, automata theory, data structures and algorithms and computability theory.

Software Examples

Some examples of other software that use compiling techniques

- Almost all user-interfaces require scanners and parsers to be used.
- All XML-based software require interpretation that uses these techniques.
- All mathematical text formatting requires the use of scanning, parsing and code-generation techniques (e.g. LaTeX).
- Model-checking and verification software are based on compiling techniques
- Synthesis of hardware circuits requires a description language and the final code that is generated is an implementation either at the register-transfer level or gate-level design.

Books and References

1. **Appel A W**. *Modern Compiler Implementation in Java* Cambridge University Press, Revised Indian Paperback edition 2001
2. **Aho A V, Sethi R, Ullman J D**. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley 1986.
3. **Muchnick S S**. *Advanced Compiler Design and Implementation*, Academic Press 1997.

A Plethora of Languages: Compiling

In general a **compiler/interpreter** for a a **source** language $\mathcal{S}$ written in some language $\mathcal{C}$ translates code written in $\mathcal{S}$ to a **target** language $\mathcal{T}$.

Source $\mathcal{S}$

Target $\mathcal{T}$

Language of the compiler/interpreter $\mathcal{C}$

Our primary concern. Compiling from a *high-level* *source* programming language to a *target* language using a high-level language $\mathcal{C}$.

A Plethora of Languages: Source

The Source language $\mathcal{S}$ could be

- a programming language, or
- a description language (e.g. Verilog, VHDL), or
- a markup language (e.g. XML, HTML, SGML, LaTeX) or
- even a "mark-down" language to simplify writing code.

A Plethora of Languages: Target

The Target language $\mathcal{T}$ could be

- an intermediate language (e.g. ASTs, IR, bytecode etc.)
- another programming language, assembly language or machine language, or
- a language for describing various objects (circuits etc.), or
- a low level language for execution, display, rendering etc. or
- even another high-level language.

The Compiling Process

Besides $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{T}$ there could be several other intermediate languages $\mathcal{I}_1, \mathcal{I}_2, \ldots$ (also called **intermediate representations**) into which the source program could be translated in the process of compiling or interpreting the source programs written in $\mathcal{S}$. In modern compilers, for portability, modularity and reasons of code improvement, there is usually at least one intermediate representation.

Some of these intermediate representations could just be data-types of a modern functional or object-oriented programming language.

Compiling as Translation

Except in the case of a *source to source* translation (for example, a Pascal to C translator which translates Pascal programs into C programs), we may think of the process of compiling *high-level* languages as one of transforming programs written in $\mathcal{S}$ into programs of *lower-level* languages such as the intermediate representation or the target language. By a *low-level* language we mean that the language is in many ways closer to the architecture of the target language.

Phases of a Compiler

A compiler or translator is a fairly complex piece of software that needs to be developed in terms of various independent modules.

In the case of most programming languages, compilers are designed in **phases**.

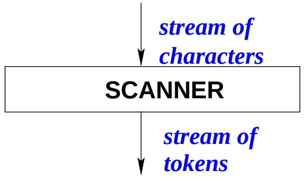
The various phases may be different from the various **passes** in compilation.

Phases vs. Passes

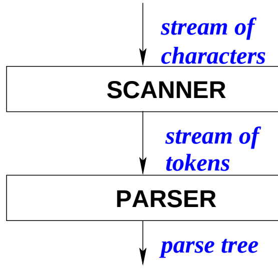
Several phases may be combined into a single pass, which essentially means that even though we describe the phases as the different transformations the whole source program undergoes, in reality various phases can be undertaken in a single pass with partial or incomplete information about the whole source program.

Most modern programming languages are designed so that a compiler for th elanguage does not require more than 2 passes

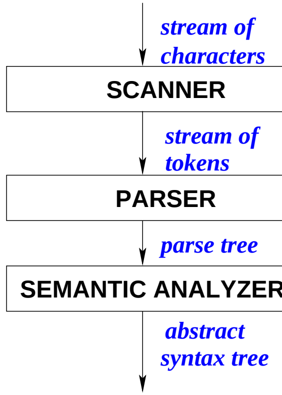
The Big Picture: 1



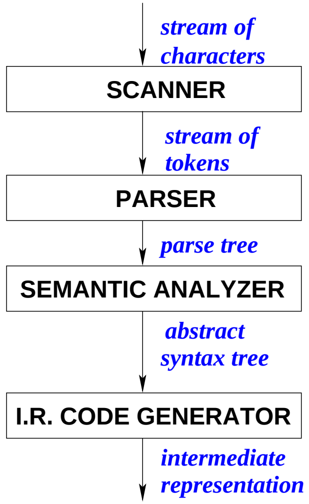
The Big Picture: 2



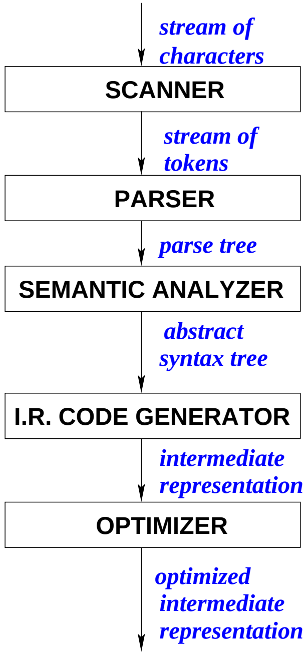
The Big Picture: 3



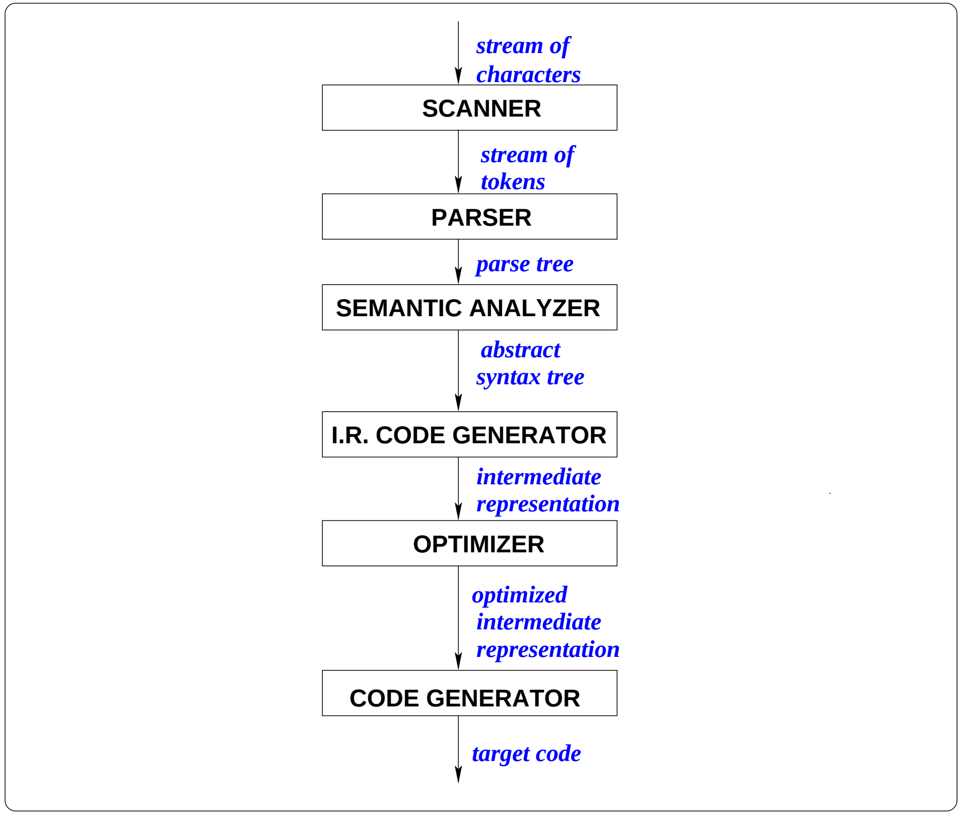
The Big Picture: 4



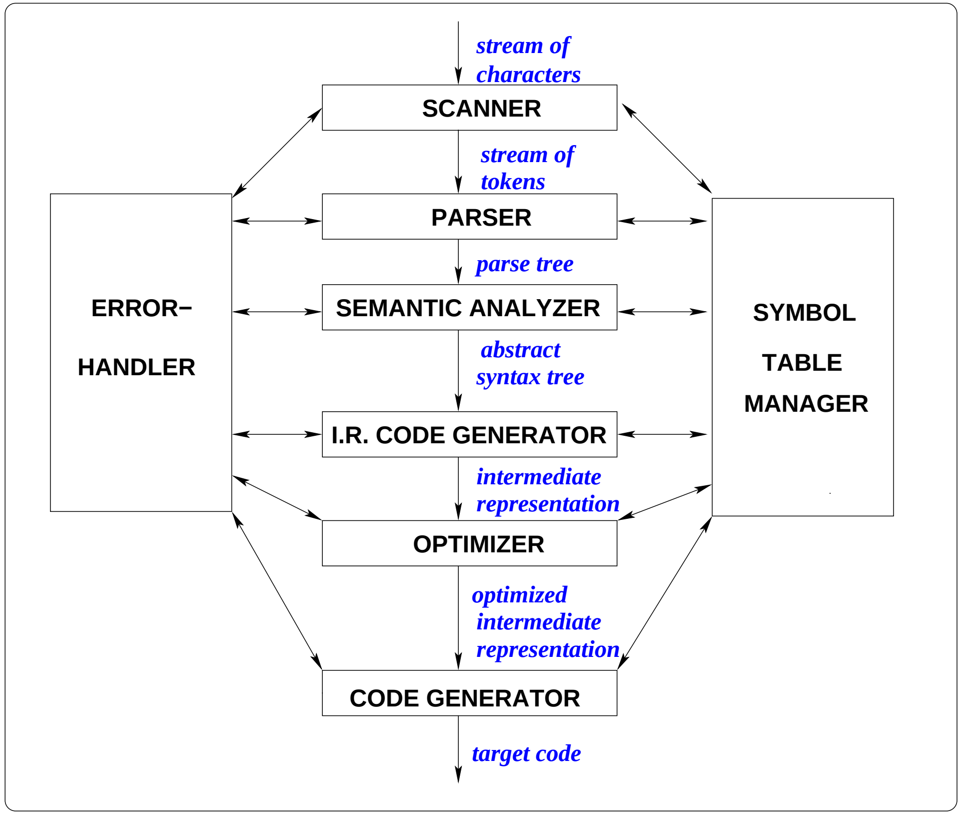
The Big Picture: 5



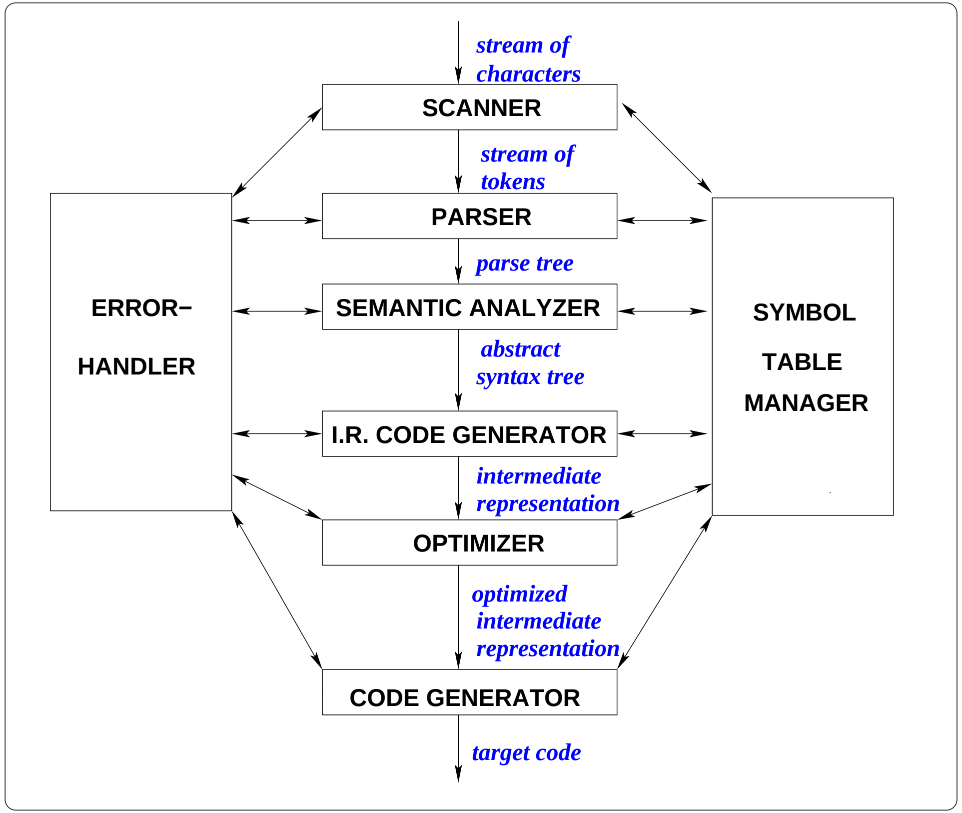
The Big Picture: 6



The Big Picture: 7



The Big Picture: 8



Scanner	*Parser*	*Semantic Analysis*	*Symbol Table*
IR	*Run-time structure*		

Lexical Analysis

Programming Language Elements

- Every language is built from a finite **alphabet** of **symbols**. The alphabet of a programming language (nowadays) consists of the symbols of the ASCII character set[a].
- Each language has a **vocabulary** consisting of **words**. Each word is a *string of (printable non-whitespace) symbols* drawn from the alphabet.
- Each language has a finite set of **punctuation symbols**, which separate phrases, clauses and sentences.
- A programming language also has a finite set of **operators**.
- The phrases, clauses and sentences of a programming language are expressions, commands, functions, procedures and programs.

[a] Previously there were others such as BCD and EBCDIC which are no longer used.

Lexical Analysis

lex-i-cal: *relating to words of a language*

- A *source program* (usually a file) consists of a stream of characters.
- Given a stream of characters that make up a *source program* the compiler must first break up this stream into a sequence of "lexemes", and other symbols.
- Each such lexeme is then classified as belonging to a certain **token type**.
- Certain lexemes may violate the pattern rules for tokens and are considered erroneous.
- Certain sequences of characters are not[a] tokens and are completely ignored (or skipped) by the compiler.

[a] E.g. comments

Erroneous lexemes

Some lexemes violate all rules of tokens. Some examples common to most programming languages

- `12ab` would not be an identifier or a number in most programming languages. If it were an integer in Hex code it would be written `0x12ab`.
- `127.0.1` is usually not any kind of number. However `127.0.0.1` may be a valid token representing an IP address.

Tokens and Non-tokens: 1

- Tokens
- Non-tokens

Tokens and Non-tokens: 2

Tokens. Typical tokens are

- **Constants**: Integer, Boolean, Real, Character and String constants.
- **Identifiers**: Names of variables, constants, procedures, functions etc.
- **Keywords/Reserved words**: `void`, `public`, `main`
- **Operators**:+, $*$, /
- **Punctuation**: `,`, `:`, `.`
- **Brackets**: `(`, `)`, `[`, `]`, `begin`, `end`, `case`, `esac`

Non-tokens

Tokens and Non-tokens: 3

Tokens

Non-tokens. Typical non-tokens are

- **whitespace**: sequences of tabs, spaces, new-line characters,
- **comments**: compiler ignores comments
- **preprocessor directives**: `#include ...`, `#define ...`
- **macros** in the beginning of C programs

Scanning: 1

During the scanning phase the compiler/interpreter

- takes a stream of characters and identifies tokens from the lexemes.
- Eliminates comments and redundant whitepace.
- Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting and handling to the user.

Scanning: 2

Definition 3.1 *A* **lexeme** *is a basic lexical unit of a language consisting of one word or several words, the elements of which do not separately convey the meaning of the whole.*

- Whitespace: A sequence of space, tab, newline, carriage-return, form-feed characters etc.
- Lexeme: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators or punctuation symbols)
- Examples of lexemes.
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives

Scanning: 3

Definition 3.2 *A* **token** *consists of an abstract name and the attributes of a lexeme.*

- Token: A sequence of characters to be treated as a single unit.
- Examples of tokens.
  - Reserved words (e.g. **begin**, **end**, **struct**, **if** etc.)
  - Keywords ($integer$, $true$ etc.)
  - Operators ($+$, $\&\&$, $++$ etc)
  - Identifiers (variable names, procedure names, parameter names)
  - Literal constants (numeric, string, character constants etc.)
  - Punctuation marks ($:$, $,$ etc.)

Scanning: 4

- Identification of tokens is usually done by a Deterministic Finite-state automaton (DFA).
- The set of tokens of a language is represented by a large regular expression.
- This regular expression is fed to a lexical-analyser generator such as Lex, Flex or JLex.
- A giant DFA is created by the Lexical analyser generator.

Lexical Rules

- Every programming language has **lexical rules** that define how a token is to be defined.

  Example. In most programming languages identifiers satisfy the following rules.

  1. An identifier consists of a sequence of of letters (`A` ... `Z`, `a` ... `z`), digits (`0` ... `9`) and the underscore (`_`) character.
  2. The first character of an identifier must be a letter.

- Any two tokens are separated by some **delimiters** (usually whitespace) or non-tokens in the source program.

Regular Expressions

Consider the string 11/12/2021? What does this string of characters represent? There are at least the following different possibilities.

- In a school mathematics text it might represent the operation of division (11/12)/2021, i.e the fraction 11/12 divided by 2021, yielding the value .00045357083951839023.
- To a student who is confused, it may also represent the operation of division 11/(12/2021) i.e. the result of dividing 11 by the fraction 12/2021 yielding the value 1852.5833333333333569846.
- In some official document from India it might represent a date (in *dd/mm/yyyy* format) viz. 11 December 2021[1].
- In some official document from America it might represent a different date (in *mm/dd/yyyy* format) viz. November 12, 2021[2].

The ambiguity inherent in such representations requires that (especially if the school mathematics text also uses some date format in some problems) a clearer specification of the individual elements be provided. These specifications of individual elements in programming languages are provided by *lexical rules*. These lexical rules specify “patterns” which are legal for use in the language.

[1]Have you heard of the 26/11 attack?
[2]Have you heard of the 9/11 attack?

Specifying Lexical Rules

We require compact and simple ways of specifying the lexical rules of the tokens of a language. In particular,

- there are an *infinite* number of legally correct identifiers (names) in any programming language.
- we require *finite descriptions/specifications* of the lexical rules so that they can cover the infinite number of legal tokens.

One way of specifying the lexical rules of a programming language is to use **regular expressions**.

Regular Expressions Language

- Any set of strings built up from the symbols of `A` is called a **language**. $\mathtt{A}^*$ is the set of all finite strings built up from `A`.
- Each **regular expression** is a *finite* sequence of symbols made up of symbols from the alphabet and other symbols called **operators**.
- A **regular expression** may be used to describe an *infinite* collection of strings.

Regular Expressions for Identifiers

Example 3.3 *The regular expression used to define the set of possible identifiers as defined by the rules is*

$$[\mathbf{A-Za-z}][\mathbf{A-Za-z0-9\_}]^*$$

- *The letters and digits in* **bold blue** *and _ denote symbols drawn from the alphabet, consisting of lower-case, upper-case letters, digits and _.*
- *The other symbols in blue — the brackets "[", "]", hyphen "–" and asterisk "*" — are operator symbols of the language of regular expressions.*
- *The hyphen operator "–" allows for range specifications in the ASCII alphabet.*
- *The asterisk "*" specifies "0 or more occurrences" of the symbols within the brackets.*

Concatenations

Consider a (finite) alphabet (of symbols) $\mathtt{A}$.

- Given any two strings x and y in a language, $x.y$ or simply xy is the **concatenation of the two strings**.

  Example 3.4 *Given the strings* $x = \mathtt{Mengesha}$ *and* $y = \mathtt{Mamo}$, $x.y = \mathtt{MengeshaMamo}$ *and* $y.x = \mathtt{MamoMengesha}$.

- Given two languages X and Y, then $X.Y$ or simply XY is the **concatenation of the languages**.

  Example 3.5 *Let* $X = \{\mathtt{Mengesha}, \mathtt{Gemechis}\}$ *and* $Y = \{\mathtt{Mamo}, \mathtt{Bekele}, \mathtt{Selassie}\}$. *Then*
  $XY = \{\mathtt{MengeshaMamo}, \mathtt{MengeshaBekele}, \mathtt{MengeshaSelassie},$
  $\mathtt{GemechisMamo}, \mathtt{GemechisBekele}, \mathtt{GemechisSelassie}\}$

Note on the Concept of “language”.

Unfortunately we have too many related but slightly different concepts, each of which is simply called a “language”. Here is a clarification of the various concepts that we use.

- Every language has a non-empty finite set of symbols called **letters**. This non-empty finite set is called the **alphabet**.
- Each **word** is a finite sequence of symbols called **letters**.
- The words of a language usually constitute its **vocabulary**. Certain sequences of symbols may not form a word in the vocabulary. A vocabulary for a natural language is defined by a *dictionary*, whereas for a programming language it is usually defined by *formation rules*.
- A **phrase**, **clause** or **sentence** is a finite sequence of words drawn from the vocabulary.
- Every natural language or programming language is a finite or infinite set of **sentences**.
- In the case of formal languages, the formal language is the set of words that can be formed using the formation rules. The language is also said to be **generated** by the formation rules.

There are a variety of languages that we need to get familiar with.

Natural languages. These are the usual languages such as *English*, *Hindi*, *French*, *Tamil* which we employ for daily communication and in teaching, reading and writing.

Programming languages. These are the languages such as *C*, *Java*, *SML*, *Perl*, *Python* etc. that are used to write computer programs in.

Formal languages. These are languages which are generated by certain formation rules.

Meta-languages. These are usually natural languages used to explain concepts related to programming languages or formal languages. We are using *English* as the meta-language to describe and explain concepts in programming languages and formal languages.

In addition, we do have the concept of a **dialect** of a natural language or a programming language. For example the natural languages like Hindi, English and French do have several dialects. A dialect (in the case of natural languages) is a particular form of a language which is peculiar to a specific region or social group. *Creole* (spoken in Mauritius) is a dialect of French, Similarly *Brij*, *Awadhi* are dialects of Hindi. A dialect (in the case of programming languages) is a version of the programming language. There are many dialects of *C* and *C++*. Similarly *SML-NJ* and *poly-ML* are dialects of Standard ML. The notion of a dialect does not really exist for formal languages.

Closer home to what we are discussing, the language of regular expressions is a *formal language* which describes the rules for forming the words of a programming language. Each regular expression represents a finite or infinite set of words in the vocabulary of a programming language. We may think of the language of regular expressions also as a *functional programming language* for describing the vocabulary of a programming language. It allows us to generate words belonging to the vocabulary of a programming language

Any formally defined language also defines an algebraic system of operators applied on a *carrier set*. Every operator in any algebraic system has a pre-defined *arity* which refers to the number of operands it requires. In the case of regular expressions, the operators are concatenation and alternation are 2-ary operators (binary operators), whereas the Kleene closure and plus closure are 1-ary operators (unary). In addition the letters of the alphabet, which are constants may be considered to be operators of arity 0.

Simple Language of Regular Expressions

We consider a **simple language of regular expressions**. Assume a (finite) alphabet $\mathtt{A}$ of symbols. Each regular expression r denotes a set of strings $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the **language** specified by the regular expression r.

Symbol. For each symbol a in A, the regular expression **a** denotes the set $\{\mathtt{a}\}$.

(Con)catenation. For any two regular expressions r and s, $r.s$ or simply rs denotes the concatenation of the languages specified by r and s. That is,

$$\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$$

Epsilon and Alternation

Epsilon. ϵ denotes the language with a single element the **empty** string ε or ("").

$$\mathcal{L}(\epsilon) = \{\varepsilon\} = \{\texttt{""}\}$$

Alternation. Given any two regular expressions r and s, $r|s$ is the set union of the languages specified by the individual expressions r and s respectively.

$$\mathcal{L}(r \mid s) = \mathcal{L}(r) \cup \mathcal{L}(s)$$

Example $\mathcal{L}(\mathbf{Menelik}|\mathbf{Selassie}|\epsilon) = \{\texttt{Menelik}, \texttt{Selassie}, \varepsilon\}$.

String Repetitions

For any string x, we may use concatenation to create a string y with as many repetitions of x as we want, by defining repetitions by induction.

$$
\begin{aligned}
x^0 &= \text{""} \\
x^1 &= x \\
x^2 &= x.x \\
&\vdots \\
x^{n+1} &= x.x^n = x^n.x \\
&\vdots
\end{aligned}
$$

Then

$$\boxed{x^* = \{x^n \mid n \geq 0\}}$$

String Repetitions Example

Example. Let $x = \texttt{Selassie}$. Then

$$
\begin{aligned}
x^0 &= \text{''''} \\
x^1 &= \texttt{Selassie} \\
x^2 &= \texttt{SelassieSelassie} \\
&\vdots \\
x^5 &= \texttt{SelassieSelassieSelassieSelassieSelassie} \\
&\vdots
\end{aligned}
$$

Then x^* is the language consisting of all strings that are finite repetitions of the string $\texttt{Selassie}$

Language Iteration

The * operator can be extended to languages in the same way. For any *language* X, we may use concatenation to create a another language Y with as many repetitions of the strings in X as we want, by defining repetitions by induction. Hence if X is nonempty, then we have

$$\begin{array}{cc} X^0 = \{""\} & X^1 = X \\ X^2 = X.X & X^3 = X^2.X \\ \vdots & \vdots \end{array}$$

In general $X^{n+1} = X.X^n = X^n.X$ and

$$\boxed{X^* = \bigcup_{n \geq 0} X^n}$$

Language Iteration Example

Example 3.6 *Let* $X = \{\texttt{Mengesha}, \texttt{Gemechis}\}$. *Then*

$$
\begin{aligned}
X^0 &= \{\text{''''}\} \\
X^1 &= \{\texttt{Mengesha}, \texttt{Gemechis}\} \\
X^2 &= \{\texttt{MengeshaMengesha}, \texttt{GemechisMengesha}, \\
&\quad \texttt{MengeshaGemechis}, \texttt{GemechisGemechis}\} \\
X^3 &= \{\texttt{MengeshaMengeshaMengesha}, \texttt{GemechisMengeshaMengesha}, \\
&\quad \texttt{MengeshaGemechisMengesha}, \texttt{GemechisGemechisMengesha}, \\
&\quad \texttt{MengeshaMengeshaGemechis}, \texttt{GemechisMengeshaGemechis}, \\
&\quad \texttt{MengeshaGemechisGemechis}, \texttt{GemechisGemechisGemechis}\} \\
&\vdots
\end{aligned}
$$

Kleene Closure

Given a regular expression r, r^n specifies the n**-fold iteration** of the language specified by r.

Given any regular expression r, the **Kleene closure** of r, denoted r^* specifies the language $(\mathcal{L}(r))^*$.

In general

$$\boxed{r^* = r^0 \mid r^1 \mid \cdots \mid r^{n+1} \mid \cdots}$$

denotes an infinite union of languages.

Further it is easy to show the following identities.

$$\begin{aligned} r^* &= \epsilon | r.r^* && (1) \\ r^* &= (r^*)^* && (2) \end{aligned}$$

Plus Closure

The Kleene closure allows for *zero or more iterations* of a language. The $+$**-closure** of a language X denoted by X^+ and defined as

$$X^+ = \bigcup_{n>0} X^n$$

denotes *one or more iterations* of the language X.
Analogously we have that r^+ specifies the language $(\mathcal{L}(r))^+$.
Notice that for any language X, $X^+ = X.X^*$ and hence for any regular expression r we have

$$\boxed{r^+ = r.r^*}$$

We also have the identity (1)

$$\boxed{r^* = \epsilon \mid r^+}$$

Range Specifications

We may specify **ranges** of various kinds as follows.

- $[\mathbf{a-c}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$. Hence the expression of Question 3 may be specified as $[\mathbf{a-c}]^*$.
- Multiple ranges: $[\mathbf{a-c0-3}] = [\mathbf{a-c}] \mid [\mathbf{0-3}]$

Exercise 3.1

1. *If* $X = \emptyset$ *what are* X^0, X^n *for* $n > 0$ *and* X^**?*
2. *Try to understand what the regular expression for identifiers really specifies.*
3. *Modify the regular expression so that all identifiers start only with upper-case letters.*
4. *Give regular expressions to specify*
   - *real numbers in* fixed decimal point notation
   - *real numbers in* floating point notation
   - *real numbers in both* fixed decimal point notation *as well as* floating point notation.

Equivalence of Regular Expressions

Definition 3.7 *Let $REGEXP_{\mathtt{A}}$ denote the set of regular expressions over a a finite non-empty set of symbols $\mathtt{A}$ and let $r, s \in REGEXP_{\mathtt{A}}$. Then*

- $r \leq_{\mathtt{A}} r$ *if and only if* $\mathcal{L}(r) \subseteq \mathcal{L}(s)$ *and*
- *they are* **equivalent** *(denoted $r =_{\mathtt{A}} s$) if they specify the same language, i.e.*

$$r =_{\mathtt{A}} s \text{ if and only if } \mathcal{L}(r) = \mathcal{L}(s)$$

We have already considered various identities (e.g. (1)) giving the equivalence between different regular expressions.

Notes on bracketing and precedence of operators

In general regular expressions could be *ambiguous* (in the sense that the same expression may be interpreted to refer to different languages. This is especially so in the presence of

- multiple binary operators
- some unary operators used in prefix form while some others are used in post-fix form. The Kleene-closure and plus closure are operators in postfix form. We have not introduced any prefix unary operator in the language of regular expressions.

All expressions may be made unambiguous by specifying them in a fully parenthesised fashion. However, that leads to too many parentheses and is often hard to read. Usually rules for precedence of operators is defined and we may use the parentheses "(" and ")" to group expressions over-riding the precedence conventions of the language.

For the operators of regular expressions we will use the precedence convention that $|$ has a lower precedence than $.$ and that all unary operators have the highest precedence.

Example 3.8 *The language of arithmetic expressions over numbers uses the "BDMAS" convention that brackets have the highest precedence, followed by division and multiplication and the operations of addition and subtraction have the lowest precedence.*

Example 3.9 *The regular expression* $r.s|t.u$ *is ambiguous because we do not know beforehand whether it represents* $(r.s)|(t.u)$ *or* $r.(s|t).u$ *or even various other possibilities. By specifying that the operator* $|$ *has lower precedence than* $.$ *we are disambiguating the expression to mean* $(r.s)|(t.u)$*.*

Example 3.10 *The language of arithmetic expressions can also be extended to include the unary post-fix operation in which case an expression such as* $-a!$ *becomes ambiguous. It could be interpreted to mean either* $(-a)!$ *or* $-(a!)$*. In the absence of a well-known convention it is best adopt parenthesisation to disambiguate the expression.*

Besides the ambiguity created by multiple binary operators, there are also ambiguities created by the same operator and in deciding in what order two or more occurrences of the same operator need to be evaluated. A classic example is the case of subtraction in arithmetic expressions.

Example 3.11 *The arithmetic expression* $a - b - c$*, in the absence of any well-defined convention could be interpreted to mean either* $(a - b) - c$ *or* $a - (b - c)$ *and the two interpretations would yield different values in general. The problem does not exist for operators such addition and multiplication on numbers, because these operators are associative. Hence even though* $a + b + c$ *may be interpreted in two different ways, both interpretations yield identical values.*

Example 3.12 *Another non-associative operator in arithmetic which often leaves students confused is the exponentiation operator. Consider the arithmetic expression* a^{b^c}*. For* $a = 2$*,* $b = 3$*,* $c = 4$ *is this expression to be interpreted as* $a^{(b^c)}$ *or as* $(a^b)^c$*?*

Exercise 3.2

1. *For what regular expression* r *will* r^* *specify a* finite *set?*
2. *How many strings will be in the language specified by* $(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c})^n$*?*
3. *Give an informal description of the language specified by* $(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c})^*$*?*
4. *Give a regular expression which specifies the language* $\{\mathtt{a}^k \mid k > 100\}$*.*
5. *Simplify the expression* $r^*.r^*$*, i.e. give a simpler regular expression which specifies the same language.*
6. *Simplify the expression* $r^+.r^+$*.*

Nondeterministic Finite Automata (NFA)

Nondeterministic Finite Automata

A regular expression is useful in defining a *finite state automaton*. An automaton is a machine (a simple program) which can be used to recognize any valid lexical token of a language.

A **nondeterministic finite automaton (NFA)** N over a finite alphabet $\mathtt{A}$ consists of

- a finite set Q of **states**,
- an **initial** state $q_0 \in Q$,
- a finite subset $F \subseteq Q$ of states called the **final states** or **accepting states**, and
- a **transition relation** $\longrightarrow \subseteq Q \times (\mathtt{A} \cup \{\varepsilon\}) \times Q$.

What is nondeterminstic?

- The transition relation may be equivalently represented as a function

$$\longrightarrow : Q \times (\mathtt{A} \cup \{\varepsilon\}) \to 2^Q$$

that for each **source** state $q \in Q$ and symbol $\mathtt{a} \in \mathtt{A}$ associates a set of **target** states.

- It is non-deterministic because for a given source state and input symbol,
  - there may not be a unique target state, there may be more than one, or
  - the set of target states could be empty.
- Another source of non-determinism is the empty string ε.

Nondeterminism and Automata

- In general the automaton *reads* the input string from left to right.
- It reads each input symbol *only once* and executes a transition to new state.
- The ε transitions represent going to a new target state *without* reading any input symbol.
- The NFA may be nondeterministic because of
  - one or more ε transitions from the same source state *different* target states,
  - one or more transitions on the *same input* symbol from one source state to two or more different target states,
  - choice between executing a transition on an input symbol and a transition on ε (and going to different states).

Acceptance of NFA

- For any alphabet $\mathtt{A}$, $\mathtt{A}^*$ denotes the set of all (finite-length) strings of symbols from $\mathtt{A}$.
- Given a string $x = \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n \in \mathtt{A}^*$, an **accepting sequence** is a sequence of transitions

$$q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{\mathtt{a}_1}\xrightarrow{\varepsilon} \cdots q_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{\mathtt{a}_2} \cdots \xrightarrow{\varepsilon}\xrightarrow{\mathtt{a}_n}\xrightarrow{\varepsilon} \cdots q_n$$

  where $q_n \in F$ is an accepting state.
- Since the automaton is nondeterministic, it is also possible that there exists another sequence of transitions

$$q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{\mathtt{a}_1}\xrightarrow{\varepsilon} \cdots q_1' \xrightarrow{\varepsilon} \cdots \xrightarrow{\mathtt{a}_2} \cdots \xrightarrow{\varepsilon}\xrightarrow{\mathtt{a}_n}\xrightarrow{\varepsilon} \cdots q_n'$$

  where q_n' is not a final state.
- The automaton **accepts** x, if there is an accepting sequence for x.

Language of a NFA

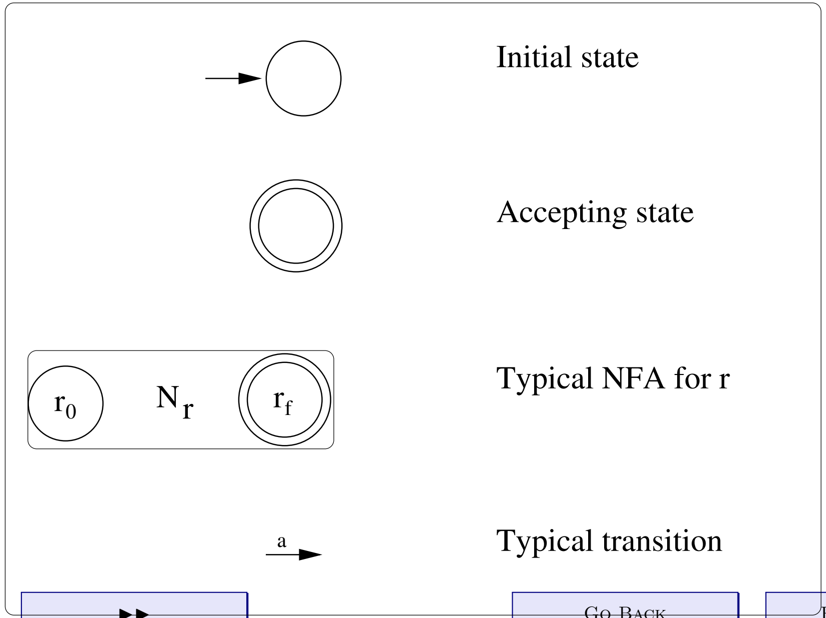
- The language **accepted** or **recognized** by a NFA is the set of strings that can be accepted by the NFA.
- $\mathcal{L}(N)$ is the language accepted by the NFA N.

Construction of NFAs

- We show how to construct an NFA to accept a certain language of strings from the regular expression specification of the language.
- The method of construction is by *induction on the structure* of the regular expression. That is, for each regular expression operator, we show how to construct the corresponding automaton assuming that the NFAs corresponding to individual components of expression have already been constructed.
- For any regular expression r the corresponding NFA constructed is denoted N_r. Hence for the regular expression $r|s$, we construct the NFA $N_{r|s}$ using the NFAs N_r and N_s as the building blocks.
- Our method requires only one initial state and one final state for each automaton. Hence in the construction of $N_{r|s}$ from N_r and N_s, the initial states and the final states of N_r and N_s are not initial or final unless explicitly used in that fashion.

Constructing NFA

- We show the construction only for the most *basic* operators on regular expressions.
- For any regular expression r, we construct a NFA N_r whose initial state is named r_0 and final state r_f.
- The following symbols show the various components used in the depiction of NFAs.



Regular Expressions to NFAs:1

We may also express the automaton in tabular form as follows:

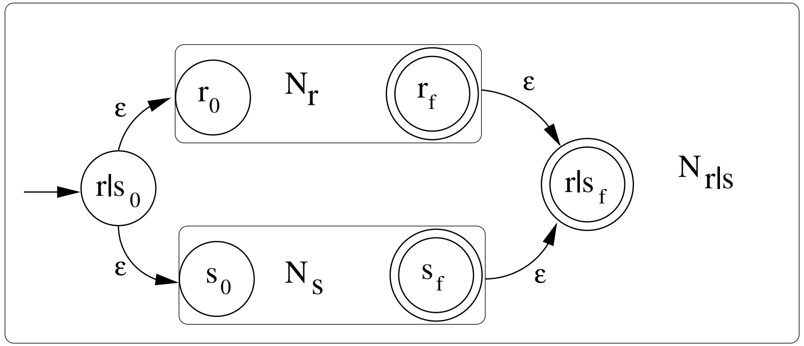
N_a	Input Symbol		
State	a	$\cdots$	ε
$\mathbf{a}_0$	$\{\mathbf{a}_f\}$	$\emptyset \cdots \emptyset$	$\emptyset$
$\mathbf{a}_f$	$\emptyset$	$\emptyset \cdots \emptyset$	$\emptyset$

Notice that all the cells except one have empty targets.

Regular Expressions to NFAs:2

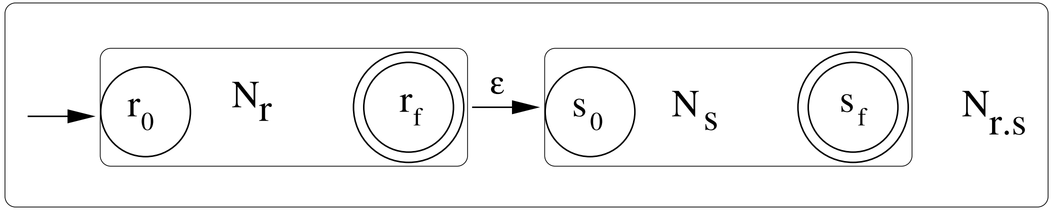
N_ε	Input Symbol		
State	a	$\cdots$	ε
ε_0	$\emptyset$	$\emptyset \cdots \emptyset$	$\{\varepsilon_f\}$
ε_f	$\emptyset$	$\emptyset \cdots \emptyset$	$\emptyset$

Regular Expressions to NFAs:3



$N_{r\|s}$	Input Symbol		
State	a	$\cdots$	ε
$r\|s_0$	$\emptyset$	$\cdots$	$\{r_0, s_0\}$
r_0	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_f	$\cdots$	$\cdots$	$\{r\|s_f\}$
s_0	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s_f	$\cdots$	$\cdots$	$\{r\|s_f\}$
$r\|s_f$	$\emptyset$	$\cdots$	$\emptyset$

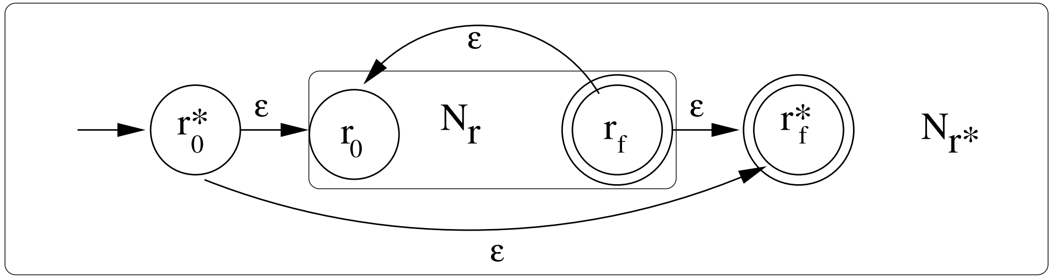
Regular Expressions to NFAs:4



$N_{r.s}$	Input Symbol		
State	a	$\cdots$	ε
r_0	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_f	$\cdots$	$\cdots$	$\{s_0\}$
s_0	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s_f	$\cdots$	$\cdots$	$\cdots$

Notice that the initial state of $N_{r.s}$ is r_0 and the final state is s_f in this case.

Regular Expressions to NFAs:5



N_{r^*}	Input Symbol		
State	a	$\cdots$	ε
r_0^*	$\emptyset$	$\cdots$	$\{r_0, r_f^*\}$
r_0	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_f	$\cdots$	$\cdots$	$\{r_0, r_f^*\}$
r_f^*	$\emptyset$	$\emptyset$	$\emptyset$

Regular expressions vs. NFAs

- It is obvious that for each regular expression r, the corresponding NFA N_r is correct *by construction* i.e.

$$\mathcal{L}(N_r) = \mathcal{L}(r)$$

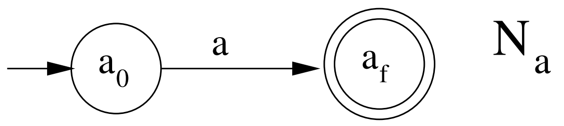
- Each regular expression operator
  - adds at most 2 new states and
  - adds at most 4 new transitions
- Every state of each N_r so constructed has
  - either 1 outgoing transition on a symbol from `A`
  - or at most 2 outgoing transitions on ε
- Hence N_r has at most $2|r|$ states and $4|r|$ transitions.

Example

We construct a NFA for the regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$.

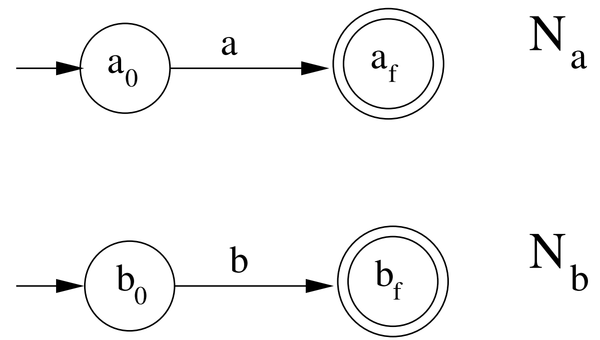
- Assume the alphabet $\mathtt{A} = \{\mathtt{a}, \mathtt{b}\}$.
- We follow the steps of the construction as given in Constructing NFA to Regular Expressions to NFAs:5
- For ease of understanding we use the regular expression itself (subscripted by 0 and f respectively) to name the two new states created by the regular expression operator.

Example:-6



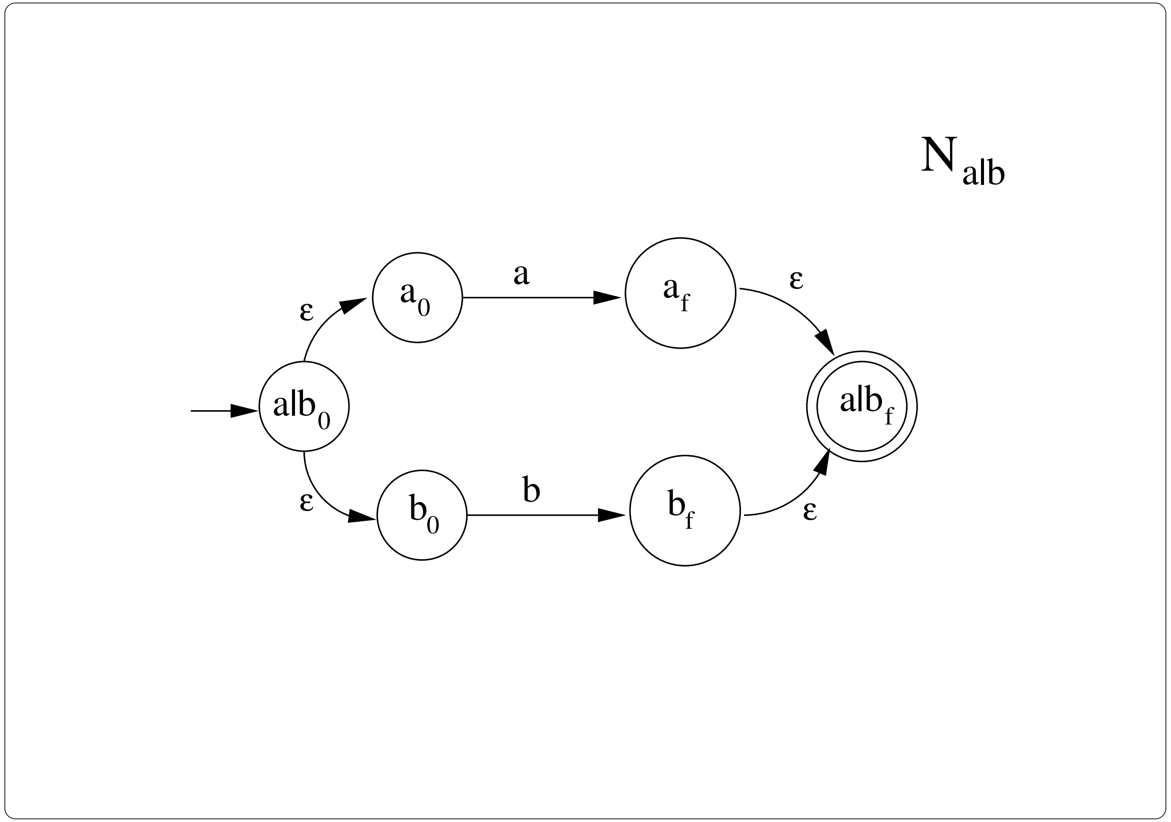
Steps in NFA for $\mathbf{(a|b)^*abb}$

Example:-5



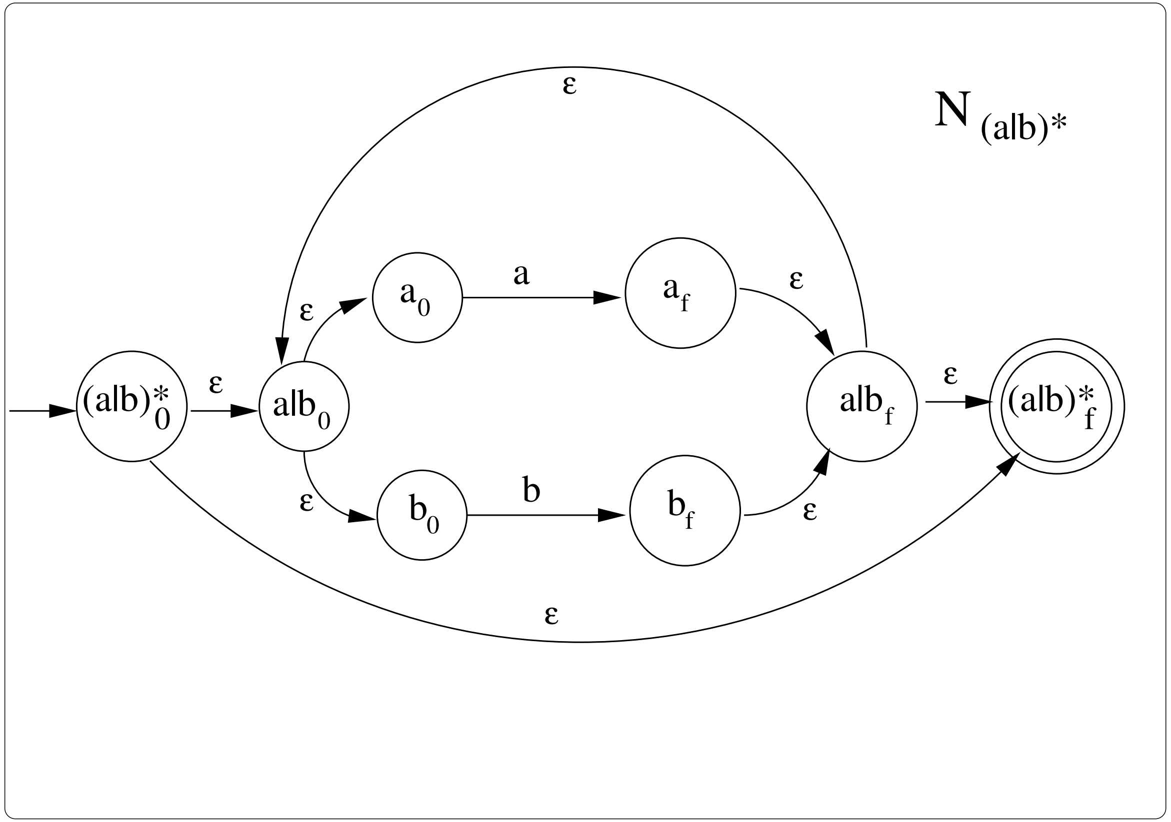
Steps in NFA for $\mathbf{(a|b)^*abb}$

Example:-4



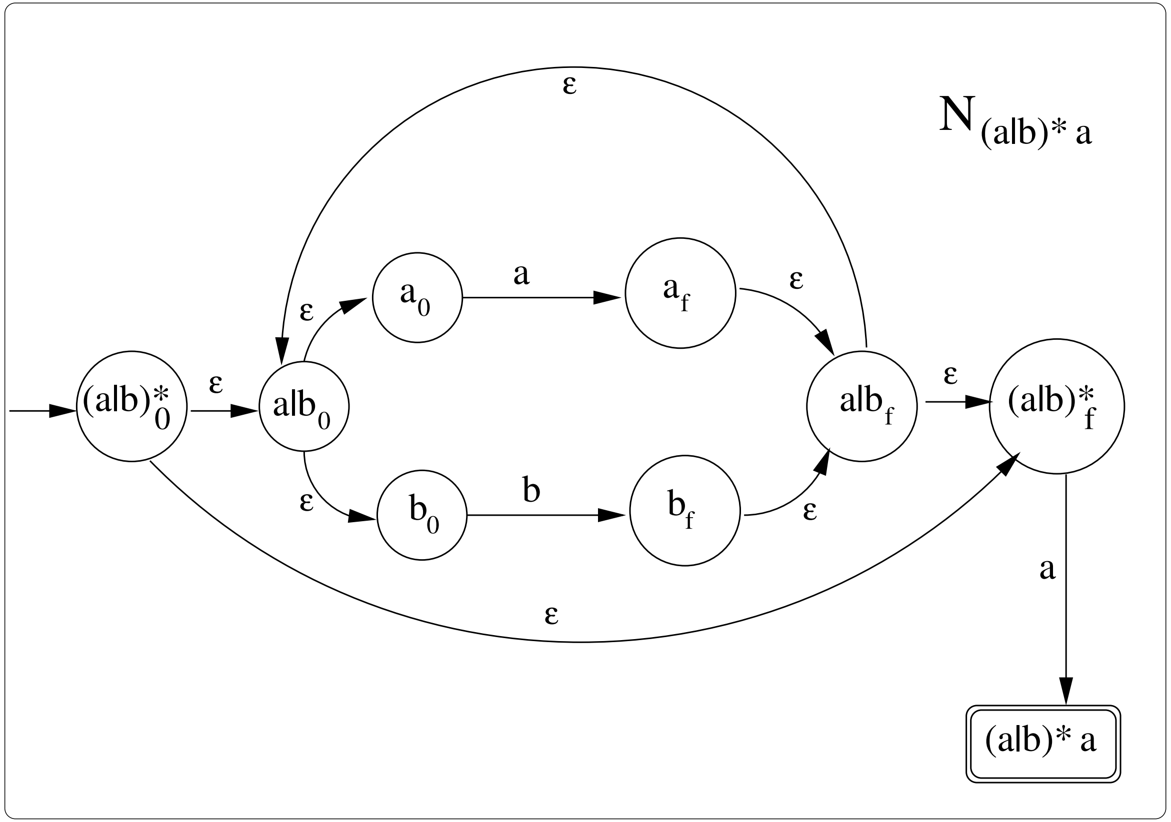
Steps in NFA for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Example:-3



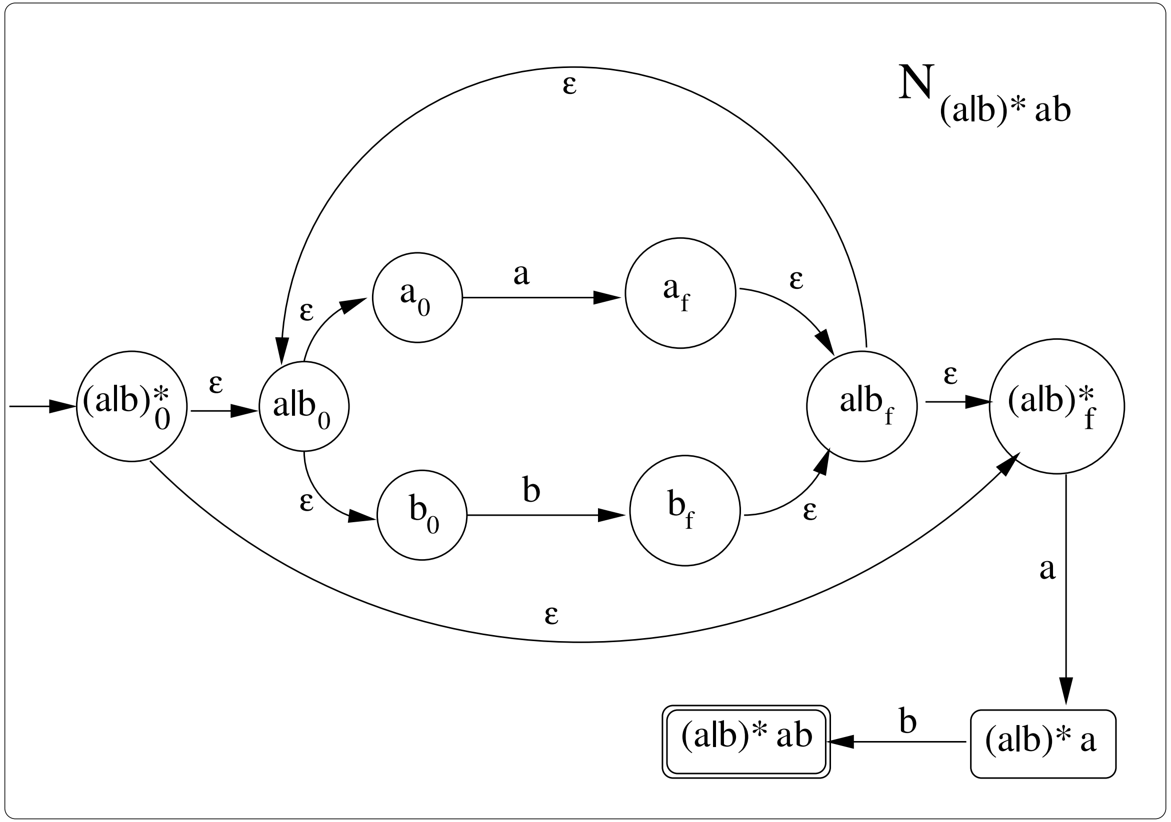
Steps in NFA for $\mathbf{(a|b)^*abb}$

Example:-2



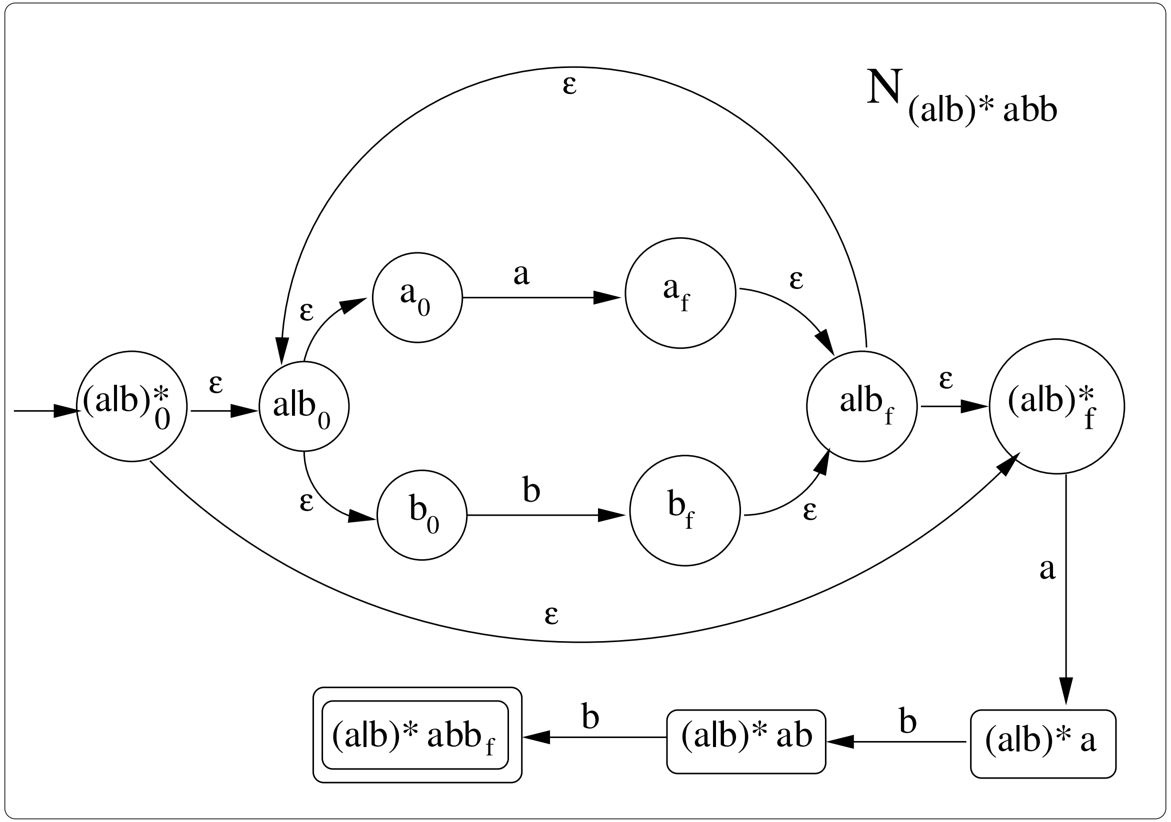
Steps in NFA for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Example:-1



Steps in NFA for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Example-final



Steps in NFA for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Exercise 3.3 *We have provided constructions for only the most basic operators on regular expressions. Here are some extensions you can attempt*

1. *Show how to construct a NFA for ranges and multiple ranges of symbols*
2. *Assuming N_r is a NFA for the regular expression r, how will you construct the NFA N_{r^+}.*
3. *Certain languages like Perl allow an operator like $r\{k, n\}$, where*

$$\mathcal{L}(r\{k, n\}) = \bigcup_{k \leq m \leq n} \mathcal{L}(r^m)$$

   Show how to construct $N_{r\{k,n\}}$ given N_r.

4. *Consider a new regular expression operator $\hat{}$ defined by $\mathcal{L}(\hat{}r) = \mathtt{A}^* - \mathcal{L}(r)$ What is the automaton $N_{\hat{}r}$ given N_r?*
5. *Perhaps out of sheer perversity or to simply confuse students, the UNIX operating system also allows the symbols "^" and "\$" to denote the beginning and the end of a line respectively. Consider the regular expression* $\hat{}(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\$$.

(a) *What is the language defined by the expression?*

(b) *Considering that "^" is overloaded, does it allow for the regular expression to define multiple different languages?*

(c) *Design an NFA which accepts some or all languages that the expression may denote.*

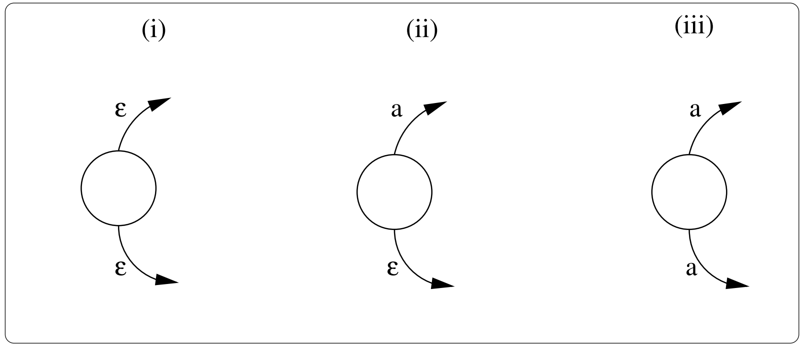
Scanning Using NFAs

Scanning and Automata

- Scanning is the only phase of the compiler in which every character of the source program is read
- The scanning phase therefore needs to be defined *accurately* and *efficiently*.
- *Accuracy* is achieved by regular expression specification of the tokens
- *Efficiency* implies that the input should not be read more than once.

Nondeterminism and Token Recognition

- The three kinds of nondeterminism in the NFA construction are depicted in the figure below.



(i) It is difficult to know which ε transition to pick without reading any further input

(ii) For two transitions on the same input symbol a it is difficult to know which of them would reach a final state on further input.

(iii) Given an input symbol a and an ε transition on the current state it is impossible to decide which one to take without looking at further input.

Nondeterministic Features

- In general it is impossible to recognize tokens in the presence of nondeterminism without *backtracking*.
- Hence NFAs are not directly useful for scanning because of the presence of nondeterminism.
- The nondeterministic feature of the construction of N_r for any regular expression r is in the ε transitions.
- The ε transitions in any automaton refer to the fact that no input character is consumed in the transition.
- *Backtracking* usually means algorithms involving them are very complex and hence inefficient.
- To avoid backtracking, the automaton should be made *deterministic*

From NFA to DFA

- Since the only source of nondeterminism in our construction are the ε, we need to eliminate them without changing the language recognized by the automaton.
- Two consecutive ε transitions are the same as one. In fact any number of ε transitions are the same as one. So as a first step we compute all finite sequences of ε transitions and collapse them into a single ε transition.
- Two states q, q' are equivalent if there are only ε transitions between them. This is called the **ε-closure** of states.

ε-Closure

Given a set T of states, then $T_\varepsilon = \varepsilon\text{-}closure(T)$ is the set of states which either belong to T or can be reached from states belonging to T only through a sequence of ε transitions.

Algorithm 3.1

ε-CLOSURE $(T) \stackrel{df}{=}$

$$\left\{\begin{array}{l}
\quad \textbf{Require: } T \subseteq Q \text{ of NFA } N = \langle Q, \mathtt{A} \cup \{\varepsilon\}, q_0, F, \longrightarrow\rangle \\
\quad \textbf{Ensures: } T_\varepsilon = \varepsilon\text{-}\mathrm{CLOSURE}(T) \\
T_\varepsilon := T; \\
\textbf{repeat} \\
\;\; T'_\varepsilon := T_\varepsilon; T_\varepsilon := T'_\varepsilon \cup \{q' \mid q' \notin T'_\varepsilon, \exists q \in T'_\varepsilon : q \xrightarrow{\varepsilon} q'\} \\
\textbf{until } T_\varepsilon = T'_\varepsilon
\end{array}\right.$$

Analysis of ε-Closure

- If $T = \emptyset$ then $T_\varepsilon = T$ in the first iteration.
- T_ε can only grow in size through each iteration
- The set T_ε cannot grow beyond the total set of states Q which is finite. Hence the algorithm always terminates for any NFA N.
- Time complexity: $O(|Q|)$.

Recognition using NFA

The following algorithm may be used to recognize a string using a NFA. In the algorithm we extend our notation for targets of transitions to include sets of sources. Thus

$$S \xrightarrow{a} = \{q' \mid \exists q \in S : q \xrightarrow{a} q'\}$$

and

$$\varepsilon\text{-}\textsc{Closure}(S \xrightarrow{a}) = \bigcup_{q' \in S \xrightarrow{a}} \varepsilon\text{-}\textsc{Closure}(q')$$

Recognition using NFA: Algorithm

Algorithm 3.2

$$\textsc{Accept}\ (N, x) \stackrel{df}{=}$$

$$\begin{cases} \quad \textbf{Require: } \text{NFA } N = \langle Q, \mathtt{A} \cup \{\varepsilon\}, q_0, F, \longrightarrow \rangle, \text{ a lexeme } x \\ \quad \textbf{Ensures: } \text{Boolean} \\ S := \varepsilon\text{-}\textsc{Closure}(q_0); a := \mathtt{nextchar}(x); \\ \textbf{while } a \neq end\_of\_string \\ \quad \textbf{do } \begin{cases} S := \varepsilon\text{-}\textsc{Closure}(S \xrightarrow{a}); \\ a := \mathtt{nextchar}(x) \end{cases} \\ \textbf{return } (S \cap F \neq \emptyset) \end{cases}$$

Analysis of Recognition using NFA

- Even if ε-closure is computed as a call from within the algorithm, the time taken to recognize a string is bounded by $O(|x|.|Q_{N_r}|)$ where $|Q_{N_r}|$ is the number of states in N_r.
- The space required for the automaton is at most $O(|r|)$.
- Given that ε-closure of each state can be pre-computed knowing the NFA, the recognition algorithm can run in time linear in the length of the input string x i.e. $O(|x|)$.
- Knowing that the above algorithm is deterministic once ε-closures are pre-computed one may then work towards a *Deterministic* automaton to reduce the space required.

Conversion of NFAs to DFAs

Deterministic Finite Automata

- A **deterministic finite automaton (DFA)** is a NFA in which
  1. there are no transitions on ε and
  2. $\longrightarrow$ yields a *exactly one* target state for each source state and symbol from $\mathtt{A}$ i.e. the transition relation is no longer a relation but a *total function*[a]

$$\delta : Q \times \mathtt{A} \rightarrow Q$$

- Clearly if every regular expression had a DFA which accepts the same language, all backtracking could be avoided.

[a] Also in the case of the NFA the relation $\longrightarrow$ may not define a transition from every state on every letter

Transition Tables of NFAs

We may think of a finite-state automaton as being defined by a 2-dimensional table of size $|Q| \times |A|$ in which for each state and each letter of the alphabet there is a set of possible *target* states defined. In the case of a non-deterministic automaton,

1. for each state there could be ε transitions to
   (a) a set consisting of a single state or
   (b) a set consisting of more than one state.
2. for each state q and letter a, there could be
   (a) an empty set of target states or
   (b) a set of target states consisting of a single state or
   (c) a set of target states consisting of more than one state

Transition Tables of DFAs

In the case of a deterministic automaton

1. there are no ε transitions, and
2. for each state q and letter a
   (a) either there is no transition in the NFA (in which case we add a new "sink" state which is a non-accepting state)
   (b) or there is a transition to a unique state q'.

The recognition problem for the same language of strings becomes simpler and would work faster (it would have no back-tracking) if the NFA could be converted into a DFA accepting the same language.

NFA to DFA

Let $N = \langle Q_N, \mathtt{A} \cup \{\varepsilon\}, s_N, F_N, \longrightarrow_N \rangle$ be a NFA . We would like to construct a DFA $D = \langle Q_D, \mathtt{A}, s_D, F_D, \longrightarrow_D \rangle$ where

- Q_D the set of states of the DFA
- $\mathtt{A}$ the alphabet (notice there is no ε),
- $s_D \in Q_D$ the start state of the DFA,
- F_D the final or accepting states of the DFA and
- $\delta_D : Q_D \times \mathtt{A} \longrightarrow Q_D$ the transition function of the DFA.

We would like $\mathcal{L}(N) = \mathcal{L}(D)$

The Subset Construction

Non-determinism .

ε-closure .

Subsets of NFA states .

Acceptance .

The Subset Construction: Non-determinism

Non-determinism A major source of non-determinism in NFAs is the presence of ε transitions. The use of ε-CLOSURE creates a cluster of "similar" states[a].

ε-closure .

Subsets of NFA states .

Acceptance .

[a] Two states are "similar" if they are reached from the start state by the same string of symbols from the alphabet

The Subset Construction: ε-closure

Non-determinism .

ε-closure . The ε-closure of each NFA state is a set of NFA states with "similar" behaviour, since they make their transitions on the same input symbols though with different numbers of εs.

Subsets of NFA states .

Acceptance .

The Subset Construction: Subsets of NFA states

Non-determinism .

ε-closure .

Subsets of NFA states . Each state of the DFA refers to a *subset of states of the NFA* which exhibit "similar" behaviour. Similarity of behaviour refers to the fact that they accept the same input symbols. The behaviour of two different NFA states may not be "identical" because they may have different numbers of ε transitions for the same input symbol.

Acceptance

The Subset Construction: Acceptance

Non-determinism .

ε-closure .

Subsets of NFA states .

Acceptance . Since the notion of acceptance of a string by an automaton, implies finding an accepting sequence even though there may be other *non-accepting sequences*, the non-accepting sequences may be ignored and those non-accepting states may be clustered with the accepting states of the NFA. So two different states reachable by the same sequence of symbols may be also thought to be similar.

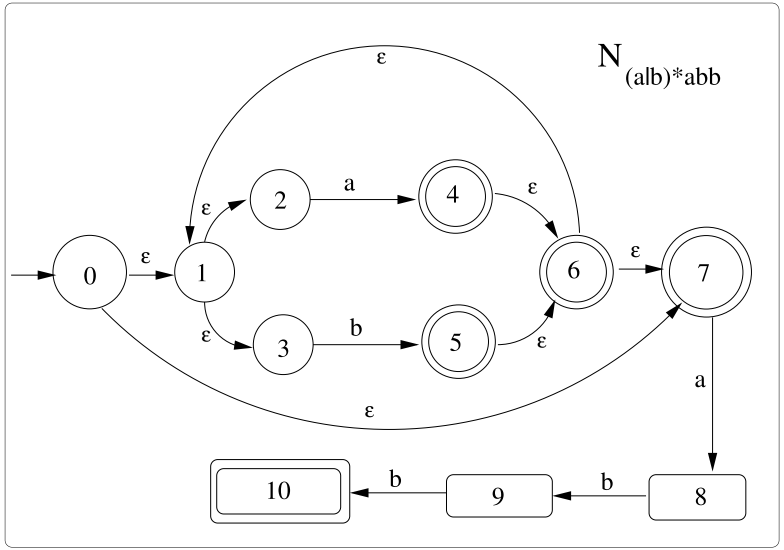
Algorithm 3.3

$\text{NFA}\textsc{to}\text{DFA}\ (N) \stackrel{df}{=}$

$$\left\{\begin{array}{l}
\quad \textbf{Requires: } \text{NFA } N = \langle Q_N, \mathtt{A} \cup \{\varepsilon\}, s_N, F_N, \longrightarrow_N \rangle \\
\quad \textbf{Yields: } \text{DFA } D = \langle Q_D, \mathtt{A}, s_D, F_D, \delta_D \rangle \text{ with } \mathcal{L}(N) = \mathcal{L}(D) \\
s_D := \varepsilon\text{-}\textsc{Closure}(\{s_N\}); Q_D := \{s_D\}; F_D := \emptyset; \delta_D := \emptyset; \\
U := \{s_D\} \quad \textbf{Note: } U \text{ is the set of unvisited states of } D \\
\textbf{while } U \neq \emptyset \\
\textbf{do} \left\{\begin{array}{l}
\textit{Choose any } q_D \in U; U := U - \{q_D\}; \quad \textbf{Note: } q_D \subseteq Q_N \\
\textbf{for each } a \in \mathtt{A} \\
\textbf{do} \left\{\begin{array}{l}
q'_D := \varepsilon\text{-}\textsc{Closure}(q_D \xrightarrow{a}_N); \delta_D(q_D, a) := q'_D \\
\textbf{if } q'_D \cap F_N \neq \emptyset \\
\quad \textbf{then } F_D := F_D \cup \{q'_D\}; \\
\textbf{if } q'_D \notin Q_D \\
\quad \textbf{then } \left\{\begin{array}{l} Q_D := Q_D \cup \{q'_D\}; \\ U := U \cup \{q'_D\} \end{array}\right.
\end{array}\right.
\end{array}\right.
\end{array}\right.$$

Example-NFA

Consider the NFA constructed for the regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$.



and apply the NFA to DFA construction algorithm

Determinising

$N_{(\mathbf{a}|\mathbf{b})^*\mathbf{abb}}$ $D_{(\mathbf{a}|\mathbf{b})^*\mathbf{abb}}$

$EC_0 = \varepsilon\text{-}\textsc{Closure}(0) = \{0, 1, 2, 3, 7\}$

$2 \xrightarrow{\texttt{a}}_N 4$ and $7 \xrightarrow{\texttt{a}}_N 8$. So $EC_0 \xrightarrow{\texttt{a}}_D \varepsilon\text{-}\textsc{Closure}(4, 8) = EC_{4,8}$. Similarly

$EC_0 \xrightarrow{\texttt{b}}_D \varepsilon\text{-}\textsc{Closure}(5) = EC_5$

$EC_{4,8} = \varepsilon\text{-}\textsc{Closure}(4, 8) = \{4, 6, 7, 1, 2, 3, 8\}$

$EC_5 = \varepsilon\text{-}\textsc{Closure}(5) = \{5, 6, 7, 1, 2, 3\}$

$EC_5 \xrightarrow{\texttt{a}}_D \varepsilon\text{-}\textsc{Closure}(4, 8) = EC_{4,8}$ and $EC_5 \xrightarrow{\texttt{b}}_D \varepsilon\text{-}\textsc{Closure}(5)$

$EC_{4,8} \xrightarrow{\texttt{a}}_D \varepsilon\text{-}\textsc{Closure}(4, 8) = EC_{4,8}$ and $EC_{4,8} \xrightarrow{\texttt{b}}_D \varepsilon\text{-}\textsc{Closure}(5, 9) = EC_{5,9}$

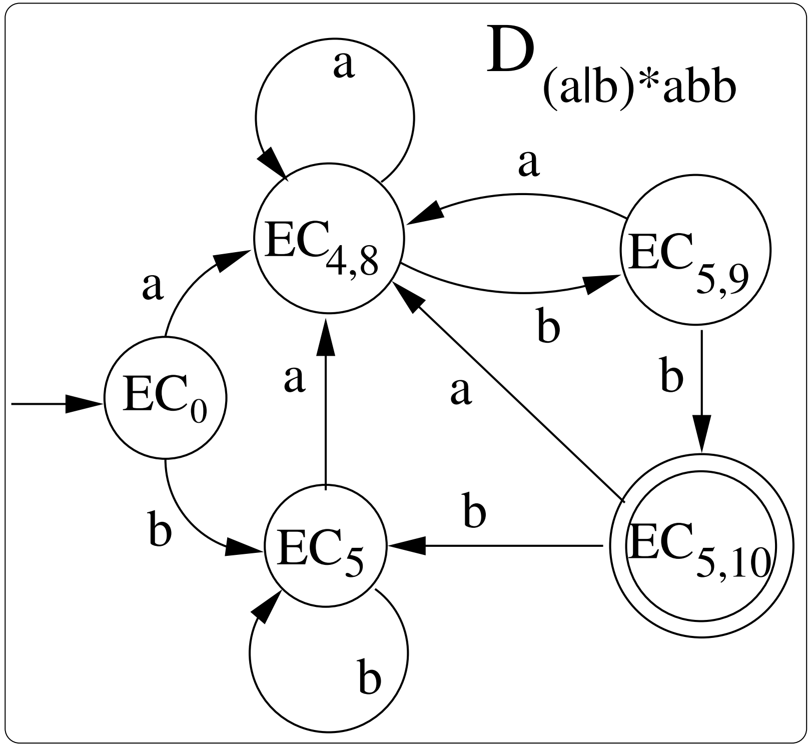
$EC_{5,9} = \varepsilon\text{-}\textsc{Closure}(5, 9) = \{5, 6, 7, 1, 2, 3, 9\}$

$EC_{5,9} \xrightarrow{\texttt{a}}_D \varepsilon\text{-}\textsc{Closure}(4, 8) = EC_{4,8}$ and $EC_{5,9} \xrightarrow{\texttt{b}}_D \varepsilon\text{-}\textsc{Closure}(5, 10) = EC_{5,10}$

$EC_{5,10} = \varepsilon\text{-}\textsc{Closure}(5, 10) = \{5, 6, 7, 1, 2, 3, 10\}$

$EC_{5,10} \xrightarrow{\texttt{a}}_D \varepsilon\text{-}\textsc{Closure}(4, 8)$ and $EC_{5,10} \xrightarrow{\texttt{b}} \varepsilon\text{-}\textsc{Closure}(5)$

Final DFA



$D_{(\mathbf{a}\vert\mathbf{b})^*\mathbf{abb}}$	Input Symbol	
State	a	b
EC_0	$EC_{4,8}$	EC_5
$EC_{4,8}$	$EC_{4,8}$	$EC_{5,9}$
EC_5	$EC_{4,8}$	EC_5
$EC_{5,9}$	$EC_{4,8}$	$EC_{5,10}$
$EC_{5,10}$	$EC_{4,8}$	EC_5

Recognition using DFA

The following algorithm may be used to recognize a string using a DFA. Compare it with the algorithm for recognition using an NFA.

Algorithm 3.4

$$\textsc{Accept}\ (D, x) \stackrel{df}{=}$$

$$\left\{\begin{array}{l} \quad \textbf{Requires: } \text{DFA } D = \langle Q, \mathtt{A}, q_0, F, \delta\rangle, \text{ a lexeme } x \in \mathtt{A}^* \\ \quad \textbf{Ensures: } \text{Boolean} \\ S := q_0; a := \mathtt{nextchar}(x); \\ \textbf{while } a \neq end\_of\_string \\ \quad \textbf{do } \left\{\begin{array}{l} S := \delta(S, a); \\ a := \mathtt{nextchar}(x) \end{array}\right. \\ \textbf{return } (S \in F) \end{array}\right.$$

Analysis of Recognition using DFA

- The running time of the algorithm is $O(|x|)$.
- The space required for the automaton is $O(|Q|.|\mathtt{A}|)$.

DFAs and Scanners

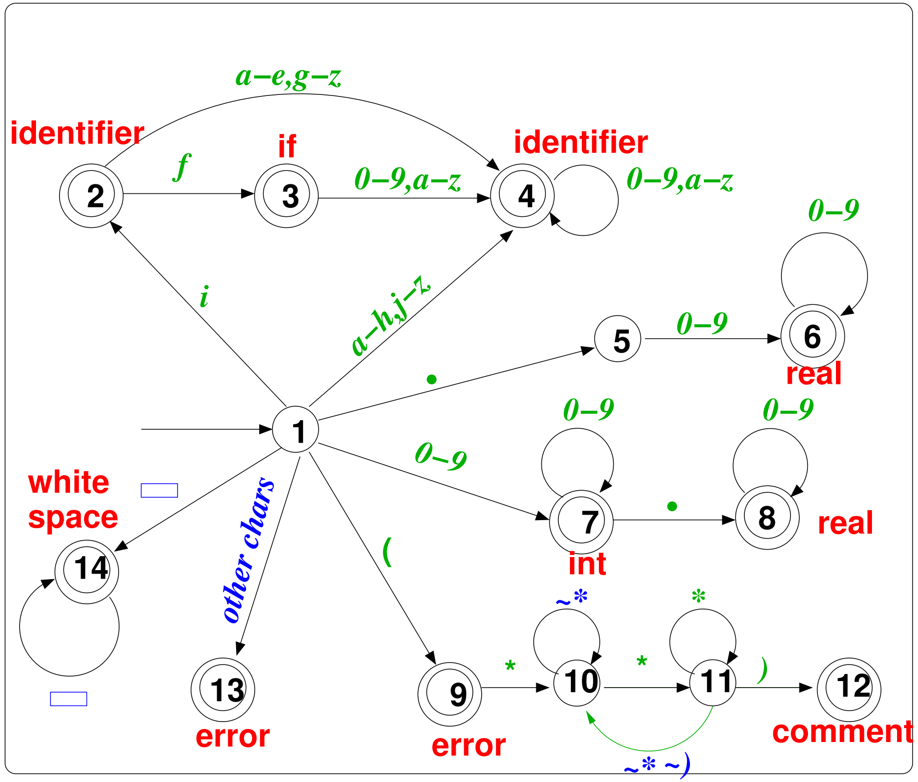
> In theory there is no difference between theory and practice. In practice there is.

A scanner differs from a simple DFA in several ways. Most importantly,

- A scanner is a DFA with outputs. It needs to output a token or spit out an error message and then proceed to the next lexeme.
- It is usually not much use minimising the number of states of a scanner, since it needs to classify based on the final state it reaches.
- In practice, the act of rejecting with an error message also requires accepting the whole lexeme. A scanner actually accepts the entire language $(\mathtt{A} - whitespaces)^{*}$[a].

[a] In the case of Python it needs to accept even whitespaces, count them and classify them as belonging to some nesting level.

Scanning With output



The Big Picture

DFA vis-a-vis Scanner: Practice

- A DFA simply accepts or rejects a lexeme. A scanner needs to recognize and classify every token and lexeme.
- A DFA may simply reach a non-accepting state in case of an unrecognizable lexeme. A scanner on the other hand needs to accept the lexeme and raise an error and proceed to the next lexeme.
- Where a token is allowed as a prefix of another (see the case of "if") scanners choose the longest lexeme[a] that is an identifiable token.
- DFAs are often "minimised" to collapse all accepting states into one accepting state. This is not desirable in the case of a scanner since tokens need to be classified separately based on the accepting states.

[a]Not true of FORTRAN

Exercise 3.4

1. *Write a regular expression to specify all numbers in binary form that are multiples of of* 4.
2. *Write regular expressions to specify all numbers in binary form that are* not *multiples of* 4.
3. *Each comment in the C language*
   - *begins with the characters "//" and ends with the newline character, or*
   - *begins with the characters "/*" and ends with "*/" and may run across several lines.*

   (a) *Write a regular expression to recognize comments in the C language.*

   (b) *Transform the regular expression into a NFA.*

   (c) *Transform the NFA into a DFA.*

   (d) *Explain why most programming languages do not allow nested comments.*

   (e) **modified C comments.** *If the character sequences "//", "/*" and "*/" are allowed to appear in 'quoted' form as "'//'", "'/*'" and "'*/'" respectively within a C comment, then give*

      i. *a modified regular expression for C comments*

      ii. *a NFA for these modified C comments*

iii. *a corresponding DFA for modified C comments*

4. *Many systems such as Windows XP and Linux recognize commands, filenames and folder names by the their shortest unique prefix. Hence given the 3 commands* **chmod**, **chgrp** *and* **chown**, *their shortest unique prefixes are respectively* **chm**, **chg** *and* **cho**. *A user can type the shortest unique prefix of the command and the system will automatically complete it for him/her.*

   (a) *Draw a DFA which recognizes all prefixes that are at least as long as the shortest unique prefix of each of the above commands.*

   (b) *Suppose the set of commands also includes two more commands* **cmp** *and* **cmpdir**, *state how you will include such commands also in your DFA where one command is a prefix of another.*

4. Parsing or Syntax Analysis

4.1. Grammars

Parsing Or Syntax Analysis

Generating a Language

Consider the DFA constructed earlier to accept the language defined by the regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$. We rename the states for convenience.

$D_{(\mathbf{a}\|\mathbf{b})^*\mathbf{abb}}$	Input	
State	a	b
S	A	B
A	A	C
B	A	B
C	A	D
D	A	B

We begin by rewriting each of the transitions as follows.

Production rules

$$
\begin{aligned}
S &\longrightarrow aA \mid bB \\
A &\longrightarrow aA \mid bC \\
B &\longrightarrow aA \mid bB \\
C &\longrightarrow aA \mid bD \\
D &\longrightarrow aA \mid bB \mid \varepsilon
\end{aligned}
$$

and think of each of the symbols S, A, B, C, D as generating symbols and thus producing (rather than consuming strings). For example, the strings abb and $aabbabb$ are generated by the above production rules as follows.

$$S \Rightarrow aA \Rightarrow abC \Rightarrow abbD \Rightarrow abb$$

$$
\begin{aligned}
&S \Rightarrow aA \Rightarrow aaA \Rightarrow aabC \Rightarrow aabbD \\
&\Rightarrow aabbaA \Rightarrow aabbabC \Rightarrow aabbabbD \Rightarrow aabbabb
\end{aligned}
$$

Formal languages: Definition, Recognition, Generation

There are three different processes used in dealing with a formal language.

Definition : Regular expressions is a formal (functional programming) language used to define or specify a formal language of tokens.

Recognition : Automata are the standard mechanism used to recognize words/phrases of a formal language. An automaton is used to determine whether a given word/phrase is a member of the formal language defined in some other way.

Generation : Grammars are used to define the generation of the words/phrases of a formal language.

Non-regular language

Consider the following two languages over an alphabet $A = \{a, b\}$.

$$R = \{a^n b^n | n < 100\}$$
$$P = \{a^n b^n | n > 0\}$$

- R may be finitely represented by a regular expression (even though the actual expression is very long).
- However, P cannot actually be represented by a regular expression
- A regular expression is not powerful enough to represent languages which require parenthesis matching to arbitrary depths.
- All high level programming languages require an underlying language of expressions which require parentheses to be nested and matched to arbitrary depth.

Grammars

Definition 4.1 *A* **grammar** $G = \langle N, T, P, S\rangle$ *consists of*

- *a set* N *of* **nonterminal** *symbols, or* **variables**,
- *a* **start** *symbol* $S \in N$,
- *a set* T *of* **terminal symbols** *or the* **alphabet**,
- *a set* P *of* **productions** *or* **rewrite rules** *where each rule is of the form* $\alpha \to \beta$ *for* $\alpha, \beta \in (N \cup T)^*$

Definition 4.2 *Given a grammar* $G = \langle N, T, P, S\rangle$, *any* $\alpha \in (N \cup T)^*$ *is called a* **sentential form**. *Any* $x \in T^*$ *is called a* **sentence**[a].

Note. Every sentence is also a sentential form.

[a] some authors call it a **word**. However we will reserve the term word to denote the tokens of a programming language.

Grammars: Notation

- Upper case roman letters $(A, B, \ldots, X, Y$, etc.$)$ denote nonterminals.
- Final upper case roman letters $(X, Y, Z$ etc.$)$ may also be used as meta-variables which denote arbitrary non-terminal symbols of a grammar.
- Initial lower case roman letters $(a, b, c$ etc.$)$ will be used to denote terminal symbols.
- Lower case greek letters $(\alpha,\ \beta$ etc.$)$ denote sentential forms (or even sentences).
- Final lower case letters $(u, v, \ldots, x, y, z$ etc.$)$ denote only sentences.
- In each case the symbols could also be decorated with sub-scripts or super-scripts.

Context-Free Grammars: Definition

Definition 4.3 *A* **grammar** $G = \langle N, T, P, S\rangle$ *is called* **context-free** *if each production is of the form* $X \longrightarrow \alpha$, *where*

- $X \in N$ *is a nonterminal and*
- $\alpha \in (N \cup T)^*$ *is a sentential form.*
- *The production is* **terminal** *if* α *is a sentence*

CFG: Example 1

$G = \langle \{S\}, \{a, b\}, P, S \rangle$, where $S \longrightarrow ab$ and $S \longrightarrow aSb$ are the only productions in P.

Derivations look like this:

- $$S \Rightarrow ab$$

- $$S \Rightarrow aSb \Rightarrow aabb$$

- $$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

- $$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$$

The first three derivations are complete while the last one is partial

Derivations

Definition 4.4 *A* **(partial) derivation** *(of length $n \in \mathbb{N}$) in a context-free grammar is a finite sequence of the form*

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \alpha_n \tag{3}$$

where each $\alpha_i \in (N \cup T)^*$ $(0 \leq i \leq n$ $)$ is a **sentential form** *where $\alpha_0 = S$ and α_{i+1} is obtained by applying a production rule to a non-terminal symbol in α_i for $0 \leq i < n$.*

Notation. $S \Rightarrow^* \alpha$ denotes that there exists a derivation of α from S.

Definition 4.5 *The derivation (3) is* **complete** *if $\alpha_n \in T^*$ i.e. α_n is a sentence. Then α_n is said to be a sentence* **generated** *by the grammar.*

Language Generation

Definition 4.6 *The language* **generated** *by a grammar* G *is the set of sentences that can be generated by* G *and is denoted* $\mathcal{L}(G)$.

Example 4.7 $\mathcal{L}(G)$, *the* **language** *generated by the grammar* G *is* $\{a^n b^n | n > 0\}$. *Prove using induction on the length of derivations.*

Regular Grammars

Definition 4.8 *A production rule of a context-free grammar is*

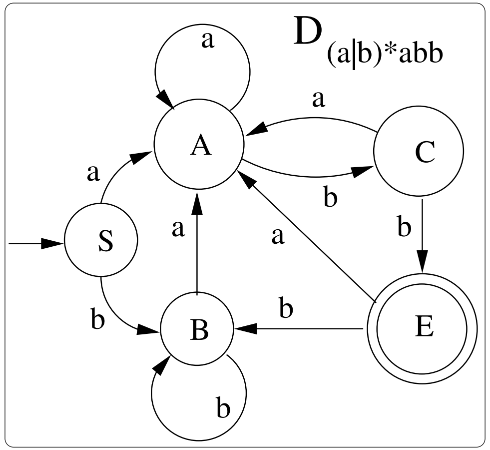
Right Linear: *if it is of the form* $X \longrightarrow a$ *or* $X \longrightarrow aY$

Left Linear: *if it is of the form* $X \longrightarrow a$ *or* $X \longrightarrow Ya$

where $a \in T$ *and* $X, Y \in N$.

Definition 4.9 *A* **regular** *grammar is a context-free grammar whose productions are either only* **right linear** *or only* **left linear**.

DFA to Regular Grammar



$D_{\mathbf{(a\|b)^*abb}}$	Input		RLG
State	a	b	Rules
S	A	B	$S \to aA\|bB$
A	A	C	$A \to aA\|bC$
B	A	B	$B \to aA\|bB$
C	A	E	$C \to aA\|bE\|b$
E	A	C	$E \to aA\|bC$

Consider the DFA with the states renamed as shown above. We could easily convert the DFA to a right linear grammar which generates the language accepted by the DFA.

CFG: Empty word

$G = \langle \{S\}, \{a, b\}, P, S \rangle$, where $S \longrightarrow SS \mid aSb \mid \varepsilon$
generates all sequences of matching nested parentheses, including the empty word ε.

A leftmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \ldots$$

A rightmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \ldots$$

Other derivations might look like *God alone knows what!*

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow \ldots$$

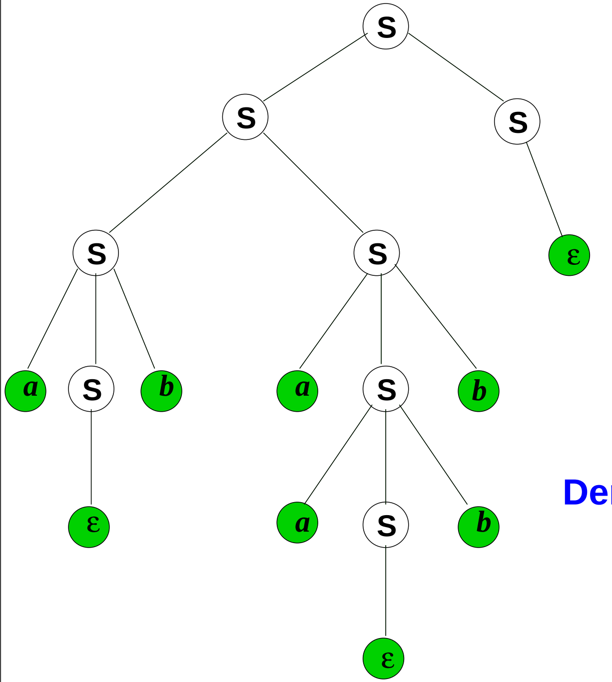
Could be quite confusing!

CFG: Derivation trees 1

Derivation sequences

- put an artificial order in which productions are fired.
- instead look at **trees** of derivations in which we may think of productions as being fired in **parallel**.
- There is then no highlighting in red to determine which copy of a nonterminal was used to get the next member of the sequence.
- Of course, generation of the empty word ε must be shown explicitly in the tree.

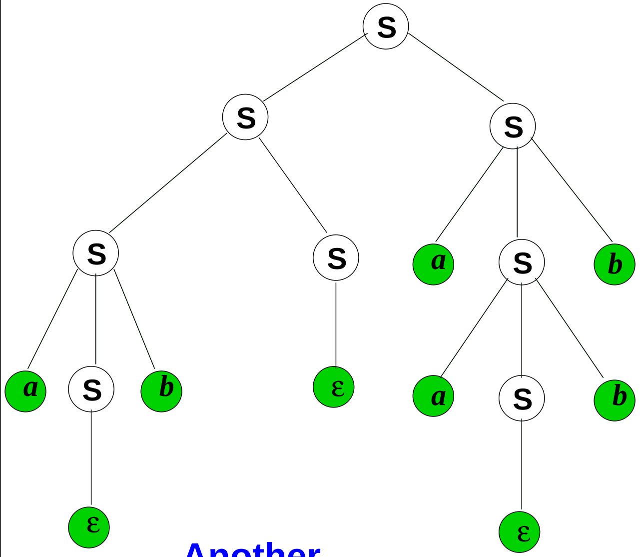
CFG: Derivation trees 2



Derivation tree of

abaabb

CFG: Derivation trees 3

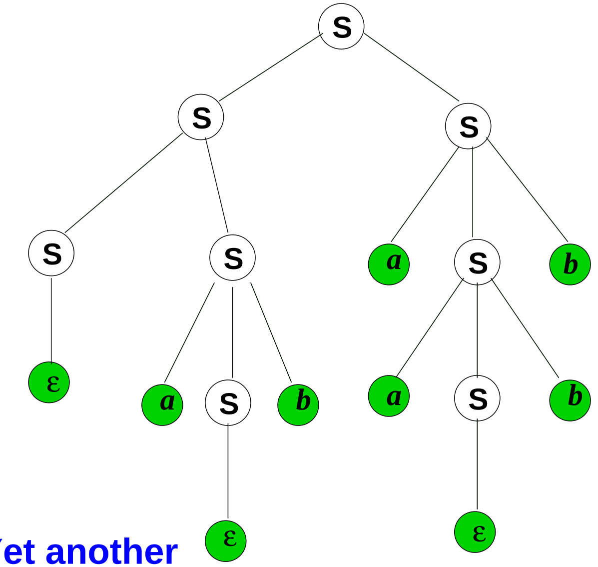


Another

Derivation tree of

abaabb

CFG: Derivation trees 4



Yet another Derivation tree of *abaabb*

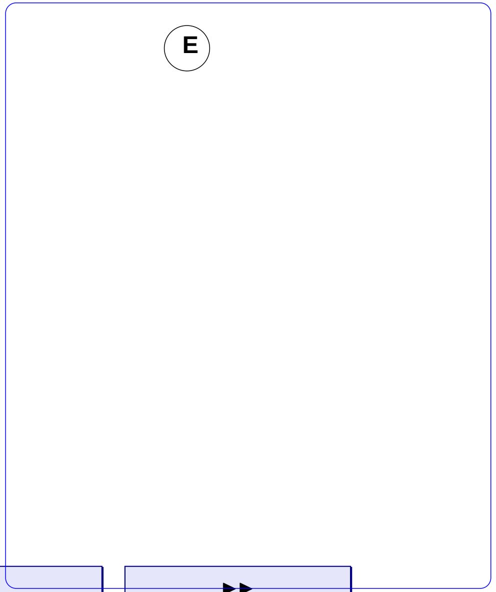
Ambiguity Disambiguation

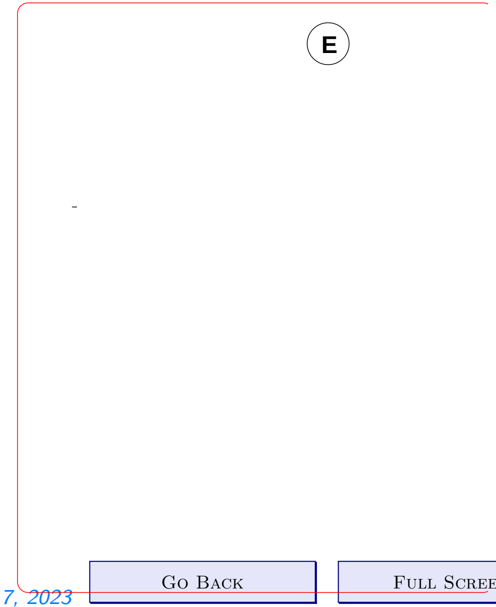
Ambiguity: 1

$G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$ where P_1 consists of the following productions.

$$
\begin{array}{lcl}
E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.



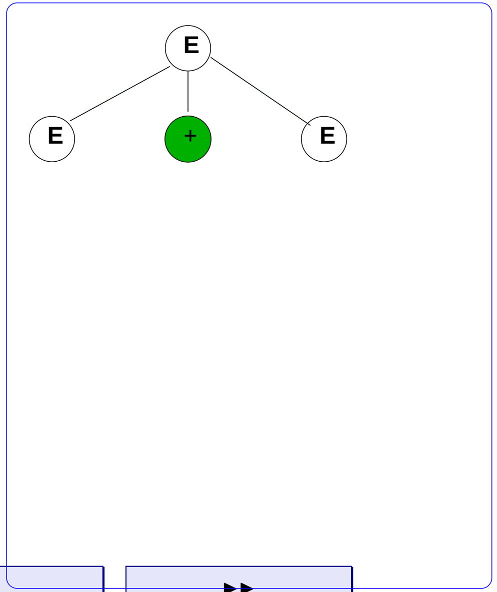


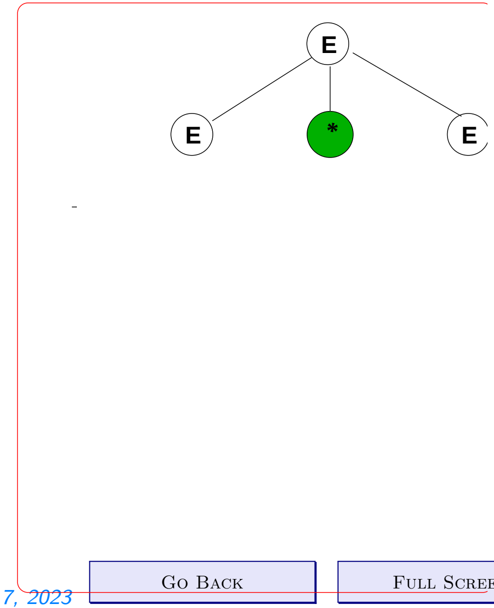
Ambiguity: 2

$G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$ where P_1 consists of the following productions.

$$\begin{array}{lcl} E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\ I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\ C & \rightarrow & \mathbf{4} \end{array}$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.



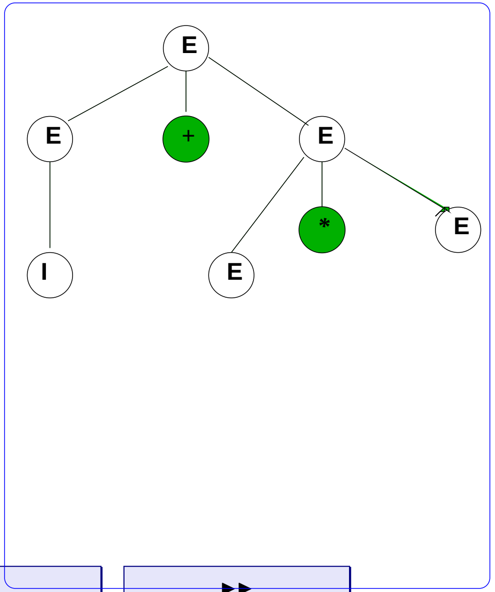


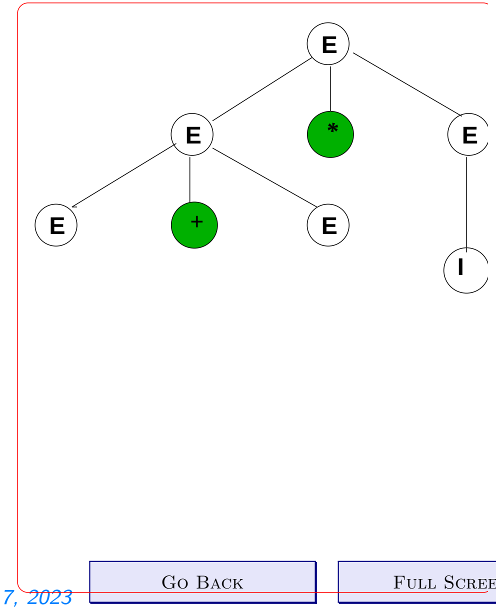
Ambiguity: 3

$G_1 = \langle\{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\}\rangle$ where P_1 consists of the following productions.

$$\begin{array}{rcl} E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\ I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\ C & \rightarrow & \mathbf{4} \end{array}$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.



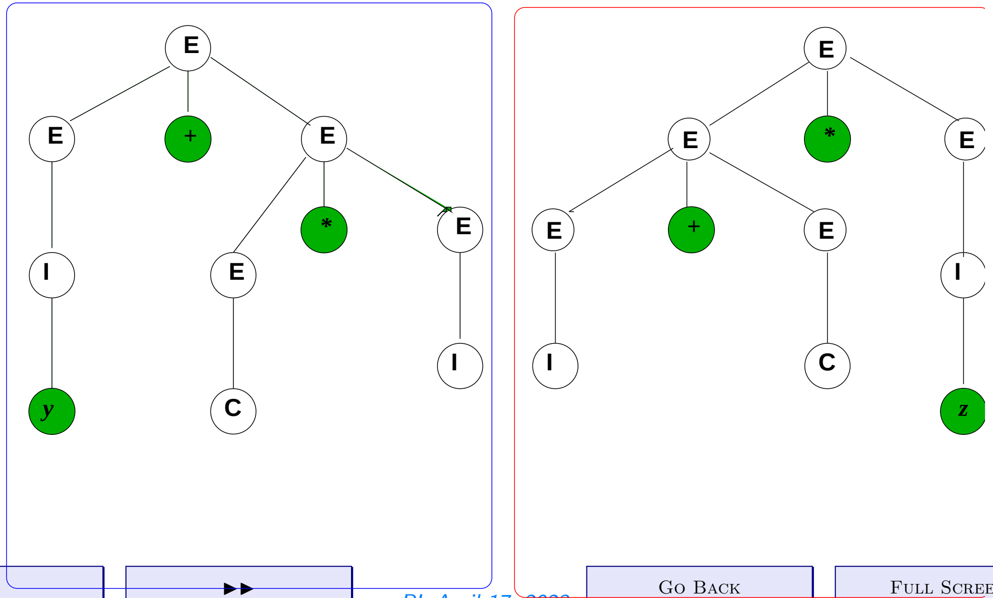


Ambiguity: 4

$G_1 = \langle\{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\}\rangle$ where P_1 consists of the following productions.

$$\begin{array}{lcl} E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\ I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\ C & \rightarrow & \mathbf{4} \end{array}$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.

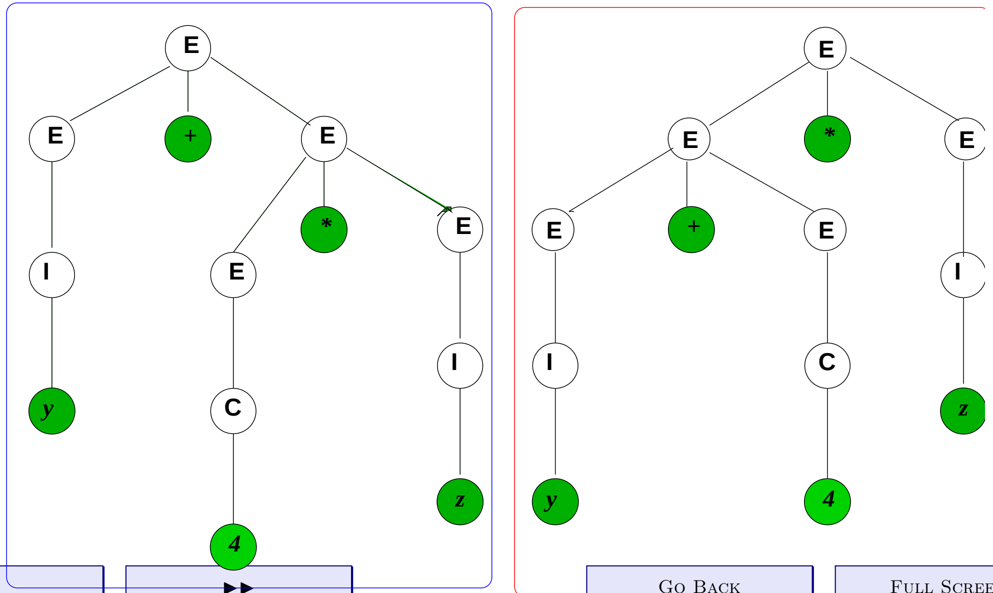


Ambiguity: 5

$G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$ where P_1 consists of the following productions.

$$
\begin{array}{lcl}
E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.



Left-most Derivation 1

Left-most derivation of $\mathbf{y+4*z}$ corresponding to the *first* derivation tree.

$$
\begin{array}{ll}
E & \Rightarrow \\
E+E & \Rightarrow \\
I+E & \Rightarrow \\
\mathbf{y}+E & \Rightarrow \\
\mathbf{y}+E*E & \Rightarrow \\
\mathbf{y}+C*E & \Rightarrow \\
\mathbf{y}+\mathbf{4}*E & \Rightarrow \\
\mathbf{y}+\mathbf{4}*I & \Rightarrow \\
\mathbf{y}+\mathbf{4}*\mathbf{z} &
\end{array}
$$

Left-most Derivation 2

Left-most derivation of $\mathbf{y}+4*\mathbf{z}$ corresponding to the *second* derivation tree.

$$
\begin{array}{ll}
E & \Rightarrow \\
E*E & \Rightarrow \\
E+E*E & \Rightarrow \\
I+E*E & \Rightarrow \\
\mathbf{y}+E*E & \Rightarrow \\
\mathbf{y}+C*E & \Rightarrow \\
\mathbf{y}+\mathbf{4}*E & \Rightarrow \\
\mathbf{y}+\mathbf{4}*I & \Rightarrow \\
\mathbf{y}+\mathbf{4}*\mathbf{z} &
\end{array}
$$

Right-most Derivation 1

Right-most derivation of $\mathbf{y{+}4{*}z}$ corresponding to the *first* derivation tree.

$$
\begin{array}{ll}
E & \Rightarrow \\
E{+}E & \Rightarrow \\
E{+}E{*}E & \Rightarrow \\
E{+}E{*}I & \Rightarrow \\
E{+}E{*}\mathbf{z} & \Rightarrow \\
E{+}C{*}\mathbf{z} & \Rightarrow \\
E{+}\mathbf{4}*\mathbf{z} & \Rightarrow \\
I{+}\mathbf{4}*\mathbf{z} & \Rightarrow \\
\mathbf{y}+\mathbf{4}*\mathbf{z} &
\end{array}
$$

Right-most Derivation 2

Right-most derivation of $\mathbf{y{+}4{*}z}$ corresponding to the *second* derivation tree.

$$\begin{array}{ll}
E & \Rightarrow \\
E{*}E & \Rightarrow \\
E{*}I & \Rightarrow \\
E{*}\mathbf{z} & \Rightarrow \\
E{+}E{*}\mathbf{z} & \Rightarrow \\
E{+}C{*}\mathbf{z} & \Rightarrow \\
E{+}\mathbf{4} * \mathbf{z} & \Rightarrow \\
I{+}\mathbf{4} * \mathbf{z} & \Rightarrow \\
\mathbf{y} + \mathbf{4} * \mathbf{z} &
\end{array}$$

Characterizing Ambiguity

The following statements are equivalent.

- A CFG is *ambiguous* if some sentence it generates has more than one *derivation tree*
- A CFG is *ambiguous* if there is a some sentence it generates with more than one *left-most derivation*
- A CFG is *ambiguous* if there is a some sentence it generates with more than one *right-most derivation*

Ambiguity in CFLs

see Wikipedia

- Some ambiguities result from incorrect grammars, i.e. there may exist a grammar which generates the same language with unique derivation trees.
- There may be some languages which are *inherently ambiguous* i.e. there is no context-free grammar for the language with only unique derivation trees for every sentence of the language.
- Whether a given CFG is ambiguous is *undecidable* i.e. there is no algorithm which can decide whether a given context-free grammar is ambiguous.
- Whether a given context-free language is *inherently ambiguous* is also *undecidable* since there is no algorithm which can decide whether any CFG that generates the language is ambiguous.

Removing ambiguity

There are essentially three ways adopted by programming language designers or compiler designers to remove ambiguity

- Change the language generated by introducing new bracketing tokens, (e.g. new reserved keywords **begin**...**end**).
- Introduce new precedence or associativity rules to disambiguate – this will invalidate certain derivation trees and may guarantee uniqueness, (e.g. the *dangling-else problem* see section 4.4).
- Change the grammar of the language (without changing the language generated)

Disambiguation

The only way to remove ambiguity (*without changing the language generated for a language which is not ambiguous*) is to change the grammar by introducing some more non-terminal symbols and changing the production rules[a]. Consider the grammar $G_1' = \langle N', \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P', \{E\}\rangle$ where $N' = N \cup \{T, F\}$ with the following production rules P'.

$$
\begin{array}{lcl}
E & \rightarrow & E{+}T \mid T \\
T & \rightarrow & T{*}F \mid F \\
F & \rightarrow & I \mid C \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

and compare it with the grammar G_1

[a] However the introduction of fresh non-terminals and rules may introduce new ambiguities, if th edesigner is not careful!

Left-most Derivation 1'

The left-most derivation of $\mathbf{y{+}4{*}z}$ is then as follows.

$$
\begin{array}{ll}
E & \Rightarrow \\
E{+}T & \Rightarrow \\
I{+}T & \Rightarrow \\
\mathbf{y}{+}T & \Rightarrow \\
\mathbf{y}{+}T{*}F & \Rightarrow \\
\mathbf{y}{+}T{*}F & \Rightarrow \\
\mathbf{y}{+}F{*}F & \Rightarrow \\
\mathbf{y}{+}C{*}F & \Rightarrow \\
\mathbf{y}{+}\mathbf{4}{*}F & \Rightarrow \\
\mathbf{y}{+}\mathbf{4}{*}I & \Rightarrow \\
\mathbf{y} + \mathbf{4} * \mathbf{z} &
\end{array}
$$

Left-most Derivations

Compare it with the Left-most Derivation 1.

G_1. $E \Rightarrow E{+}E \Rightarrow I{+}E \Rightarrow \mathbf{y}{+}E \Rightarrow \mathbf{y}{+}E{*}E \Rightarrow \mathbf{y}{+}C{*}E \Rightarrow \mathbf{y}{+}\mathbf{4}{*}E \Rightarrow \mathbf{y}{+}\mathbf{4}{*}I \Rightarrow \mathbf{y} + \mathbf{4} * \mathbf{z}$

G_1'. $E \Rightarrow E{+}T \Rightarrow I{+}T \Rightarrow \mathbf{y}{+}T \Rightarrow \mathbf{y}{+}T{*}F \Rightarrow \mathbf{y}{+}T{*}F \Rightarrow \mathbf{y}{+}F{*}F \Rightarrow \mathbf{y}{+}C{*}F \Rightarrow \mathbf{y}{+}\mathbf{4}{*}F \Rightarrow \mathbf{y}{+}\mathbf{4}{*}I \Rightarrow \mathbf{y} + \mathbf{4} * \mathbf{z}$

There is no derivation in G_1' corresponding to Left-most Derivation 2 (*Why not?*).

Right-most Derivation 1'

Right-most derivation of $\mathbf{y}+\mathbf{4}$*$\mathbf{z}$ corresponding to the *first* derivation tree.

$$
\begin{array}{ll}
E & \Rightarrow \\
E+T & \Rightarrow \\
E+T*F & \Rightarrow \\
E+T*I & \Rightarrow \\
E+T*\mathbf{z} & \Rightarrow \\
E+C*\mathbf{z} & \Rightarrow \\
E+\mathbf{4}*\mathbf{z} & \Rightarrow \\
F+\mathbf{4}*\mathbf{z} & \Rightarrow \\
I+\mathbf{4}*\mathbf{z} & \Rightarrow \\
+\mathbf{4}*\mathbf{z} & \Rightarrow \\
\mathbf{y}+\mathbf{4}*\mathbf{z} &
\end{array}
$$

Compare it with the Right-most Derivation 1.

There is no derivation corresponding to Right-most Derivation 2.

Disambiguation by Parenthesization

Another method of disambiguating a language is to change the language generated, by introducing suitable bracketing mechanisms.

Example 4.10 *Compare the following fully parenthesized grammar* G_2 *(which has the extra terminal symbols* $($ *and* $)$*) with the grammar* G_1 *without parentheses*

$$\begin{array}{lcl} E & \rightarrow & I \mid C \mid (E+E) \mid (E*E) \\ I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\ C & \rightarrow & \mathbf{4} \end{array}$$

Though unambiguous, the language defined by this grammar is different from that of the original grammar without parentheses.

Associativity and Precedence

The grammar G_1' implements

Precedence. $*$ has higher precedence than $+$.

Associativity. $*$ and $+$ are both left associative operators.

but is parentheses-free, whereas grammar G_2 generates a different language which is unambiguous. We may combine the two with the benefits of both.

Example 4.11 *Compare the following parenthesized grammar G_2' which combines the benefits of both G_1' and G_2 (parenthesization wherever required by implementing the $bodmas$ rule). $G_2' = \langle N', \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +, (,)\}, P_2', \{E\}\rangle$ where $N' = N \cup \{T, F\}$ with the following production rules P_2'.*

$$
\begin{array}{lcl}
E & \rightarrow & E{+}T \mid T \\
T & \rightarrow & T{*}F \mid F \\
F & \rightarrow & I \mid C \mid (E) \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

4.4. The "dangling else" problem

- Some programming languages like FORTRAN and Assembly (conditional jumps) have a single **if**. . . **then** construct. We write $it(b, C)$ to denote **if** b **then** C.
- Some programming languages like ML, OCAML have a single **if**. . . **then**. . . **else** construct and we write $ite(b, C, C')$ to denote **if** b **then** C **else** C'.
- Many programming languages have both **if**. . . **then** and **if**. . . **then**. . . **else** constructs which potentially may lead to a *dangling-else* problem.

The dangling-else problem potentially is an ambiguity associated with a compound construct such as

$$\textbf{if } b_1 \textbf{ then if } b_2 \textbf{ then } C_1 \textbf{ else } C_2 \tag{4}$$

where b_1 and b_2 are boolean expressions and C_1 and C_2 are appropriate constructs (expressions or commands) that are allowed by the language.

The ambiguity arises because the construct (4) may be interpreted as denoting either $it(b_1, ite(b_2, C_1, C_2))$ or $ite(b_1, it(b_2, C_1), C_2)$.

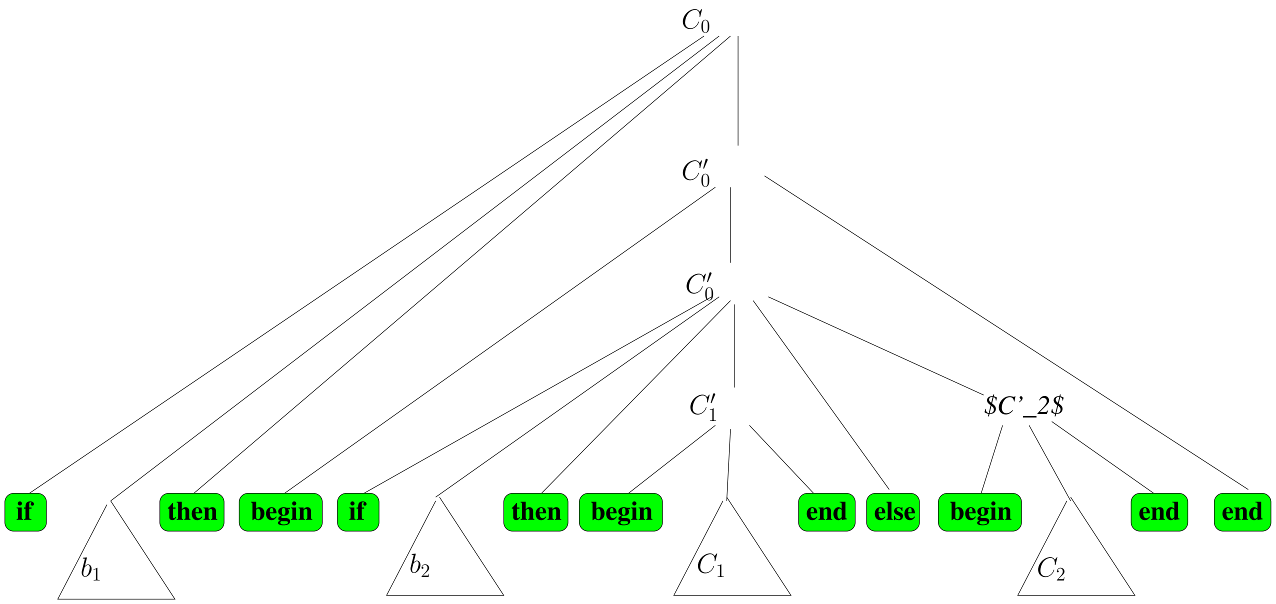


Figure 1: $it(b_1, ite(b_2, C_1, C_2))$

Disambiguation

1. Disambiguation may be achieved in the language by introducing new bracketing symbols (e.g. **begin**...**end**) for all constructs of the kind that C belongs to. If the use of these brackets is made mandatory in the language then the construct (4) itself would be syntactically illegal and would have to be replaced by one of the following depending upon the programmer's intention.

- If the programmer's intention corresponds to $it(b_1, ite(b_2, C_1, C_2))$ (see the parse tree in figure 1) then

if b_1 **then**
  begin
    if b_2 **then**
      begin
        C_1
      end
    else
      begin
        C_2
      end
  end

- If programmer intended $ite(b_1, it(b_2, C_1), C_2)$ (see the parse tree in figure 2).

if b_1 **then**
  begin
    if b_2 **then**
      begin
        C_1
      end
  end

else
  begin
    C_2
  end

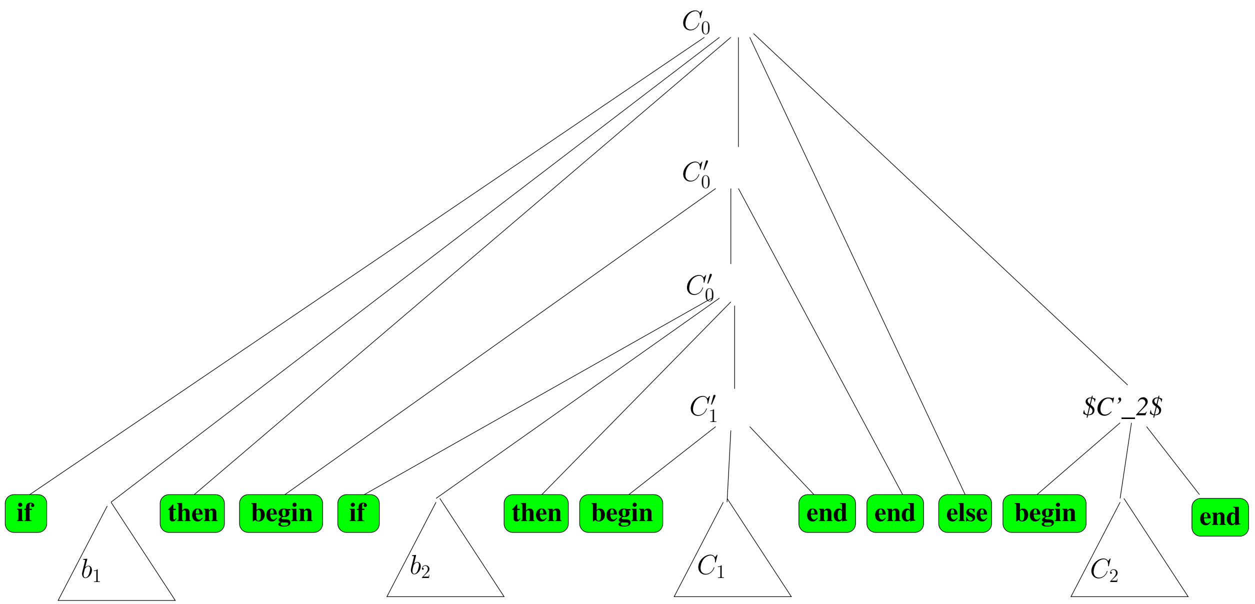


Figure 2: $ite(b_1, it(b_2, C_1), C_2)$

2. While the use of **begin**. . . **end** is general-purpose enough for all constructs of the kind that C is, it tends to introduce too many tokens in an actual program. Some languages (e.g. Bash) instead introduce a unique *closing* token for

each construct. That is, the constructs come with pairs of unique opening and closing tokens (e.g. **if**. . . **then**. . . **fi**, **if**. . . **then**. . . **else**. . . **fi**, **case**. . . **esac** etc.) In such a language the constructs corresponding to $it(b_1, ite(b_2, C_1, C_2))$ would then be written as follows (see also the parse tree in figure 3).

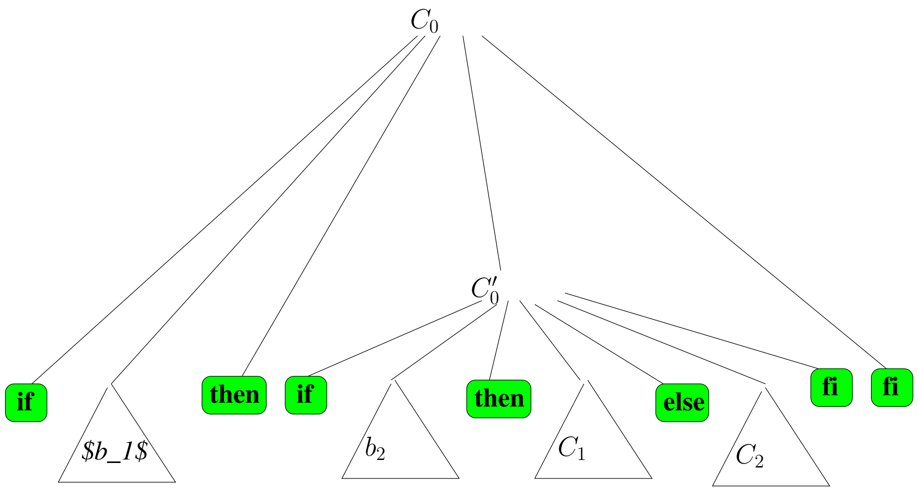


Figure 3: $it(b_1, ite(b_2, C_1, C_2))$

if b_1 **then**
  if b_2 **then**
    C_1
  else
    C_2
fi

while $ite(b_1, it(b_2, C_1), C_2)$ would be written as (see also the parse tree in figure 4)

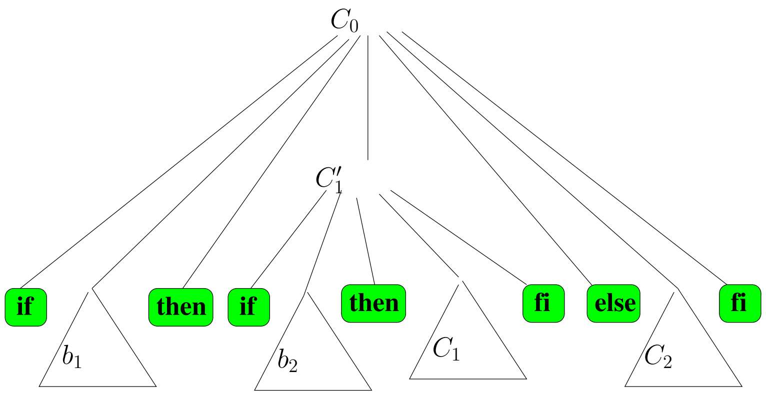


Figure 4: $it(b_1, ite(b_2, C_1, C_2))$

if b_1 **then**
  if b_2 **then**
    C_1
  fi
else
  C_2
fi

In general this solution leads to a larger number of reserved words in the language but a smaller number of tokens (produced after scanning) per syntactically valid program as opposed to the previous solution. Languages like C and

Perl also dispense with the reserved word **then** by insisting that all conditions in such statements be enclosed in parentheses.

3. Languages like Pascal which use a single bracketing mechanism for command constructs, often try to reduce the number of tokens produced per program by relaxing the mandatory requirement of bracketing, by stipulating that bracketing is required only for compound commands. Thus for atomic commands c_1 and c_2 the ambiguity in

$$\textbf{if } b_1 \textbf{ then if } b_2 \textbf{ then } c_1 \textbf{ else } c_2 \tag{5}$$

is resolved by introducing an associativety rule that each **else** is associated with the nearest enclosing condition. That is the construct (5) is interpreted as referring to $it(b_1, ite(b_2, C_1, C_2))$.

4. There are other means of achieving disambiguation of which the most ingenious is the use of white-space indentation in Python to keep it unambiguous. Hence in Python $it(b_1, ite(b_2, C_1, C_2))$ would be written as

if b_1:
    if b_2:
        C_1
    else:
        C_2

and $ite(b_1, it(b_2, C_1), C_2)$ would be written as

if b_1:

if b_2:

C_1

else:

C_2

Exercise 4.1

1. *Two context-free grammars are considered equivalent if they generate the same language. Prove that* G_1 *and* G'_1 *are equivalent.*
2. *Palindromes. A* palindrome *is a string that is equal to its reverse i.e. it is the same when read backwards (e.g.* `aabbaa` *and* `abaabaaba` *are both palindromes). Design a grammar for generating all palindromes over the terminal symbols* `a` *and* `b`*.*
3. *Matching brackets.*
   (a) *Design a context-free grammar to generate sequences of matching brackets when the set of terminals consists of three pairs of brackets* $\{(,), [,], \{, \}\}$*.*
   (b) *If your grammar is ambiguous give two rightmost derivations of the same string and draw the two derivation trees. Explain how you will modify the grammar to make it unambiguous.*
   (c) *If your grammar is not ambiguous prove that it is not ambiguous.*
4. *Design an unambiguous grammar for the expression language on integers consisting of expressions made up of operators* `+`, `-`, `*`, `/`, `%` *and the bracketing symbols* `(` *and* `)`*, assuming the usual rules of precedence among operators that you have learned in school.*

5. *Modify the above grammar to include the exponentiation operator* ^ *which has a higher precedence than the other operators and is right-associative.*

6. *How will you modify the grammar above to include the unary minus operator* - *where the unary minus has a higher precedence than other operators?*

7. *The language specified by a regular expression can also be generated by a context-free grammar.*

   (a) *Design a context-free grammar to generate all floating-point numbers allowed by the C language.*

   (b) *Design a context-free grammar to generate all numbers in binary form that are* not *multiples of* 4.

   (c) *Write a regular expression to specify all numbers in binary form that are multiples of of* 3.

8. *Prove that the* G_1' *is indeed unambiguous.*

9. *Prove that the grammar of fully parenthesized expressions is unambiguous.*

10. *Explain how the grammar* G_1' *implements left associativity and precedence.*

Specification of Syntax: EBNF

The EBNF specification of a programming language is a collection of rules that defines the (context-free) grammar of the language. It specifies the formation rules for the correct grammatical construction of the phrases of the language. In order to reduce the number of rules unambiguously regular expression operators such as alternation, Kleene closure and +-closure are also used. (Con)catenation is represented by juxtaposition. In addition, a period is used to terminate a rule. The rules are written usually in a “top-down fashion”.

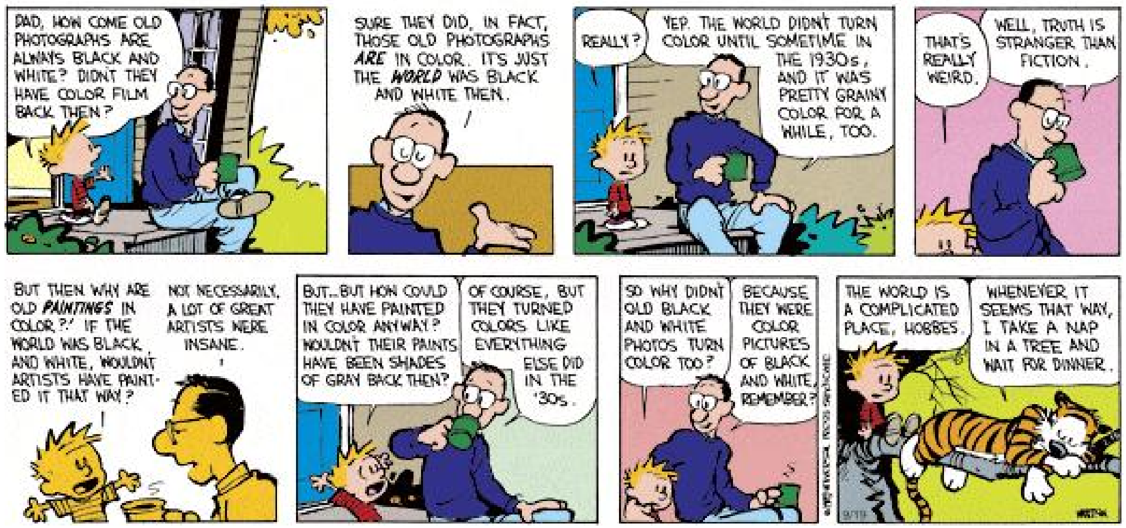
Start symbol. The very first rule gives the productions of the start symbol of the grammar.

Non-terminals. Uses English words or phrases to denote non-terminal symbols. These words or phrases are suggestive of the nature or meaning of the constructs.

Metasymbols.

- Sequences of constructs enclosed in “{” and “}” denote zero or more occurrences of the construct (c.f. Kleene closure on regular expressions).
- Sequences of constructs enclosed in “[” and “]” denote that the enclosed constructs are optional i.e. there can be only zero or one occurrence of the sequence.
- Constructs are enclosed in “(” and “)” to group them together.
- “ | ” separates alternatives.
- “ ::= ” defines the productions of each non-terminal symbol.
- “ .” terminates the possibly many rewrite rules for a non-terminal.

Terminals. Terminal symbol strings are sometimes enclosed in double-quotes when written in monochrome (we shall additionally colour-code them).



Note.

We have chosen to colour-code the EBNF specification in order to clearly separate the colours of the EBNF operators from those of the language that is being specified. Further we have chosen to use different colours for the *Nonterminal* symbols and the *terminal* symbols. In the bad old days when the world was only black-and-white and the only font available was the `type-writer` font, the `<Nonterminal>` symbols were usually enclosed in "`<>`" while the `terminal` symbols were written directly (optionally enclosed in double-quotes (`"`)).

Balanced Parentheses: CFG

Example 4.12 *A context-free grammar for balanced parentheses (including the empty string) over the terminal alphabet* $\{(,), [,], \{, \}\}$ *could be given as* $BP_3 = \langle\{S\}, \{(,), [,], \{, \}\}, P, \{S\}\rangle$, *where* P *consists of the productions*

$$
\begin{aligned}
S &\rightarrow \epsilon, \\
S &\rightarrow (S)S, \\
S &\rightarrow [S]S, \\
S &\rightarrow \{S\}S
\end{aligned}
$$

Balanced Parentheses: EBNF

Example 4.13 BP_3 *may be expressed in EBNF as follows:*

$$
\begin{array}{lcl}
BracketSeq & ::= & \{Bracket\}\ . \\
Bracket & ::= & LeftParen\ BracketSeq\ RightParen\ | \\
 & & LeftSqbracket\ BracketSeq\ RightSqbracket\ | \\
 & & LeftBrace\ BracketSeq\ RightBrace\ . \\
LeftParen & ::= & ``(" \ . \\
RightParen & ::= & ``)" \ . \\
LeftSqbracket & ::= & ``[" \ . \\
RightSqbracket & ::= & ``]" \ . \\
LeftBrace & ::= & ``\{" \ . \\
RightBrace & ::= & ``\}" \ .
\end{array}
$$

EBNF in EBNF

EBNF has its own grammar which is again context-free. Hence EBNF (4.5) may be used to define EBNF in its own syntax as follows:

$$
\begin{array}{lcl}
Syntax & ::= & \{Production\}\ . \\
Production & ::= & NonTerminal\ \text{``}::=\text{''}\ PossibleRewrites\ \text{``}.\text{''}\ . \\
PossibleRewrites & ::= & Rewrite\ \{\text{``}|\text{''}\ Rewrite\}\ . \\
Rewrite & ::= & Symbol\ \{Symbol\}\ . \\
Symbol & ::= & NonTerminal \mid Terminal \mid GroupRewrites\ . \\
GroupRewrites & ::= & \text{``}\{\text{''}\ PossibleRewrites\ \text{``}\}\text{''} \mid \\
 & & \text{``}[\text{''}\ PossibleRewrites\ \text{``}]\text{''} \mid \\
 & & \text{``}(\text{''}\ PossibleRewrites\ \text{``})\text{''}\ . \\
NonTerminal & ::= & Letter\ \{Letter \mid Digit\}\ . \\
Terminal & ::= & Character\ \{Character\}\ .
\end{array}
$$

EBNF: Character Set

The character set used in EBNF is described below.

$$
\begin{array}{lcl}
Character & ::= & Letter \mid Digit \mid SpecialChar \\
Letter & ::= & UpperCase \mid LowerCase \\
UpperCase & ::= & \text{"}A\text{"} \mid \text{"}B\text{"} \mid \text{"}C\text{"} \mid \text{"}D\text{"} \mid \text{"}E\text{"} \mid \text{"}F\text{"} \mid \text{"}G\text{"} \mid \text{"}H\text{"} \mid \\
 & & \text{"}I\text{"} \mid \text{"}J\text{"} \mid \text{"}K\text{"} \mid \text{"}L\text{"} \mid \text{"}M\text{"} \mid \text{"}N\text{"} \mid \text{"}O\text{"} \mid \text{"}P\text{"} \mid \text{"}Q\text{"} \mid \\
 & & \text{"}R\text{"} \mid \text{"}S\text{"} \mid \text{"}T\text{"} \mid \text{"}U\text{"} \mid \text{"}V\text{"} \mid \text{"}W\text{"} \mid \text{"}X\text{"} \mid \text{"}Y\text{"} \mid \text{"}Z\text{"} \\
LowerCase & ::= & \text{"}a\text{"} \mid \text{"}b\text{"} \mid \text{"}c\text{"} \mid \text{"}d\text{"} \mid \text{"}e\text{"} \mid \text{"}f\text{"} \mid \text{"}g\text{"} \mid \text{"}h\text{"} \mid \\
 & & \text{"}i\text{"} \mid \text{"}j\text{"} \mid \text{"}k\text{"} \mid \text{"}l\text{"} \mid \text{"}m\text{"} \mid \text{"}n\text{"} \mid \text{"}o\text{"} \mid \text{"}p\text{"} \mid \text{"}q\text{"} \mid \\
 & & \text{"}r\text{"} \mid \text{"}s\text{"} \mid \text{"}t\text{"} \mid \text{"}u\text{"} \mid \text{"}v\text{"} \mid \text{"}w\text{"} \mid \text{"}x\text{"} \mid \text{"}y\text{"} \mid \text{"}z\text{"} \\
Digit & ::= & \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"} \\
SpecialChar & ::= & \text{"!"} \mid \text{"""} \mid \text{"\#"} \mid \text{"\$"} \mid \text{"\%"} \mid \text{"\&"} \mid \text{"'"} \mid \text{"("} \mid \text{")"} \mid \text{"}*\text{"} \mid \\
 & & \text{"}+\text{"} \mid \text{","} \mid \text{"}-\text{"} \mid \text{"."} \mid \text{"}/\text{"} \mid \text{":"} \mid \text{";"} \mid \text{"}<\text{"} \mid \text{"}=\text{"} \mid \text{"}>\text{"} \mid \text{"?"} \\
 & & \text{"@"} \mid \text{"["} \mid \text{"}\backslash\text{"} \mid \text{"]"} \mid \text{"\^{}"} \mid \text{"\_"} \mid \text{"`"} \mid \text{"}\{\text{"} \mid \text{"}|\text{"} \mid \text{"}\}\text{"} \mid \text{"\~{}"}
\end{array}
$$

EBNF in ASCII-EBNF

```
<Syntax> ::= {<Production>}.
<Production> ::= <NonTerminal> "::=" <PossibleRewrites>".".
<PossibleRewrites> ::= <Rewrite> {"|" <Rewrite>}.
<Rewrite> ::= <Symbol> {<Symbol>}.
<Symbol> ::= <NonTerminal> | <Terminal> | <GroupRewrites>.
<GroupRewrites> ::= "{"<PossibleRewrites>"}" |
                    "["<PossibleRewrites>"]"  |
                    "("<PossibleRewrites>")".
<NonTerminal> ::= <Letter>{<Letter> | <Digit>}.
<Terminal> ::= <Character>{<Character>}.
```

We leave it to the interested reader to define the nonterminals <Digit>, <Letter> and <Character>. Many languages even now are specified in some slight variant of the above notation.

4.6. The WHILE Programming Language: Syntax

All words written in **bold** font are reserved words and cannot be used as identifiers in any program.

$$
\begin{array}{lcl}
Program & ::= & \text{“}\textbf{program}\text{”}\ Identifier\ \text{“::”}Block\ . \\
Block & ::= & DeclarationSeq\ CommandSeq\ . \\
DeclarationSeq & ::= & \{Declaration\}\ . \\
Declaration & ::= & \text{“}\textbf{var}\text{”}\ VariableList\text{“:”}Type\text{“;”}\ . \\
Type & ::= & \text{“}\textbf{int}\text{”}\ |\ \text{“}\textbf{bool}\text{”}\ . \\
VariableList & ::= & Variable\{\text{“,”}\ Variable\}\ . \\
CommandSeq & ::= & \text{“\{”}\{Command\text{“;”}\}\text{“\}”}\ . \\
Command & ::= & Variable\text{“:=”}Expression\ | \\
 & & \text{“}\textbf{read}\text{”}\ Variable\ | \\
 & & \text{“}\textbf{write}\text{”}\ IntExpression\ | \\
 & & \text{“}\textbf{if}\text{”}\ BoolExpression\ \text{“}\textbf{then}\text{”}\ CommandSeq \\
 & & \text{“}\textbf{else}\text{”}\ CommandSeq \\
 & & \text{“}\textbf{endif}\text{”}\ | \\
 & & \text{“}\textbf{while}\text{”}\ BoolExpression\ \text{“}\textbf{do}\text{”}\ CommandSeq \\
 & & \text{“}\textbf{endwh}\text{”}\ .
\end{array}
$$

$$
\begin{array}{lcl}
Expression & ::= & IntExpression \mid BoolExpression\ . \\
IntExpression & ::= & IntExpression\ AddOp\ IntTerm \mid IntTerm\ . \\
IntTerm & ::= & IntTerm\ MultOp\ IntFactor \mid IntFactor\ . \\
IntFactor & ::= & Numeral \mid Variable \mid \\
 & & \text{“(”}IntExpression\text{“)”} \mid \text{“\textasciitilde”}IntFactor\ . \\
BoolExpression & ::= & BoolExpression\ \text{“||”}\ BoolTerm \mid BoolTerm\ . \\
BoolTerm & ::= & BoolTerm\ \text{“\&\&”}\ BoolFactor \mid BoolFactor\ . \\
BoolFactor & ::= & \text{“}\mathbf{tt}\text{”} \mid \text{“}\mathbf{ff}\text{”} \mid Variable \mid Comparison \mid \\
 & & \text{“(”}BoolExpression\text{“)”} \mid \text{“!”}BoolFactor\ . \\
Comparison & ::= & IntExpression\ RelOp\ IntExpression\ . \\
Variable & ::= & Identifier\ . \\
RelOp & ::= & \text{“}<\text{”} \mid \text{“}<=\text{”} \mid \text{“}=\text{”} \mid \text{“}>\text{”} \mid \text{“}>=\text{”} \mid \text{“}<>\text{”}\ . \\
AddOp & ::= & \text{“}+\text{”} \mid \text{“}-\text{”}\ . \\
MultOp & ::= & \text{“}*\text{”} \mid \text{“}/\text{”} \mid \text{“}\%\text{”}\ . \\
Identifier & ::= & Letter\{Letter \mid Digit\}\ . \\
Numeral & ::= & [\text{“}+\text{”} \mid \text{“\textasciitilde”}]Digit\{Digit\}\ .
\end{array}
$$

Note

1. “;” acts as a terminator for both *Declarations* and *Commands*.
2. “,” acts as a separator in *VariableList*

3. *Comparison* has a higher precedence than *BoolTerm* and *BoolExpression*.
4. *RelOp*s have lower precedence than any of the integer operations specified in *MultOp* and *AddOp*.
5. The nonterminals *Letter* and *Digit* are as specified earlier in the EBNF character set

Syntax Diagrams

- EBNF was first used to define the grammar of ALGOL-60 and the syntax was used to design the parser for the language.
- EBNF also has a diagrammatic rendering called syntax diagrams or railroad diagrams. The grammar of SML has been rendered by a set of syntax diagrams.
- Pascal has been defined using both the text-version of EBNF and through syntax diagrams.
- While the text form of EBNF helps in parsing, the diagrammatic rendering is only for the purpose of readability.
- EBNF is a specification language that almost all modern programming languages use to define the grammar of the programming language

Syntax Specifications

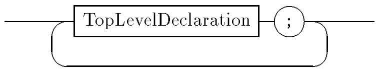
- BNF of C
- BNF of Java
- EBNF of Pascal
- Pascal Syntax diagrams
- BNF of Standard ML
- BNF of Datalog
- BNF of Prolog

Syntax of Standard ML

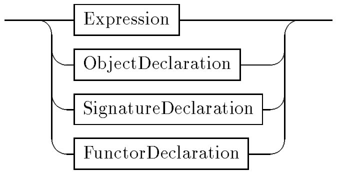
Tobias Nipkow and Larry Paulson

PROGRAMS AND MODULES

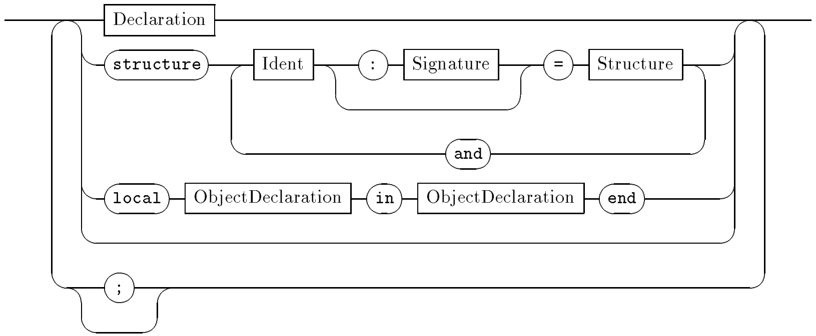
Program



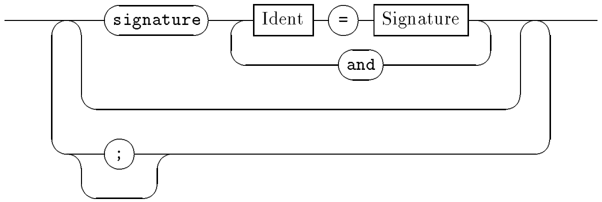
TopLevelDeclaration



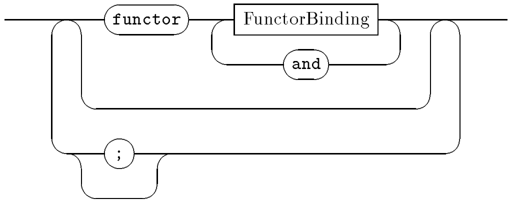
ObjectDeclaration



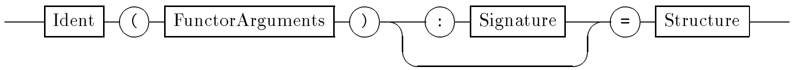
SignatureDeclaration



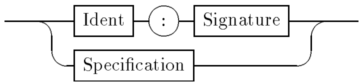
FunctorDeclaration



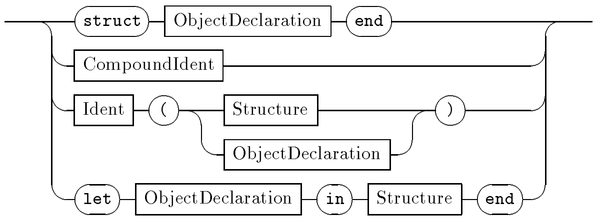
FunctorBinding



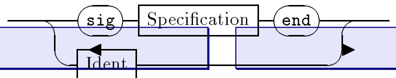
FunctorArguments



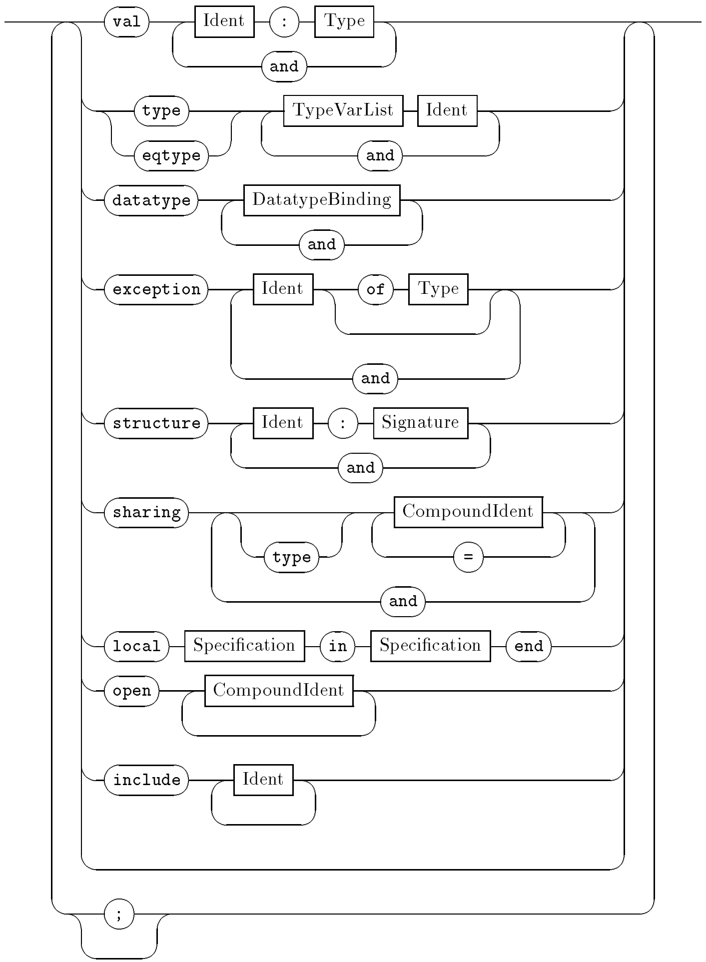
Structure

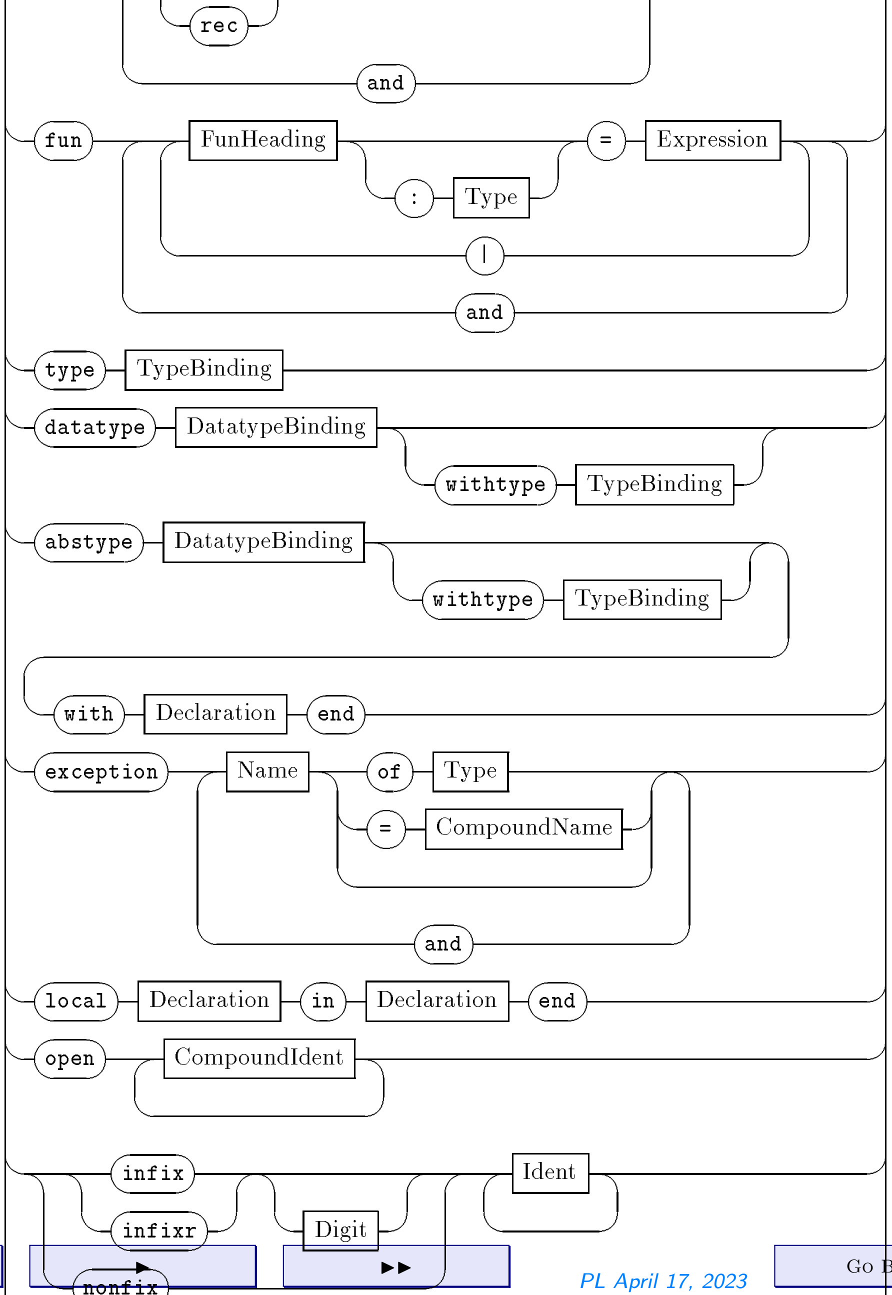


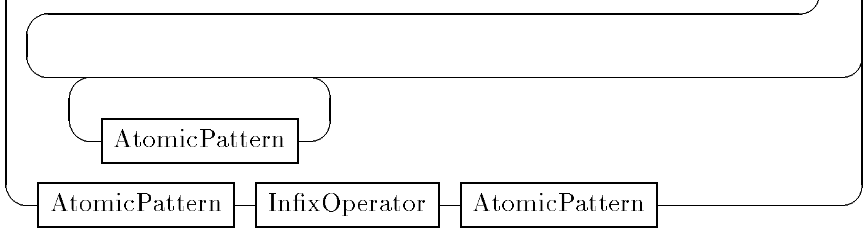
Signature



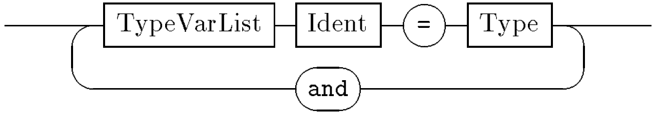
Specification



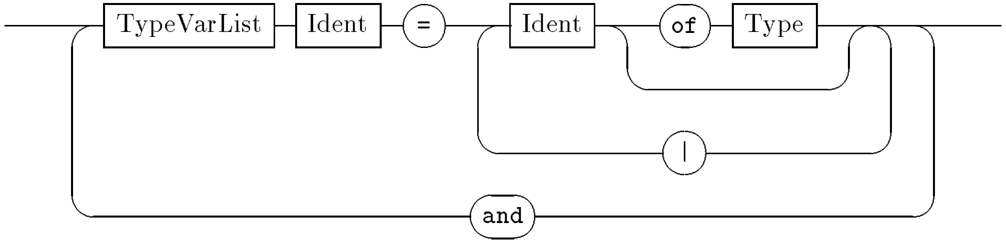

rec
and
fun
FunHeading
:
Type
=
Expression
|
and
type
TypeBinding
datatype
DatatypeBinding
withtype
TypeBinding
abstype
DatatypeBinding
withtype
TypeBinding
with
Declaration
end
exception
Name
of
Type
=
CompoundName
and
local
Declaration
in
Declaration
end
open
CompoundIdent
infix
infixr
nonfix
Digit
Ident



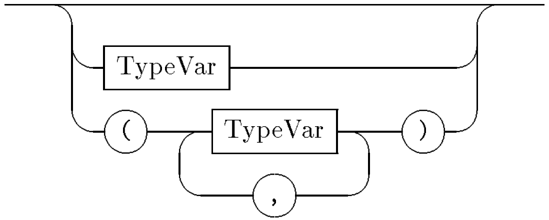
TypeBinding

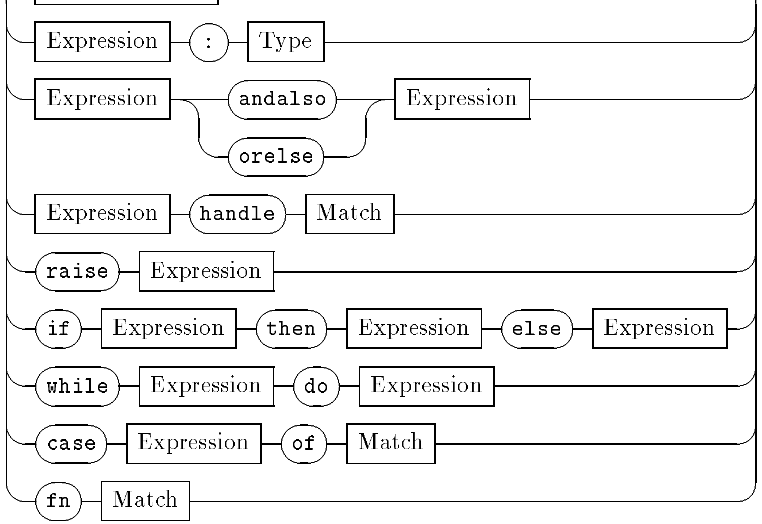


DatatypeBinding

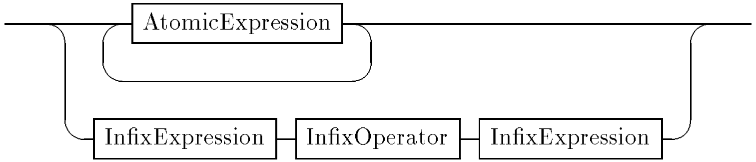


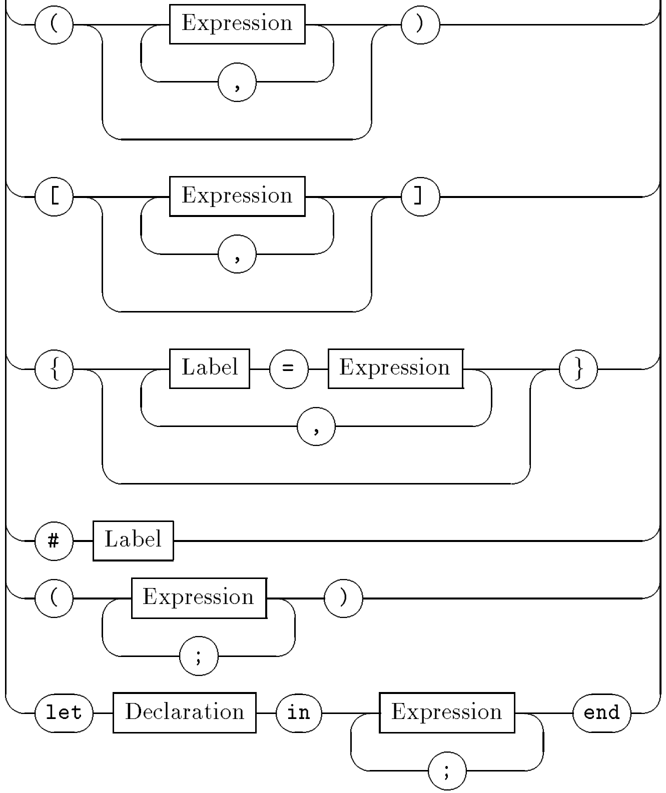
TypeVarList





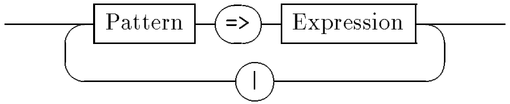
InfixExpression



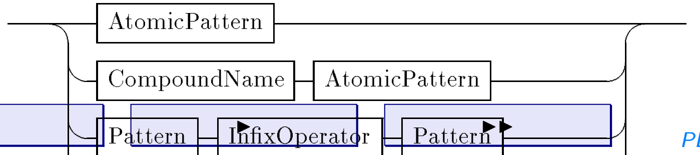


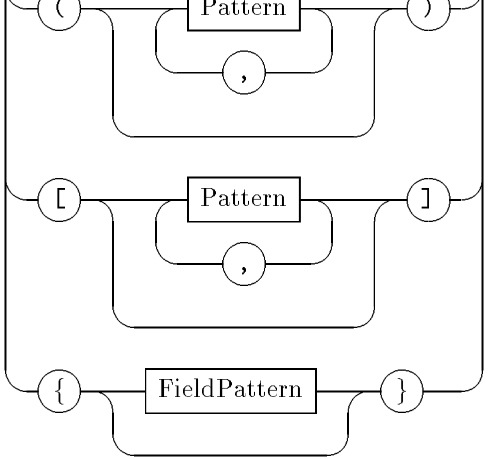
MATCHES AND PATTERNS

Match

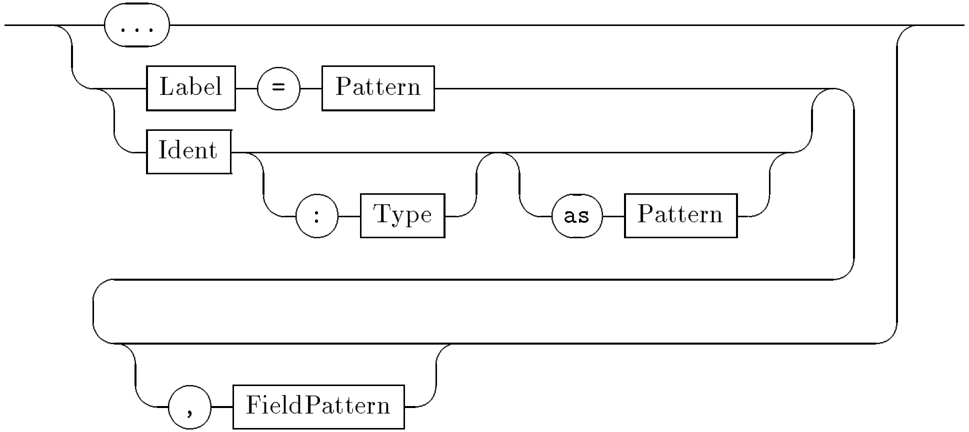


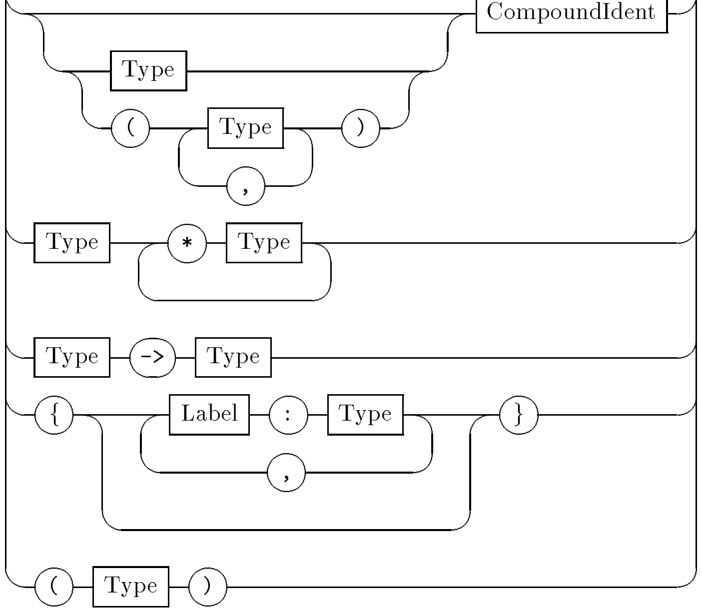
Pattern





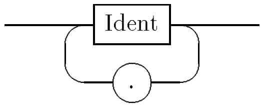
FieldPattern



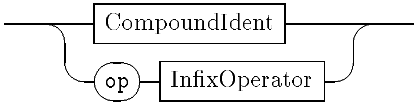


LEXICAL MATTERS: IDENTIFIERS, CONSTANTS, COMMENTS

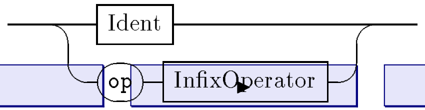
CompoundIdent

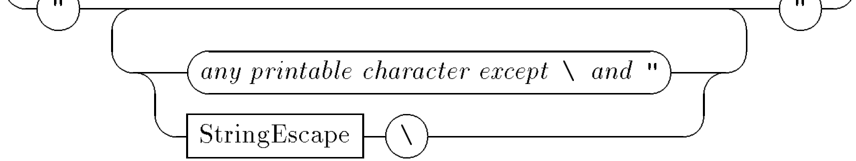


CompoundName

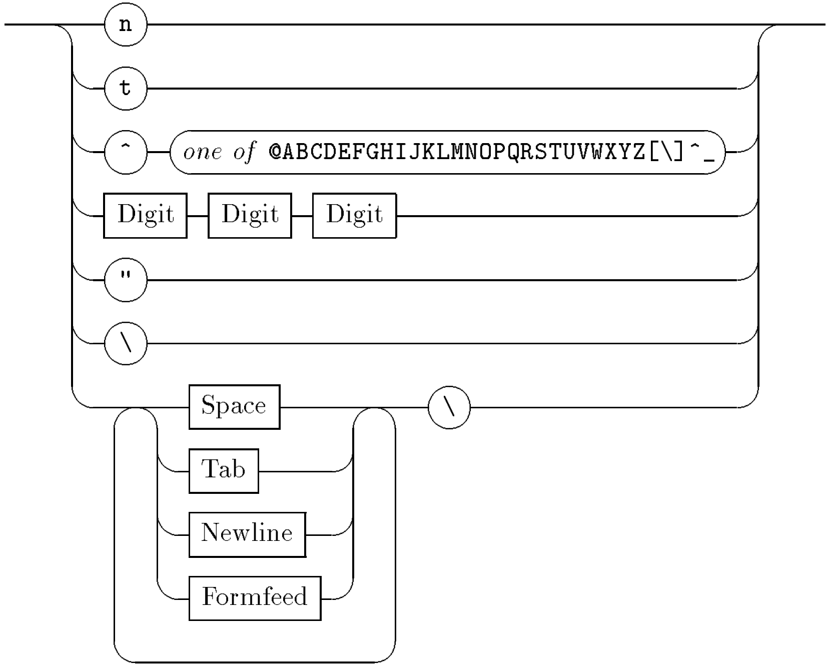


Name

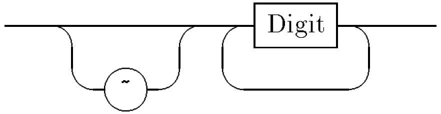




StringEscape

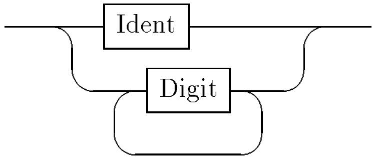


Numeral

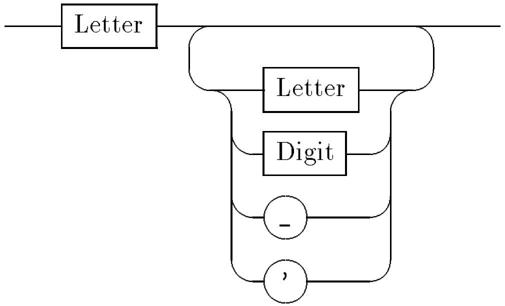


TypeVar

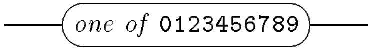
Label



AlphanumericIdent

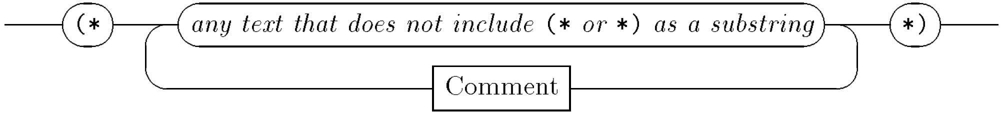


Digit



Letter

Comment



Exercise 4.2

1. *Translate all the context-free grammars that we have so far seen into EBNF specifications.*
2. *Specify the language of regular expressions over a non-empty finite alphabet A in EBNF.*
3. *Given a textual EBNF specification write an algorithm to render each non-terminal as a syntax diagram.*

Introduction to Parsing

Overview of Parsing

Since

- parsing requires the checking whether a given token stream *conforms* to the rules of the grammar and
- since a context-free grammar may generate an infinite number of different strings

any parsing method should be guided by the given input (token) string, so that a deterministic strategy may be evolved.

Parsing Methods

Two kinds of parsing methods

Top-down parsing Try to **generate** the given input sentence from the start symbol of the grammar by applying the production rules.

Bottom-up parsing Try to **reduce** the given input sentence to the start symbol by applying the rules in *reverse*

In general top-down parsing requires long *look-aheads* in order to do a deterministic guess from the given input token stream. On the other hand bottom-up parsing yields better results and can be automated by software tools.

Top-down Parsing

- Try to **generate** the given input sentence from the start symbol of the grammar by applying the production rules.
- Not the most general.
- But most modern high-level programming languages are designed to be efficiently parsed by this method.
- Recursive-descent is the most frequently employed technique when language $\mathcal{C}$ in which the compiler is written, supports recursion.

Recursive Descent Parsing

- Suitable for grammars that are LL(1)[a] parseable.
- A set of (mutually) recursive procedures
- Has a single procedure/function for each non-terminal symbol
- Allows for syntax errors to be pinpointed more accurately than most other parsing methods

[a] *L*eft-to-right *L*eft-most derivations with *1* look-ahead

Caveats with RDP: Direct Left Recursion

Any left-recursion in the grammar can lead to infinite recursive calls during which no input token is consumed and there is no return from the recursion. That is, they should not be of the form

$$A \longrightarrow A\alpha$$

This would result in an infinite recursion with no input token consumed.

Caveats with RDP: Indirect Left Recursion

- A production cannot even be *indirectly* left recursive. For instance the following is *indirect* left-recursion of cycle length 2.

  Example 4.14

  $$\begin{aligned} A &\longrightarrow B\beta \\ B &\longrightarrow A\alpha \end{aligned}$$

  where $\alpha, \beta \in (N \cup T)^*$.

- In general it should be impossible to have derivation sequences of the form $A \Rightarrow A_1\alpha_1 \cdots \Rightarrow A_{n-1}\alpha_{n-1} \Rightarrow A\alpha_n$ for nonterminal symbols $A, A_1, \ldots, A_{n-1}$ for any $n > 0$.

Caveats with RDP: Left Factoring

For RDP to succeed without backtracking, for each input token and each non-terminal symbol there should be only one rule applicable;

Example 4.15 *A set of productions of the form*

$$A \longrightarrow aB\beta \mid aC\gamma$$

where B and C stand for different phrases would lead to non-determinism. The normal practice then would be to left-factor *the two productions by introducing a new non-terminal symbol A' and rewrite the rule as*

$$\begin{aligned} A &\longrightarrow aA' \\ A' &\longrightarrow B\beta \mid C\gamma \end{aligned}$$

provided B and C generate terminal strings with different first symbols (otherwise more left-factoring needs to be performed).

A Simple Left-recursive Grammar

The following grammar is unambiguous and implements both left-associativity and precedence of operators. $G = \langle\{E, T, D\}, \{a, b, -, /(,)\}, P, E\rangle$ whose productions are

$$
\begin{aligned}
E &\rightarrow E{-}T \mid T \\
T &\rightarrow T/D \mid D \\
D &\rightarrow a \mid b \mid (E)
\end{aligned}
$$

Left Recursion Removal

The grammar G is clearly left-recursive in both the nonterminals E and T and hence is not amenable to recursive-descent parsing.

The grammar may then have to be modified as follows:

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & -TE' \mid \varepsilon \\
T & \rightarrow & DT' \\
T' & \rightarrow & /DT' \mid \varepsilon \\
D & \rightarrow & a \mid b \mid (E)
\end{array}
$$

Now this grammar is no longer left-recursive and may then be parsed by a recursive-descent parser.

Recursive Descent Parsing: Determinization

RDP can be deterministic only if

- the input token lookahead uniquely determines the production to be applied.
- We need to define the FIRST symbols that will be generated by each production.
- In the presence of ε productions, symbols that can FOLLOW a given non-terminal symbol also need to be specified.

Nullable

A nonterminal symbol X is **nullable** if it can derive the empty string, i.e. $X \Rightarrow^* \varepsilon$. The following algorithm computes $nullable(X)$ for each nonterminal symbol. For convenience $nullable$ is set to false for each terminal symbol in the grammar. $NULLABLE(N)$ is the set of boolean values specifying for each nonterminal symbol whether it is $nullable$.

Algorithm 4.1

$\textsc{Nullable}\ (N) \stackrel{df}{=}$

$$\begin{cases} \quad \textbf{Require: } \text{CFG } G = \langle N, T, P, S \rangle \\ \quad \textbf{Yields: } \mathit{NULLABLE}(N) = \{\mathit{nullable}(X) \mid X \in N\} \\ \textbf{for each } \mathtt{a} \in T \\ \quad \textbf{do } \mathit{nullable}(\mathtt{a}) := \textbf{ false } ; \\ \textbf{for each } X \in N \\ \quad \textbf{do } \mathit{nullable}(X) := \exists X \to \varepsilon \in P \\ \textbf{repeat} \\ \quad \textbf{for each } X \to \alpha_1 \dots \alpha_k \in P \\ \qquad \textbf{do } \begin{cases} \textbf{if } \forall i : 1 \leq i \leq k : \mathit{nullable}(\alpha_i) \\ \quad \textbf{then } \mathit{nullable}(X) := \textbf{ true} \end{cases} \\ \textbf{until } \mathit{NULLABLE}(N) \textit{ is unchanged} \end{cases}$$

First

$first(\alpha)$ is the set of terminal symbols that can be the first symbol of any string that α can derive, i.e. $a \in first(\alpha)$ if and only if there exists a derivation $\alpha \Rightarrow^* ax$ for any string of terminals x.

Notice that

- the computation of $first$ requires $nullable$ to be available. Also the $first$ of any terminal symbol is itself.
- also that if $X \to \alpha Z \beta$ is a production then one cannot ignore the $first(Z)$ in computing $first(X)$ especially if $\alpha \Rightarrow^* \varepsilon$. Further if Z is also nullable then $first(\beta) \subseteq first(X)$.

Algorithm 4.2

$\text{FIRST}\ (N \cup T) \stackrel{df}{=}$

$$\begin{cases} \textbf{Require: } \text{CFG } G = \langle N, T, P, S \rangle \\ \textbf{Yields: } FIRST(N \cup T) = \{ first(\alpha) \mid \alpha \in N \cup T \} \\ \textbf{for each } \mathtt{a} \in T \\ \quad \textbf{do } first(\mathtt{a}) := \{\mathtt{a}\} \\ \textbf{for each } X \in N \\ \quad \textbf{do } first(X) := \emptyset \\ \textbf{repeat} \\ \quad \textbf{for each } X \to \alpha_1 \ldots \alpha_k \in P \\ \qquad \textbf{do } \begin{cases} \textbf{if } \forall i' : 1 \leq i' < i : nullable(\alpha_{i'}) \\ \quad \textbf{then } first(X) := first(X) \cup first(\alpha_i) \end{cases} \\ \textbf{until } FIRST(N) \textit{ is unchanged} \end{cases}$$

First And Follow

$follow(X)$ for any nonterminal symbol X is the set of terminal symbols a such that there exists a rightmost derivation of the form

$$S \Rightarrow^* \cdots Xa \cdots \Rightarrow^*$$

i.e. $follow(X)$ is the set of all terminal symbols that can occur immediately to the right of X in a rightmost derivation.
Notice that if there exists a a rightmost derivation of the form

$$S \Rightarrow^* \cdots X\alpha_1 \ldots \alpha_k a \cdots \Rightarrow^*$$

such that $\alpha_1, \ldots, \alpha_k$ are all nullable then again we have

$$S \Rightarrow^* \cdots X\alpha_1 \ldots \alpha_k a \cdots \Rightarrow^* \cdots Xa \cdots \Rightarrow^*$$

Algorithm 4.3

$\textsc{Follow}\ (N) \stackrel{df}{=}$

$$
\left\{
\begin{array}{l}
\quad \textbf{Require:}\ \text{CFG}\ G = \langle N, T, P, S\rangle \\
\quad \textbf{Yields:}\ FOLLOW(N \cup T) = \{follow(\alpha) \mid \alpha \in N \cup T\} \\
\textbf{for each}\ \alpha \in N \cup T \\
\quad \textbf{do}\ follow(\alpha) := \emptyset \\
\textbf{repeat} \\
\quad \textbf{for each}\ X \to \alpha_1 \ldots \alpha_k \in P \\
\qquad \textbf{do} \left\{
\begin{array}{l}
\textbf{for}\ i := 1\ \textbf{to}\ k \\
\qquad \textbf{do} \left\{
\begin{array}{l}
\textbf{if}\ \forall i' : i+1 \leq i' \leq k : nullable(\alpha_{i'}) \\
\quad \textbf{then}\ follow(\alpha_i) := follow(\alpha_i) \cup follow(X) \\
\textbf{for}\ j := i+1\ \textbf{to}\ k \\
\quad \textbf{do} \left\{
\begin{array}{l}
\textbf{if}\ \forall i' : i+1 \leq i' < j : nullable(\alpha_{i'}) \\
\quad \textbf{then}\ follow(\alpha_i) := follow(\alpha_i) \cup first(\alpha_j)
\end{array}
\right.
\end{array}
\right.
\end{array}
\right. \\
\textbf{until}\ FOLLOW(N \cup T)\ \textit{is unchanged}
\end{array}
\right.
$$

Recursive Descent Parsing: Pragmatics

- In any collection of (possibly) mutually recursive procedures, it is necessary to clearly specify the entry point into the collection and the exits. So we add a new start symbol S and a new end-of-file token EOF (represented by a new terminal symbol $\$$) with the unique production $S \rightarrow E\$$.

- Tokens are in upper case and the correspondence between the lexemes and tokens is as follows:

$$\begin{array}{lll}
ID(a) & \leftrightarrow a & , ID(b) & \leftrightarrow b \\
LPAREN & \leftrightarrow (& , RPAREN & \leftrightarrow) \\
MINUS & \leftrightarrow - & , DIVIDE & \leftrightarrow / \\
EOF & \leftrightarrow \$ & &
\end{array}$$

4.9. A recursive descent parser

```
program Parse;

   var input_token: token;

   function get_token:token;
   begin
      (* lex *)
   end;

   procedure match(expected);
      label 99;
   begin
      if input_token = expected then
      begin
         consume (input_token);
         if input_token <> EOF then input_token := get_token
         else goto 99
      end
      else parse_error
      99:
   end;
```

```
(* The system of mutually recursive procedures begins here *)

procedure Expression; Forward;

procedure Division (* D -> a | b | (E) *)
begin
   case input_token of
   ID: match(ID);
   LPAREN: Expression; match (RPAREN);
   else parse_error
end;


procedure Term_tail (* T' -> /DT' | <epsilon> *)
begin
   case input_token of
   DIVIDE: Division; Term_tail;
   MINUS: Term_tail; (* epsilon production *)
   RPAREN, ID: ; (* skip epsilon production *)
   else parse_error
end;

procedure Term (* T -> DT' *)
begin
   case input_token of
   ID(a), ID(b), LPAREN: Division; Term_tail;
   else parse_error
```

```
    end;

    procedure Expression_tail (* E' -> -TE' | <epsilon> *)
    begin
       case input_token of
       MINUS: Term; Expression_tail;
       RPAREN, ID: ; (* skip epsilon production *)
       else parse_error
    end;

   procedure Expression (* E -> TE' *)
    begin
       case input_token of
       ID(a), ID(b), LPAREN: Term; Expression_tail;
       else parse_error
    end;

begin (* main S -> E$ *)
   input_token := get_token;
   Expression; match (EOF)
end.
```

Bottom-Up Parsing Strategy

The main problem is to match parentheses of arbitrary nesting depths. This requires a stack[a] data structure to do the parsing so that unbounded nested parentheses and varieties of brackets may be matched.
Our basic parsing strategy is going to be based on a technique called *shift-reduce* parsing.

shift. Refers to moving the next token from the input token stream into a *parsing* stack.

reduce. Refers to applying a production rule in reverse i.e. given a production $X \rightarrow \alpha$ we reduce any occurrence of α in the parsing stack to X.

[a] In the case of recursive-descent parsing the stack is provided by the recursion facility in the language of implementation.

Reverse of Right-most Derivations

The result of a Bottom-Up Parsing technique is usually to produce a *reverse of the right-most derivation* of a sentence.

Example For the ambiguous grammar G_1 and corresponding to the right-most derivation 2 we get

$$\begin{array}{ll}
\mathbf{y} + \mathbf{4} * \mathbf{z} & \Leftarrow \\
I + \mathbf{4} * \mathbf{z} & \Leftarrow \\
E + \mathbf{4} * \mathbf{z} & \Leftarrow \\
E + C * \mathbf{z} & \Leftarrow \\
E + E * \mathbf{z} & \Leftarrow \\
E * \mathbf{z} & \Leftarrow \\
E * I & \Leftarrow \\
E * E & \Leftarrow \\
E & \Leftarrow
\end{array}$$

Fully Bracketed Expression

Consider an example of a fully bracketed expression generated by the simple left-recursive grammar defined earlier.

The main questions are

- When to **shift** and when to **reduce**?
- If **reduce** then what production to use?

Shift-reduce parsing: Invariant

Given a sentence generated by the grammar, at any stage of the parsing, the contents of the stack concatenated with the rest of the input token stream should be a sentential form of a right-most derivation of the sentence.

Parsing: FB0

r1. E $\longrightarrow$ E – T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

(a – (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Shift

Parsing: FB1

r1. E → E – T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

a – (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

(

Shift

Parsing: FB2

r1. E ⟶ E – T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

– (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

a

(

Shift

r1. E $\longrightarrow$ E – T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

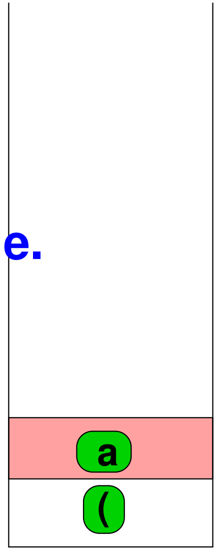
r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

– (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

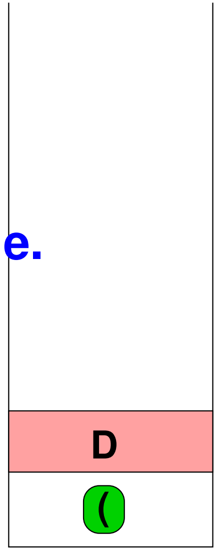
r4 T ⟶ D

r5 D ⟶ a | b | (E)

- (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

Parsing: FB5

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

- (a / b))

T

(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce

Parsing: FB6

r1. E ⟶ E – T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

– (a / b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

E

(

Shift

Parsing: FB7

r1. E → E - T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

(a / b))

-

E

(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Shift

Parsing: FB8

r1. E ⟶ E – T

r2 E ⟶ T

r3 T ⟶ T / D

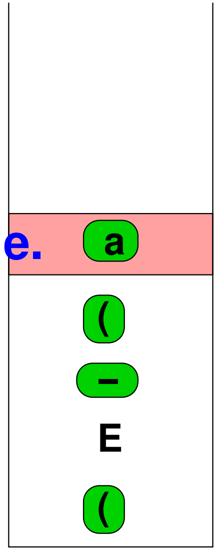
r4 T ⟶ D

r5 D ⟶ a | b | (E)

/ b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

Parsing: FB9

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

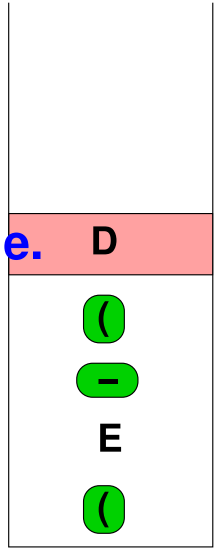
r4 T ⟶ D

r5 D ⟶ a | b | (E)

/ b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

Parsing: FB10

r1. $E \longrightarrow E \; - \; T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$

/ b))

T

(

–

E

(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce?

Parsing: FB11

r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \,/\, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$

/ b))

T
(
–
E
(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

~~Reduce~~? Shift

Parsing: FB12

r1. E $\longrightarrow$ E – T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

b))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

/
T
(
–
E
(

Shift

r1. E → E − T

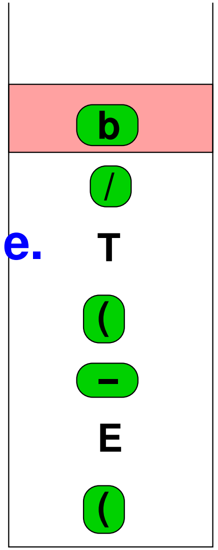
r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

))



Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce

Parsing: FB14

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

))

D
/
T
(
-
E
(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce

Parsing: FB15

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

))

D
/
T
(
-
E
(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce?

r1. E → E – T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

))

D
/
T
(
–
E
(

Principle:

Reduce whenever possible. Shift only when reduce is impossible

No, REDUCE!

r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

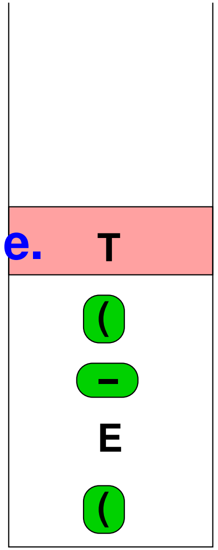
r4 T ⟶ D

r5 D ⟶ a | b | (E)

))

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce?

Parsing: FB18

r1. E $\longrightarrow$ E – T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

))

Principle:

Reduce whenever possible. Shift only when reduce is impossible

E
(
–
E
(

Shift

Parsing: FB19

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

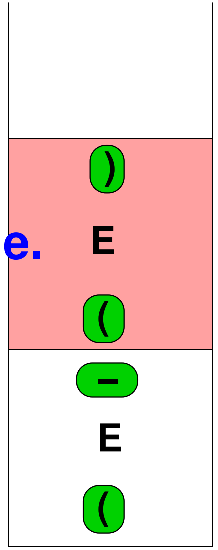
r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

)

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

Parsing: FB20

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

)

Principle:

Reduce whenever possible. Shift only when reduce is impossible

D

-

E

(

Reduce

Parsing: FB21

r1. E $\longrightarrow$ E – T

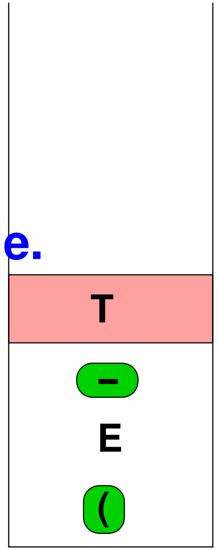
r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

)



Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce?

Parsing: FB22

r1. E → E – T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

)

Principle:

Reduce whenever possible. Shift only when reduce is impossible

T
–
E
(

No, REDUCE!

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

)

Principle:

Reduce whenever possible. Shift only when reduce is impossible

E

(

Shift

Parsing: FB24

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

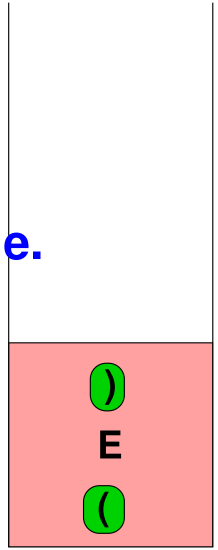
r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

Parsing: FB25

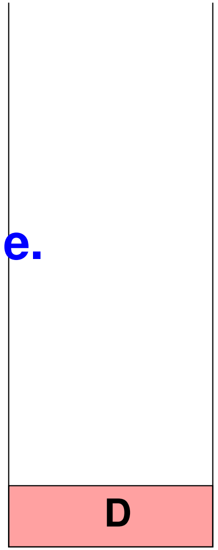
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Principle:

Reduce whenever possible. Shift only when reduce is impossible

Reduce

Parsing: FB26

r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

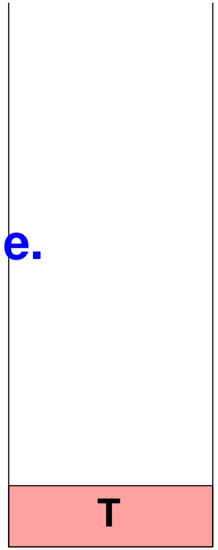
r3 $T \longrightarrow T / D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Reduce

r1. E ⟶ E – T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

Principle:

Reduce whenever possible. Shift only when reduce is impossible

E

Reduce

Unbracketed Expression

Consider an example of an unbracketed expression which relies on the precedence rules as defined in the grammar.

Parsing: UB0

```
r1. E ⟶ E - T
r2  E ⟶ T
r3  T ⟶ T / D
r4  T ⟶ D
r5  D ⟶ a | b | ( E )
```

a - a / b

Parsing: UB1

r1. E → E - T

r2 E → T

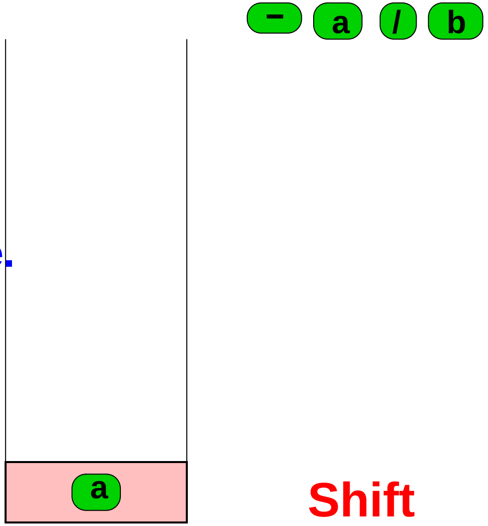
r3 T → T / D

r4 T → D

r5 D → a | b | (E)

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Shift

Parsing: UB2

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

- a / b

D

Reduce by r5

Parsing: UB3

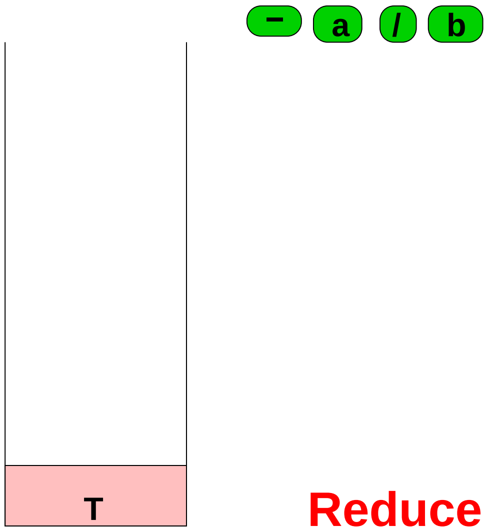
r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Reduce by r4

Parsing: UB4

r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

− a / b

E

Reduce by r2

Parsing: UB5

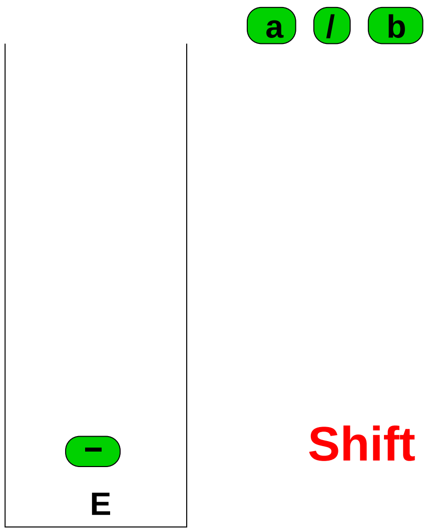
r1. E ⟶ E – T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Parsing: UB6

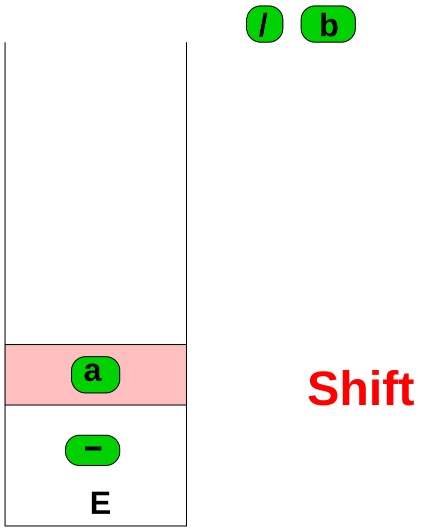
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parsing: UB7

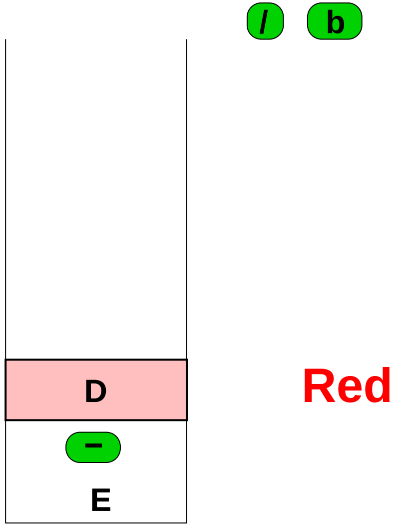
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Reduce by r5

Parsing: UB8

r1. E ⟶ E − T

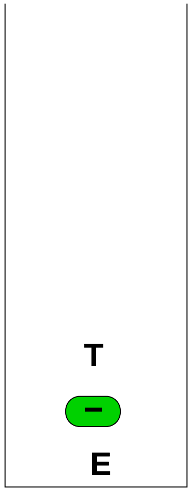
r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

/ b



Reduce by r4

Parsing: UB8a

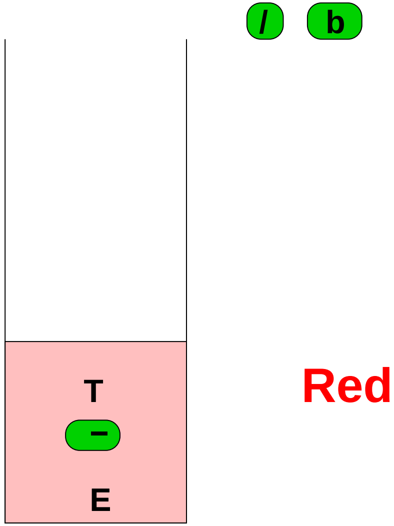
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Reduce by r4

Parsing: UB9a

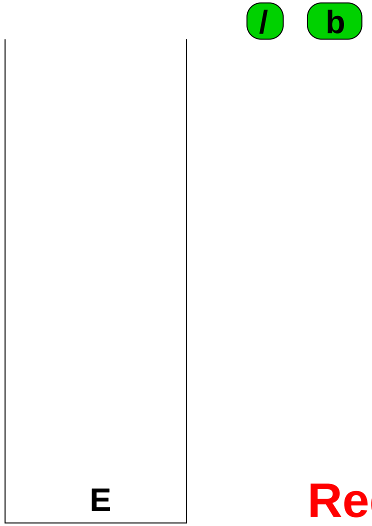
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Reduce by r1

Parsing: UB10a

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

b

/

E

Shift

Parsing: UB11a

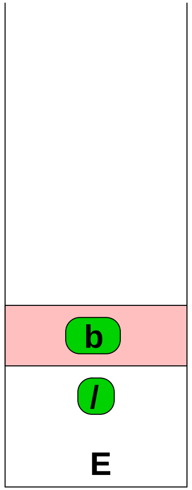
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Shift

Parsing: UB12a

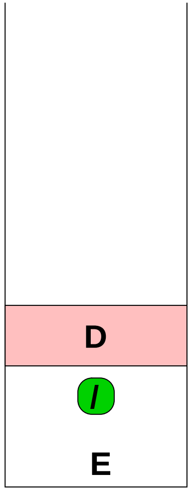
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Reduce by r5

Parsing: UB13a

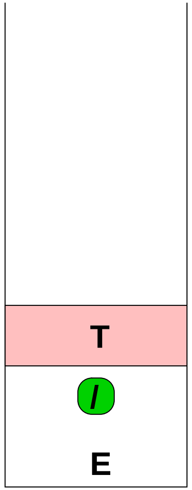
r1. E $\longrightarrow$ E **-** T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T **/** D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ **a** | **b** | **(** E **)**



Reduce by r4

Parsing: UB14a

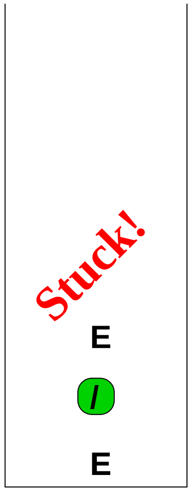
r1. E $\longrightarrow$ E $-$ T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Get back!

Reduce by r2

Parsing: UB14b

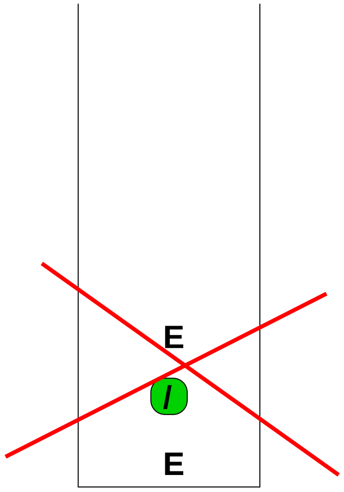
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Get back!

Reduce by r2

Parsing: UB13b

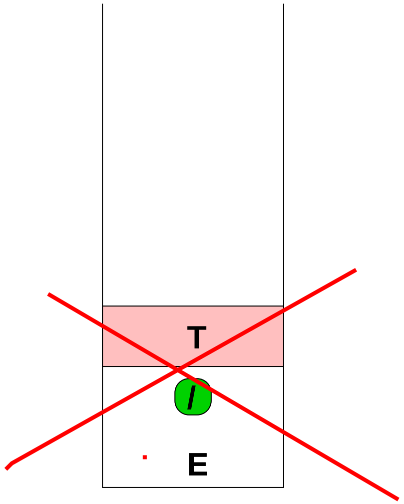
r1. E → E - T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)



Get back!

Reduce by r4

Parsing: UB12b

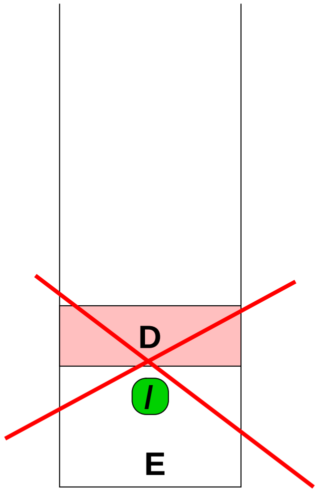
r1. E $\longrightarrow$ E $-$ T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Get back!

Reduce by r5

Parsing: UB11b

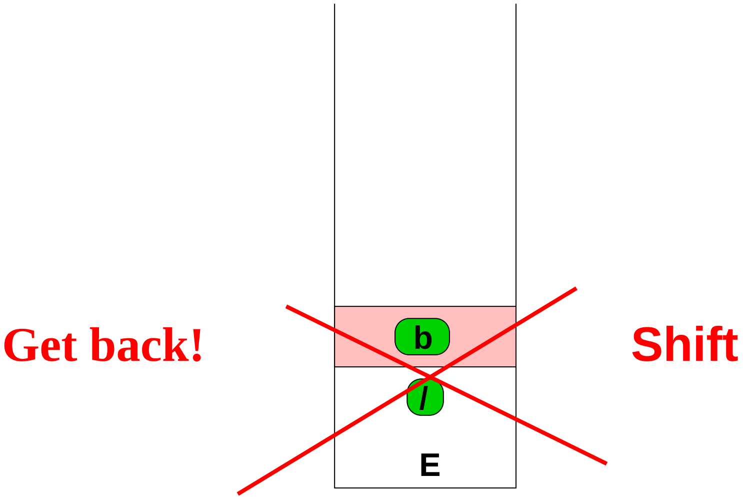
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

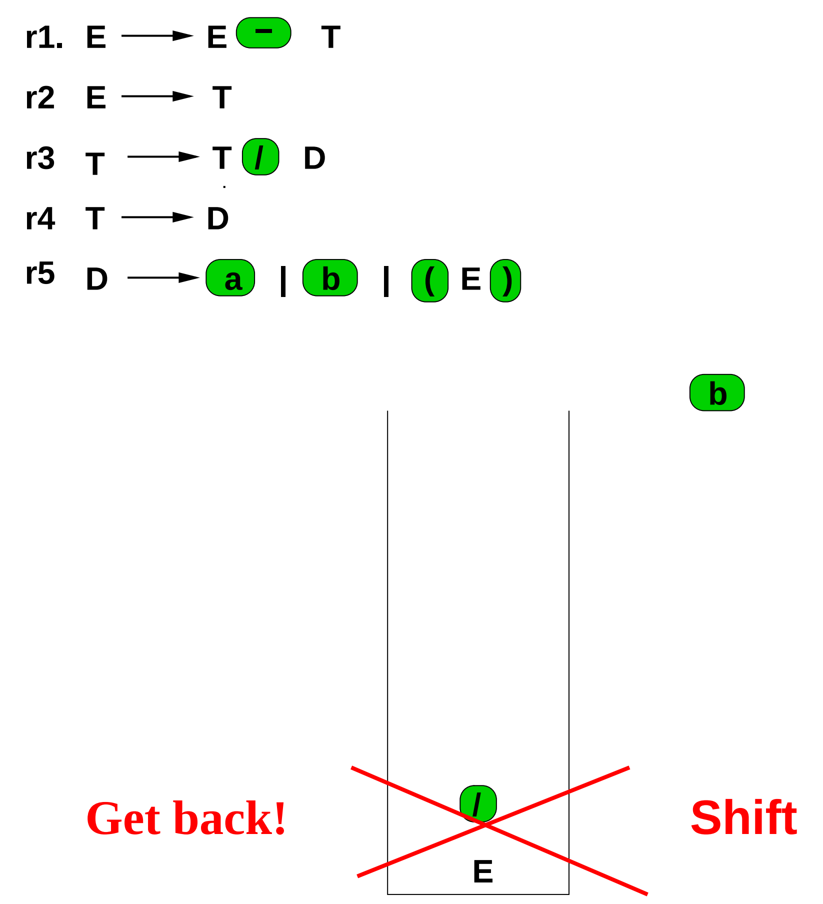
r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parsing: UB10b



Parsing: UB9b

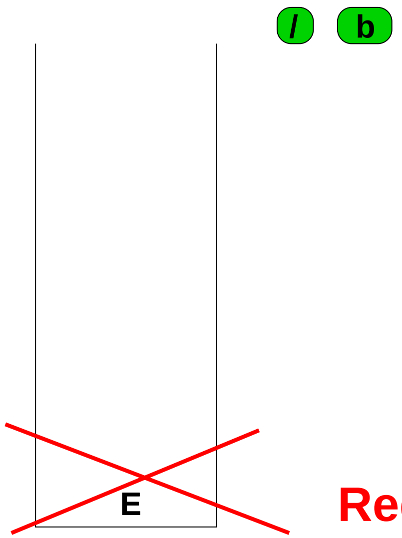
r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Get back to where you once belonged!

Reduce by r1

Parsing: UB8b

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

modified Principle:

Reduce whenever possible, but

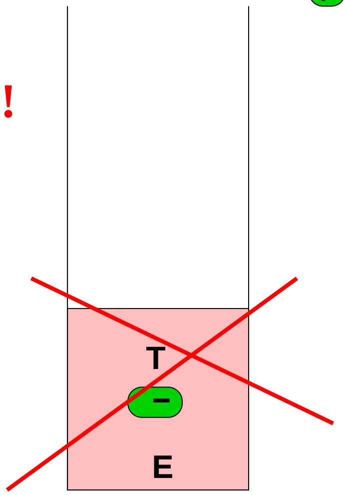
but depending upon

lookahead

/ b

Shift instead of reduce here!

Shift−reduce conflict



Reduce by r4

Parsing: UB8

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

/ b

Stack
T
−
E

Reduce by r4

Parsing: UB9

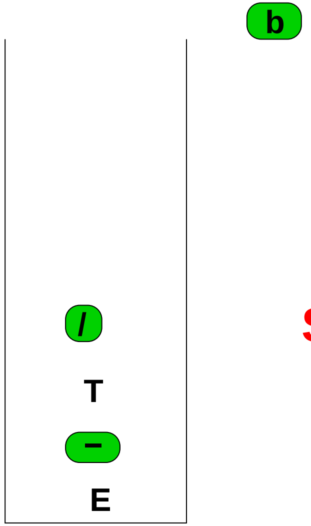
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Shift

Parsing: UB10

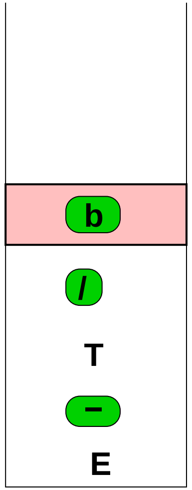
r1. E $\longrightarrow$ E `-` T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T `/` D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ `a` | `b` | `(` E `)`



Shift

Parsing: UB11

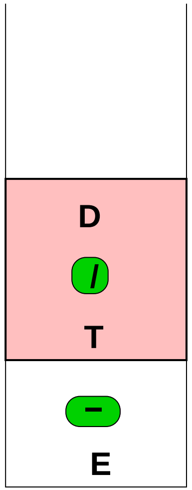
r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Reduce by r5

Parsing: UB12

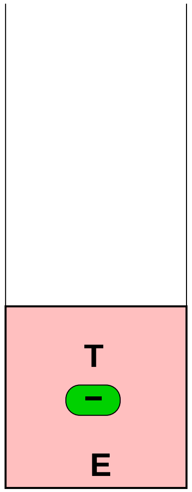
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Reduce by r3

Parsing: UB13

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

E

Reduce by r1

Bottom-Up Parsing

r1. $E \longrightarrow E$ - T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ / D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ a | b | (E)

a - a / b

r1. $E \longrightarrow E\ -\ T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T\ /\ D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\ E\ )$

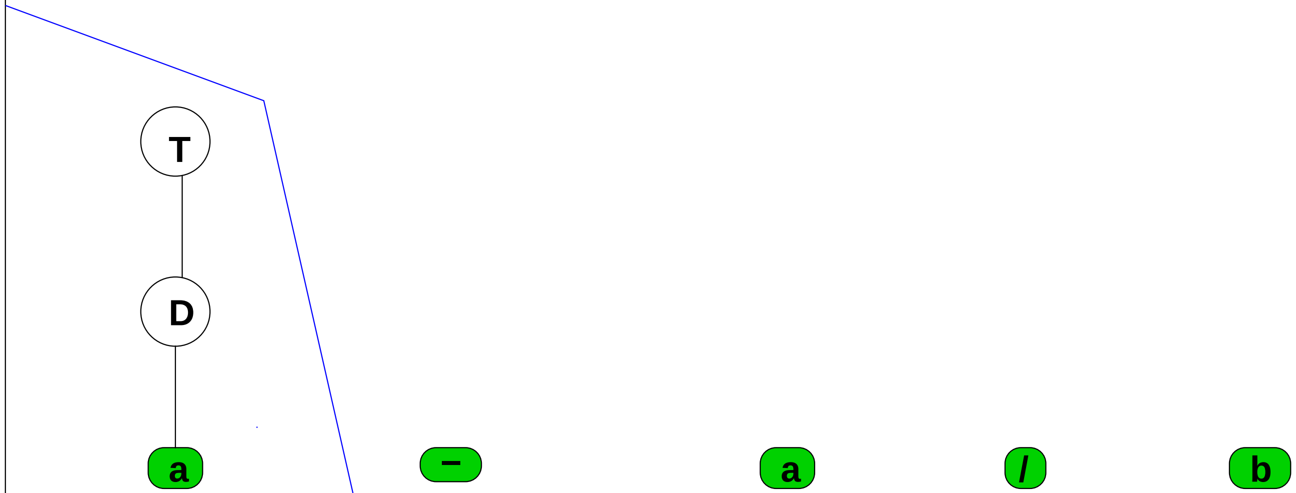
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



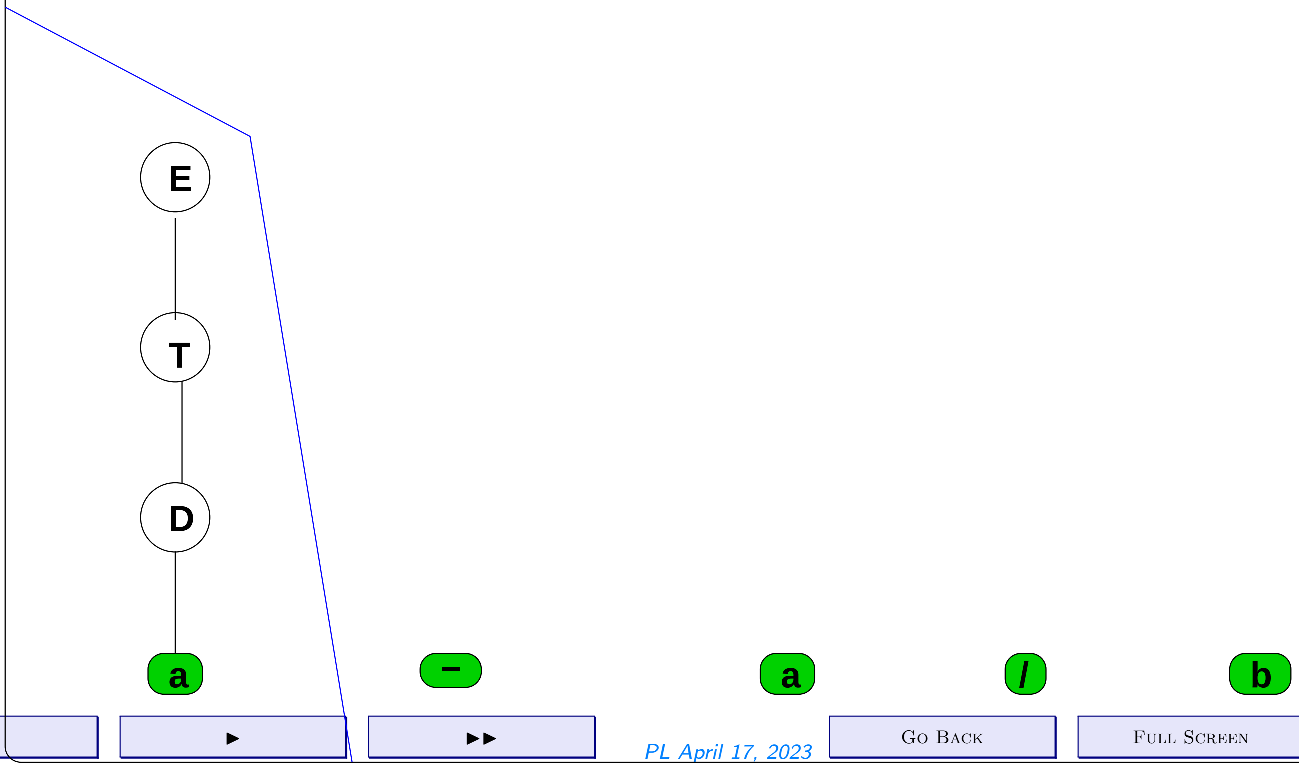
r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



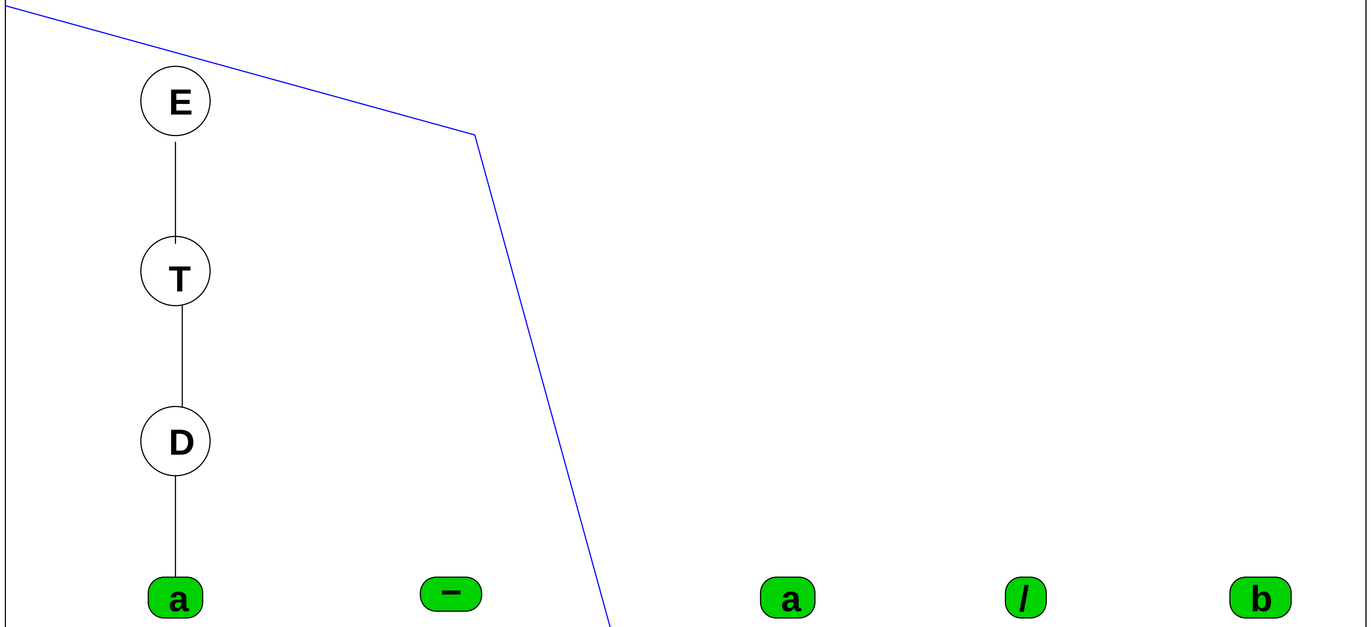
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



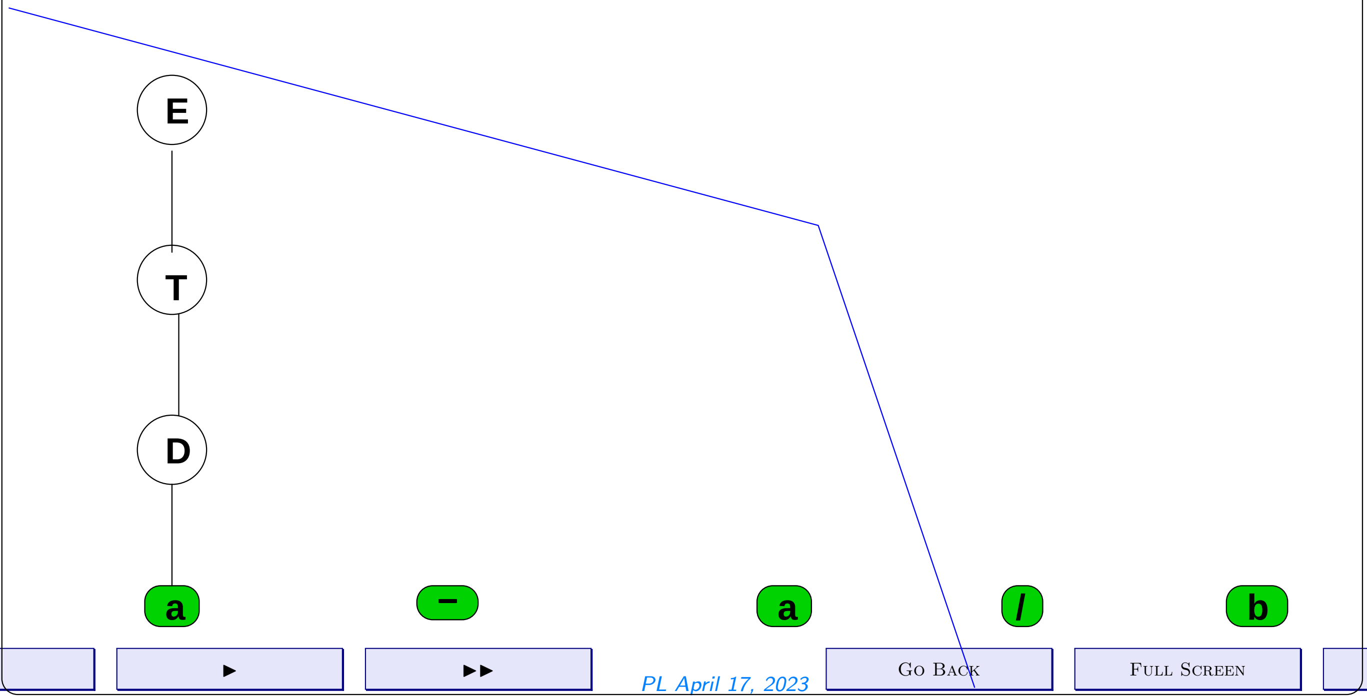
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \,/\, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\, E \,)$



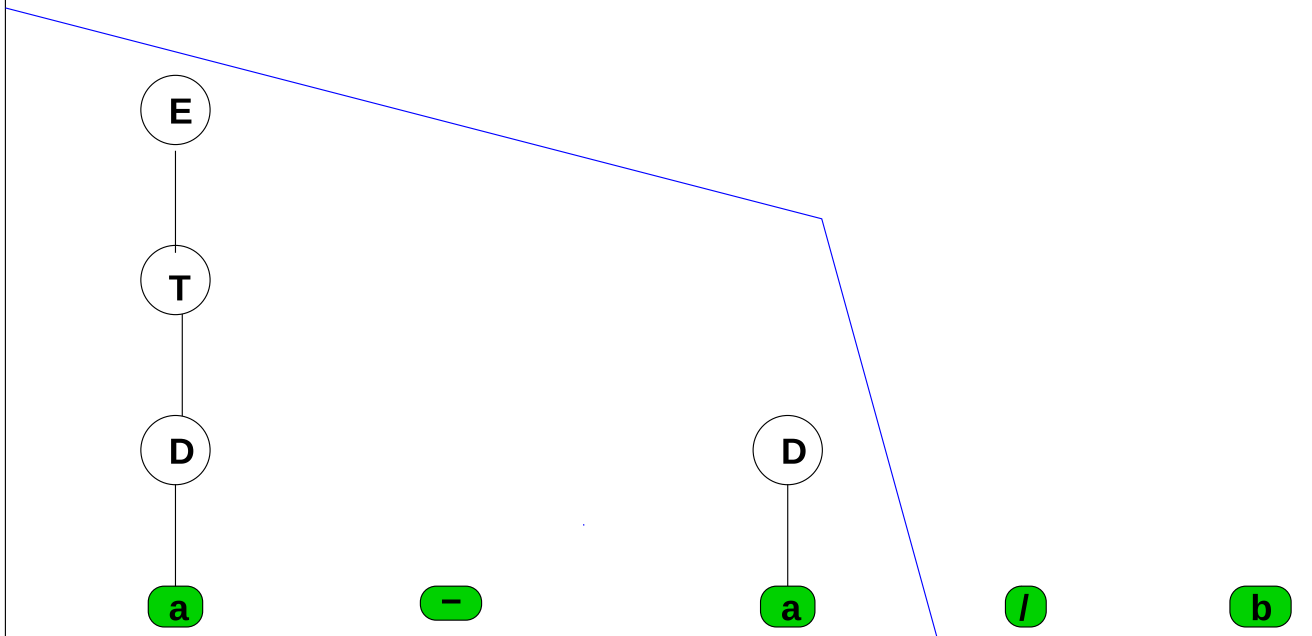
r1. $E \longrightarrow E$ **-** T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ **/** D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ **a** | **b** | **(** E **)**



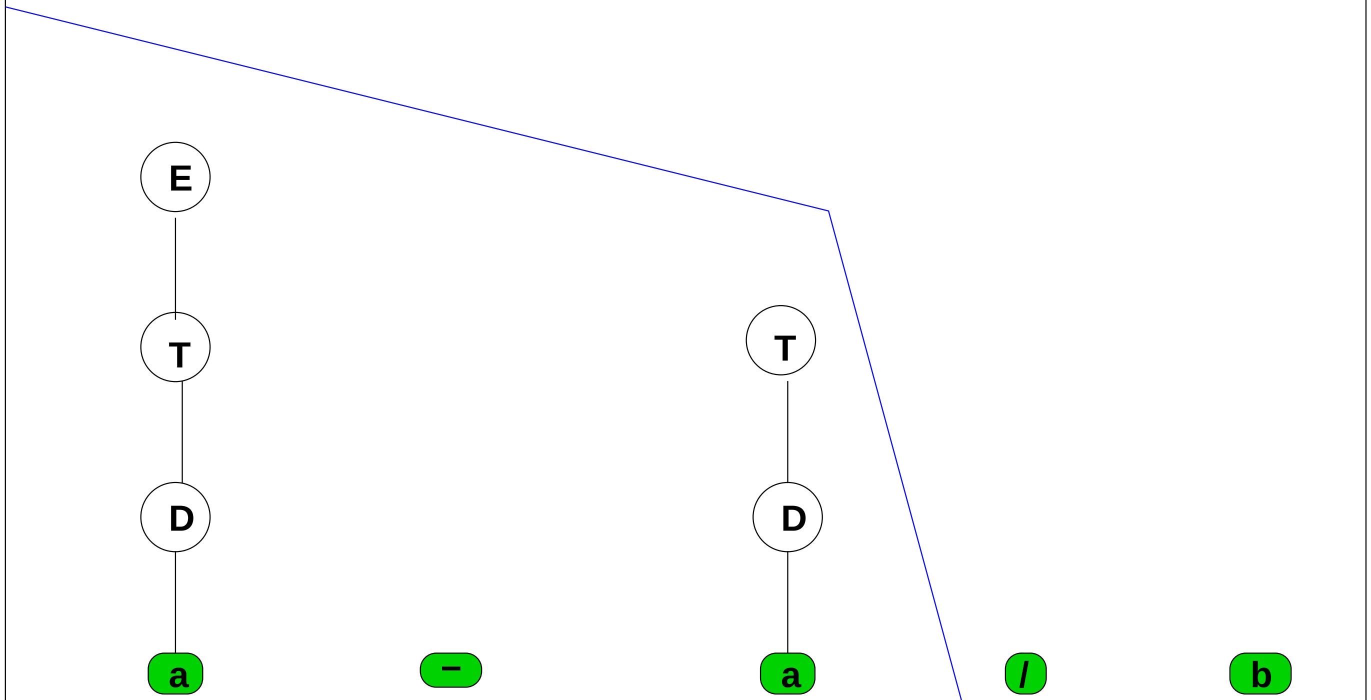
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



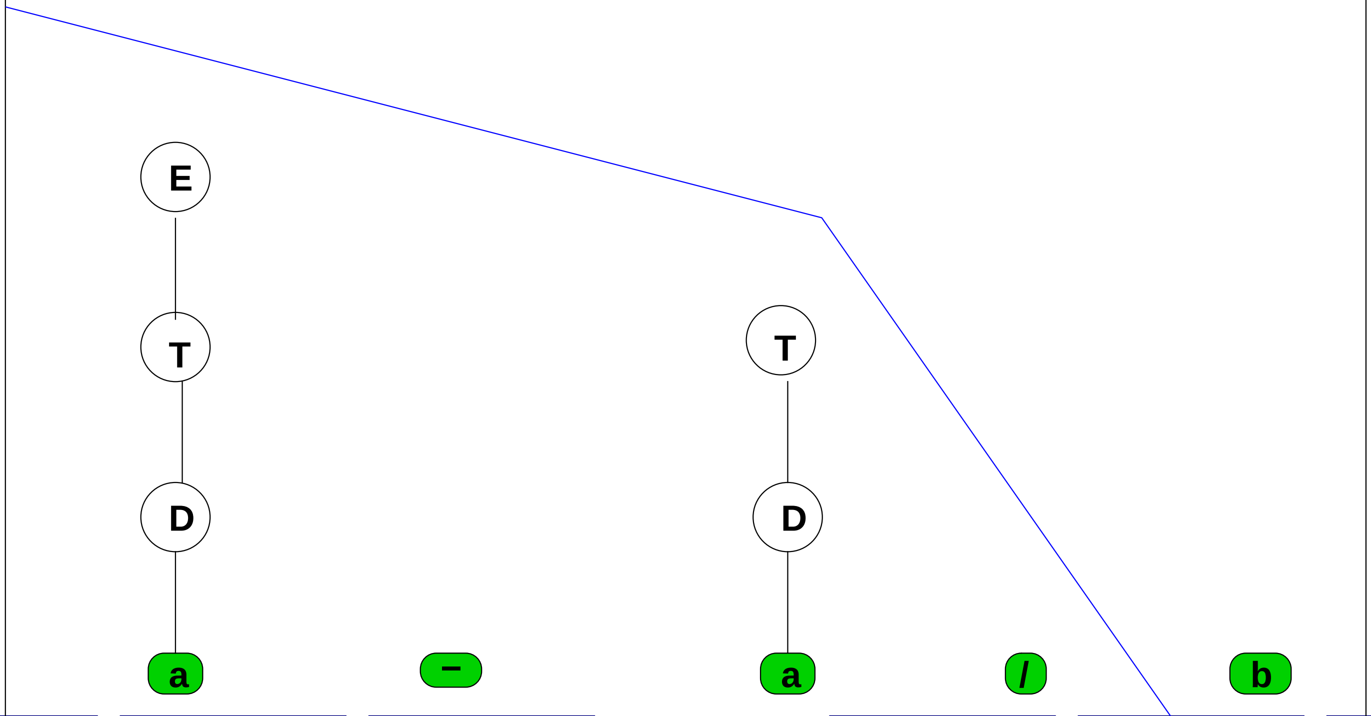
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



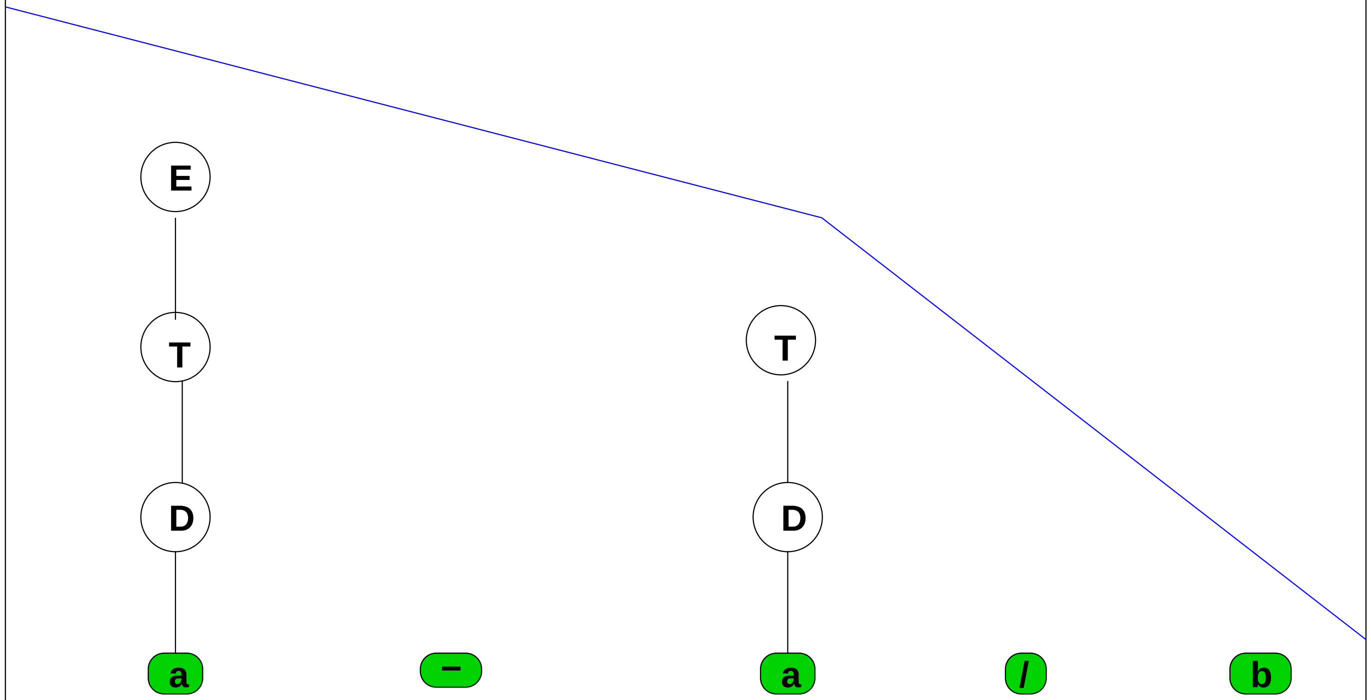
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \, / \, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



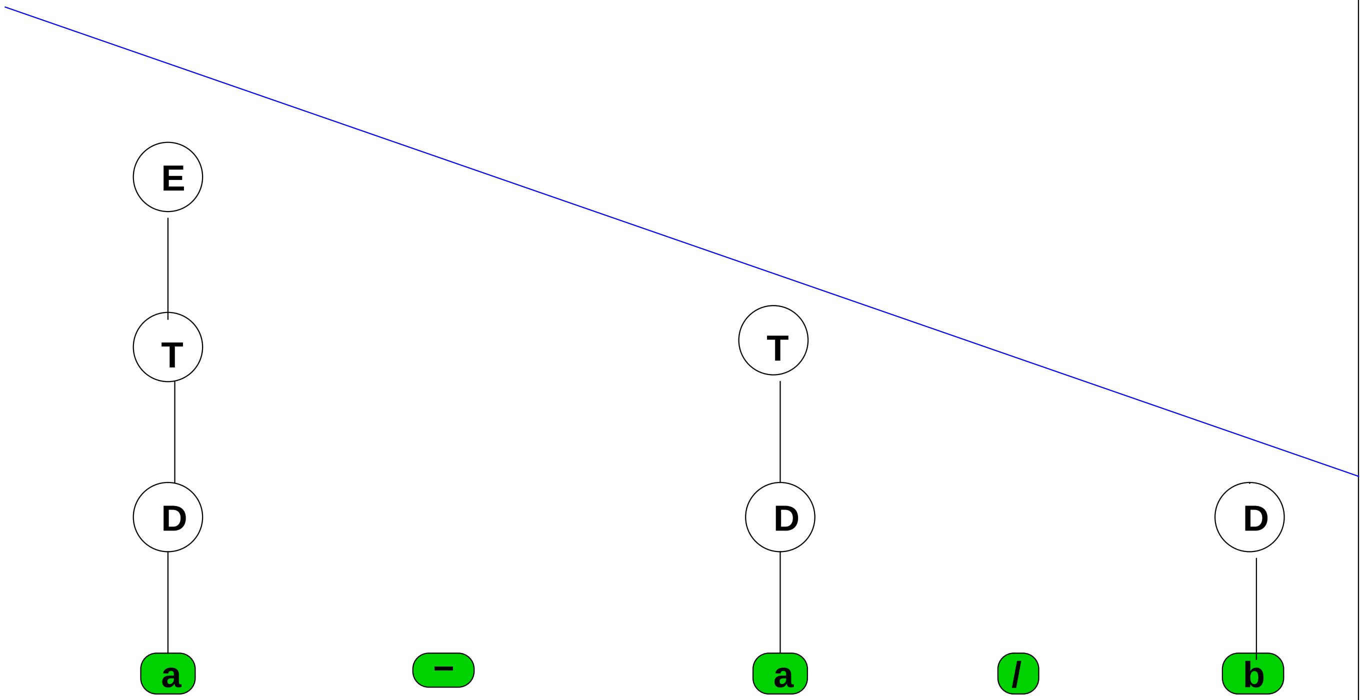
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



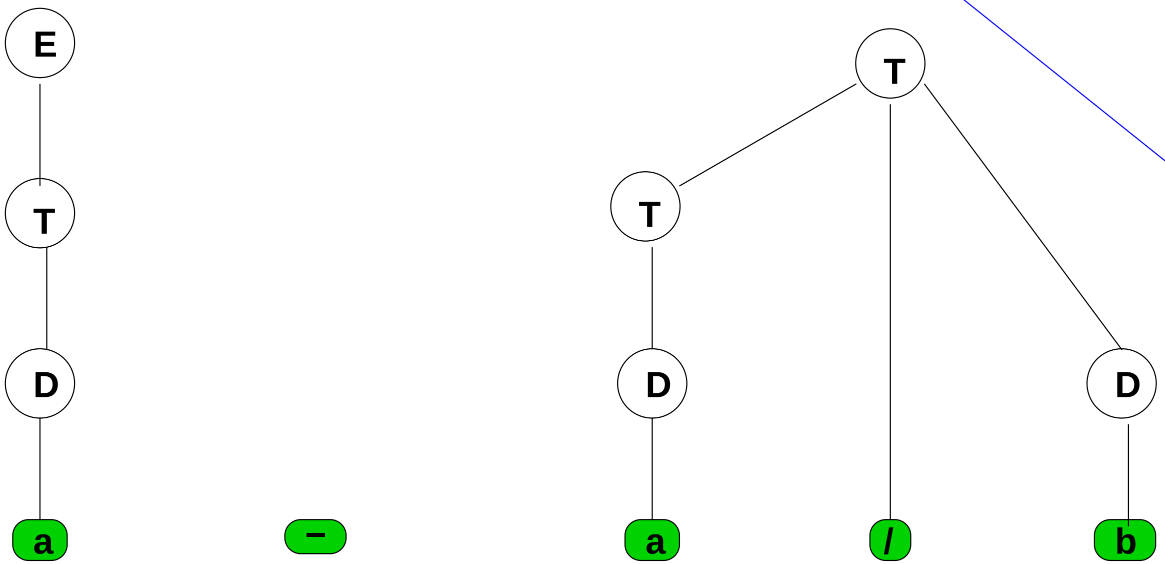
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \, / \, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



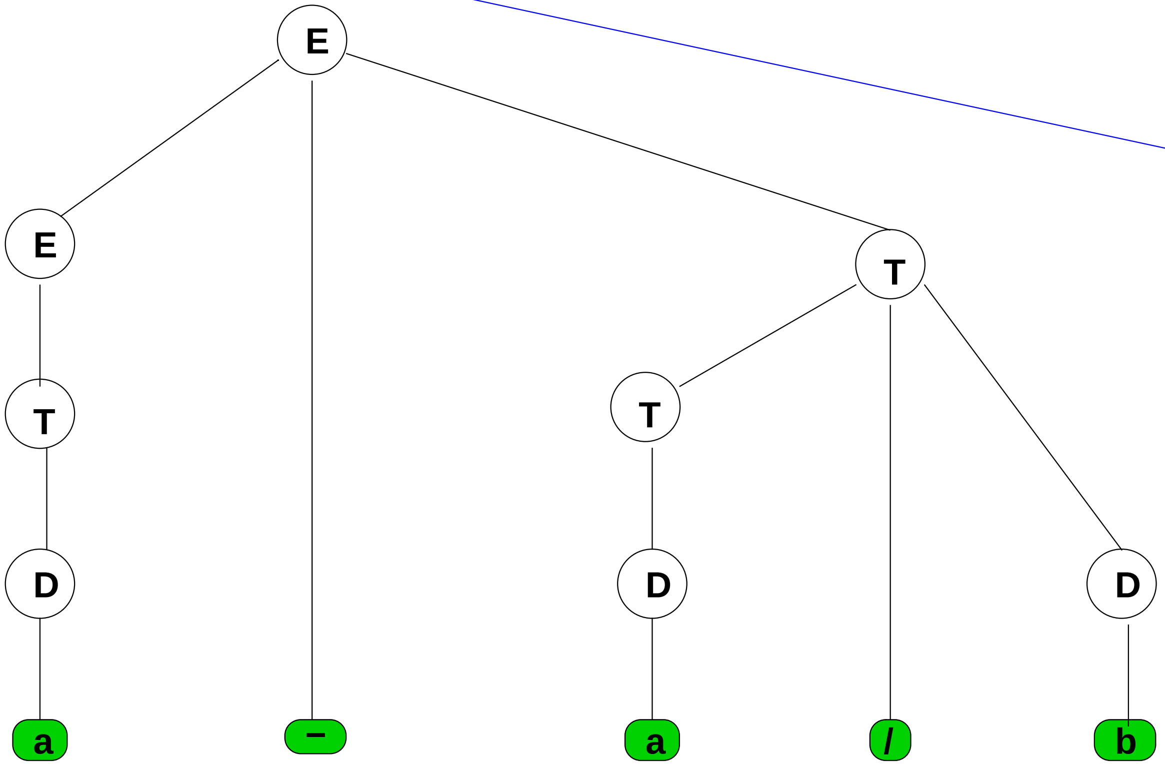
r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)



Parsing: Summary: 1

- All high-level languages are designed so that they may be parsed in this fashion with only a single token look-ahead.
- Parsers for a language can be automatically constructed by parser-generators such as Yacc, Bison, ML-Yacc and CUP in the case of Java.
- Shift-reduce conflicts if any, are automatically detected and reported by the parser-generator.
- Shift-reduce conflicts may be avoided by suitably redesigning the context-free grammar.

Parsing: Summary: 2

- Very often shift-reduce conflicts may occur because of the prefix problem. In such cases many parser-generators resolve the conflict in favour of shifting.
- There is also a possiblility of reduce-reduce conflicts. This usually happens when there is more than one nonterminal symbol to which the contents of the stack may reduce.
- A minor reworking of the grammar to avoid redundant non-terminal symbols will get rid of reduce-reduce conflicts.

The Big Picture

Parsing Problems 1

The main question in shift-reduce parsing is:

When to *shift* and when to *reduce*?

To answer this question we require

- more information from the input token stream,
- to look at the rest of the input token stream and then take a decision.

But the decision has to be automatic. So the parser requires some rules. Once given the rules we may construct the parser to follow the rules.

Parsing Problems 2

But for a very large program it may be impossible to look at *all* the input before taking a decision. So clearly the parser can look at only a limited amount of the input to take a decision. So
The next question:

How much of the input token stream would the parser require?

Disregarding the very next input token as always available, the length of the extra amount of input required for a shift-reduce decision is called the lookahead.

Parsing Problems 3

Once all the input has been read, the parser should be able to decide

in case of a valid sentence that it should only apply reduction rules and attempt to reach the start symbol of the grammar only through reductions and

in case of an invalid sentence that a grammatical error has occurred in the parsing process

To solve this problem we augment every grammar with a new start symbol S and a new terminal token $\$$ and a new special rule. For our previous grammar we have the new rule

$$\boxed{S \rightarrow E\$}$$

Augmented Grammar

Consider the following (simplified) augmented grammar with a single binary operator $-$ and parenthesis. We also number the rules.

$$
\begin{array}{llll}
1. & S & \rightarrow & E\$ \\
2. & E & \rightarrow & E-T \\
3. & E & \rightarrow & T \\
4. & T & \rightarrow & a \\
5. & T & \rightarrow & (E)
\end{array}
$$

In an augmented grammar the start symbol does not occur on the right hand side of any production.

LR(0) Languages

LR(0) languages are those context-free languages that may be parsed by taking *deterministic shift-reduce decisions* only based on the contents of the parsing stack and without viewing any lookahead.

- "L" refers to reading the input from *left to right*,
- "R" refers to the *(reverse) of rightmost derivation*
- "0" refers to *no-lookahead*..
- Many simple CFLs are LR(0). But the LR(0) parsing method is too weak for most high-level programming languages.
- But understanding the LR(0) parsing method is most crucial for understanding other more powerful LR-parsing methods which require lookaheads for deterministic *shift-reduce* decision-making

LR-Parsing Invariant

In any LR-parsing technique the following invariant holds.

For any syntactically valid sentence generated by the augmented grammar, the concatenation of the stack contents with the rest of the input gives a sentential form of a rightmost derivation.

Hence at any stage of the parsing if $\alpha \in (N \cup T)^*$ is the contents of the parsing stack and $x \in T^*\$$ is the rest of the input that has not yet been read, then αx is a sentential form of a right-most derivation.

LR(0) Item

An LR(0) item consists of an LR(0) production rule with a special marker $\blacktriangle$ on the right hand side of rule.

- The marker is different from any of the terminal or nonterminal symbols of the grammar.
- The marker separates the contents of the stack from the expected form of some prefix of the rest of the input.
- Given a rule $X \to \alpha$, where X is a nonterminal symbol and α is a string consisting of terminal and non-terminal symbols, an LR(0) item is of the form

$$\boxed{X \to \beta_\blacktriangle\gamma}$$

where $\alpha = \beta\gamma$.

- For each rule $X \to \alpha$, there are $|\alpha| + 1$ distinct LR(0) items – one for each position in α.

What does an LR(0) item signify?

The LR(0) item

$$\boxed{X \to \beta_{\blacktriangle}\gamma}$$

signifies that at some stage of parsing

- β is the string (of terminals and nonterminals) on the top of the stack and
- some prefix of the rest of the input can be generated by the sentential form γ

so that whenever $\beta\gamma$ appears on the stack, $\beta\gamma$ may be reduced immediately to X.

LR0 Parsing Strategy

The LR0 parsing strategy is to

1. construct a DFA whose alphabet is $N \cup T \cup \{\$\}$
2. use the parsing stack to perform reductions at appropriate points

The LR0 parsing table is hence a DFA with 3 kinds of entries.

shift i in which a terminal symbol is shifted on to the parsing stack and the DFA moves to state i.

reduce j a reduction using the production rule j is performed

goto k Based on the contents of the stack, the DFA moves to state k.

Favourite Example

Consider our favourite augmented grammar

$$
\begin{array}{llcl}
1. & S & \rightarrow & E\$ \\
2. & E & \rightarrow & E{-}T \\
3. & E & \rightarrow & T \\
4. & T & \rightarrow & a \\
5. & T & \rightarrow & (E)
\end{array}
$$

Rule 1: Items

Rule 1

$$R1.\ S \rightarrow E\$$$

has the following three items

$$I1.1\ S \rightarrow \blacktriangle E\$$$
$$I1.2\ S \rightarrow E\blacktriangle\$$$
$$I1.3\ S \rightarrow E\$\blacktriangle$$

one for each position on the right hand side of the rule.

Rule 2: Items

Rule 2

$$R2.\ E \rightarrow E{-}T$$

has the following items

$$\begin{array}{llcl} I2.1 & E & \rightarrow & \blacktriangle E{-}T \\ I2.2 & E & \rightarrow & E\blacktriangle{-}T \\ I2.3 & E & \rightarrow & E{-}\blacktriangle T \\ I2.4 & E & \rightarrow & E{-}T\blacktriangle \end{array}$$

Rule 3: Items

Rule 3

$$R3.\ E \rightarrow T$$

has just the items

$$I3.1\ E \rightarrow\ \blacktriangle T$$
$$I3.2\ E \rightarrow T \blacktriangle$$

Rule 4: Items

Rule 4

$$R4.\ T \rightarrow a$$

has the items

$$I4.1\ T \rightarrow \blacktriangle a$$
$$I4.2\ T \rightarrow a \blacktriangle$$

Rule 5: Items

Rule 5

$$R5.\ T \rightarrow (E)$$

has the items

$$I5.1\ T \rightarrow \blacktriangle(E)$$
$$I5.2\ T \rightarrow (\blacktriangle E)$$
$$I5.3\ T \rightarrow (E\blacktriangle)$$
$$I5.4\ T \rightarrow (E)\blacktriangle$$

Significance of I1.*

I1.1 $S \to {}_\blacktriangle E\$$. Hence

1. The parsing stack is empty and
2. the entire input (which has not been read yet) should be reducible to E followed by the $\$$.

I1.2 $S \to E_\blacktriangle\$$. Hence

1. E is the only symbol on the parsing stack and
2. the rest of the input consists of the terminating symbol $\$$.

I1.3 $S \to E\$_\blacktriangle$. Hence

1. There is no input left to be read and
2. the stack contents may be reduced to the start symbol

DFA States: Initial and Final

- Clearly the *initial* state S1 of the DFA will correspond to item I1.1.
- There should be a state corresponding to item I1.2.
- There should be a **goto** transition on the nonterminal symbol E from the *initial state* (corresponding to item I1.1) to the state corresponding to item I1.2.
- The *accepting* state of the DFA will correspond to item item I1.3.
- There would also be a **shift** transition on $\$$ from the state corresponding to item I1.2 to the accepting state corresponding to item I1.3.
- There should be a **reduce** action using rule 1 when the DFA reaches the state corresponding to item I1.3.

Input Possibilities: 1

Consider item I1.1.

1. How will a grammatically valid sentence input reduce to $E\$$?

   From the grammar it is obvious that this can happen *only if* the input is of a form such that

   (a) it can be reduced to $E-T$ (recursively) or

   (b) it can be reduced to T

2. How can the input be reduced to the form T?

3. How can the input be reduced to the form $E-T$?

Input Possibilities: 2

Consider item I1.1.

1. How will a grammatically valid sentence input reduce to $E\$$?
2. How can the input be reduced to the form T?
   (a) If the enire input consists of only a then it could be reduced to T or
   (b) If the entire input could be reduced to the form (E) then it could be reduced to T.
3. How can the input be reduced to the form $E{-}T$?

Input Possibilities: 3

Consider item I1.1.

1. How will a grammatically valid sentence input reduce to $E\$$?
2. How can the input be reduced to the form T?
3. How can the input be reduced to the form $E-T$?
   (a) If the entire input could be split into 3 parts α, β and γ such that
      i. α is a prefix that can be reduced to E, and
      ii. $\beta = -$, and
      iii. γ is a suffix that can be reduced to T

      then it could be reduced to $E-T$

Closures of Items

Theoretically each item is a state of a NFA. The above reasoning leads to forming closures of items to obtain DFA states, in a manner similar to the the subset construction. Essentially all NFA states with similar initial behaviours are grouped together to form a single DFA state.

NFA to DFA construction

Algorithm 4.4

$\textsc{ClosureOfItems}\ (I) \stackrel{df}{=}$

$$\left\{\begin{array}{l}
\quad \textbf{Requires: } \text{Set } I \subseteq \mathcal{I} \text{ of LR(0) items of a CFG with rule set } P \\
\quad \textbf{Ensures: } \text{Closure of } I \text{ for a subset } I \subseteq \mathcal{I} \text{ of items} \\
\textbf{repeat} \\
\ \textbf{for each } A \to \alpha_{\blacktriangle} X\beta \in I \\
\quad \textbf{do} \left\{\begin{array}{l} \textbf{for each } X \to \gamma \in P \\ \quad \textbf{do } I := I \cup \{X \to {}_{\blacktriangle}\gamma\} \end{array}\right. \\
\textbf{until } \textit{no more changes occur in } I
\end{array}\right.$$

State Changes on Nonterminals

As in the case of the NFA to DFA construction with each state transition we also need to compute closures on the target states.

Algorithm 4.5

$\text{GoTo}\ (I, X) \stackrel{df}{=}$

$$\left\{\begin{array}{l} \quad \textbf{Requires: } I \subseteq \mathcal{I} \text{ of LR(0) items of a CFG } G = \langle N, T, P, S \rangle,\ X \in N \\ \quad \textbf{Ensures: } \text{States of the DFA: Each state in the DFA is a closure of items} \\ J := \emptyset; \\ \textbf{for each } A \to \alpha_\blacktriangle X\beta \in I \\ \quad \textbf{do } J := J \cup \{A \to \alpha X_\blacktriangle \beta\}; \\ K := \text{ClosureOfItems}(J); \\ \textbf{return } (K) \end{array}\right.$$

State S1

$$S1 = \textsc{ClosureOfItems}(\{S \rightarrow {\blacktriangle} E\$\}) \\ = \{S \rightarrow {\blacktriangle} E\$, E \rightarrow {\blacktriangle} E-T, E \rightarrow {\blacktriangle} T, \\ T \rightarrow {\blacktriangle} a, T \rightarrow {\blacktriangle} (E)\}$$

$$\boxed{S1 \xrightarrow{(} \textsc{ClosureOfItems}(\{T \rightarrow ({\blacktriangle} E)\}) = S2}$$

$$\boxed{S1 \xrightarrow{E} \textsc{ClosureOfItems}(\{S \rightarrow E{\blacktriangle}\$, E \rightarrow E{\blacktriangle}-T\}) = S3}$$

$$\boxed{S1 \xrightarrow{T} \textsc{ClosureOfItems}(\{E \rightarrow T{\blacktriangle}\}) = S7}$$

$$\boxed{S1 \xrightarrow{a} \textsc{ClosureOfItems}(\{T \rightarrow a{\blacktriangle}\}) = S8}$$

State S2

$$S2 = \textsc{ClosureOfItems}(\{T \to (\blacktriangle E)\})$$
$$= \{T \to \blacktriangle(E), E \to \blacktriangle E-T, E \to \blacktriangle T,$$
$$T \to \blacktriangle a, T \to \blacktriangle(E)\}$$

$$\boxed{S2 \xrightarrow{(} \textsc{ClosureOfItems}(\{T \to (\blacktriangle E)\}) = S2}$$

$$\boxed{S2 \xrightarrow{E} \textsc{ClosureOfItems}(\{T \to (E\blacktriangle), E \to E\blacktriangle -T\}) = S9}$$

$$\boxed{S2 \xrightarrow{T} \textsc{ClosureOfItems}(\{E \to T\blacktriangle\}) = S7}$$

$$\boxed{S2 \xrightarrow{a} \textsc{ClosureOfItems}(\{T \to a\blacktriangle\}) = S8}$$

$$
\begin{aligned}
S3 &= \textsc{ClosureOfItems}(\{S \to E_\blacktriangle \$, E \to E_\blacktriangle - T\}) \\
&= \{S \to E_\blacktriangle \$, E \to E_\blacktriangle - T\}
\end{aligned}
$$

However,

$$\boxed{S3 \xrightarrow{-} \textsc{ClosureOfItems}(\{E \to E -_\blacktriangle T\})}$$

and

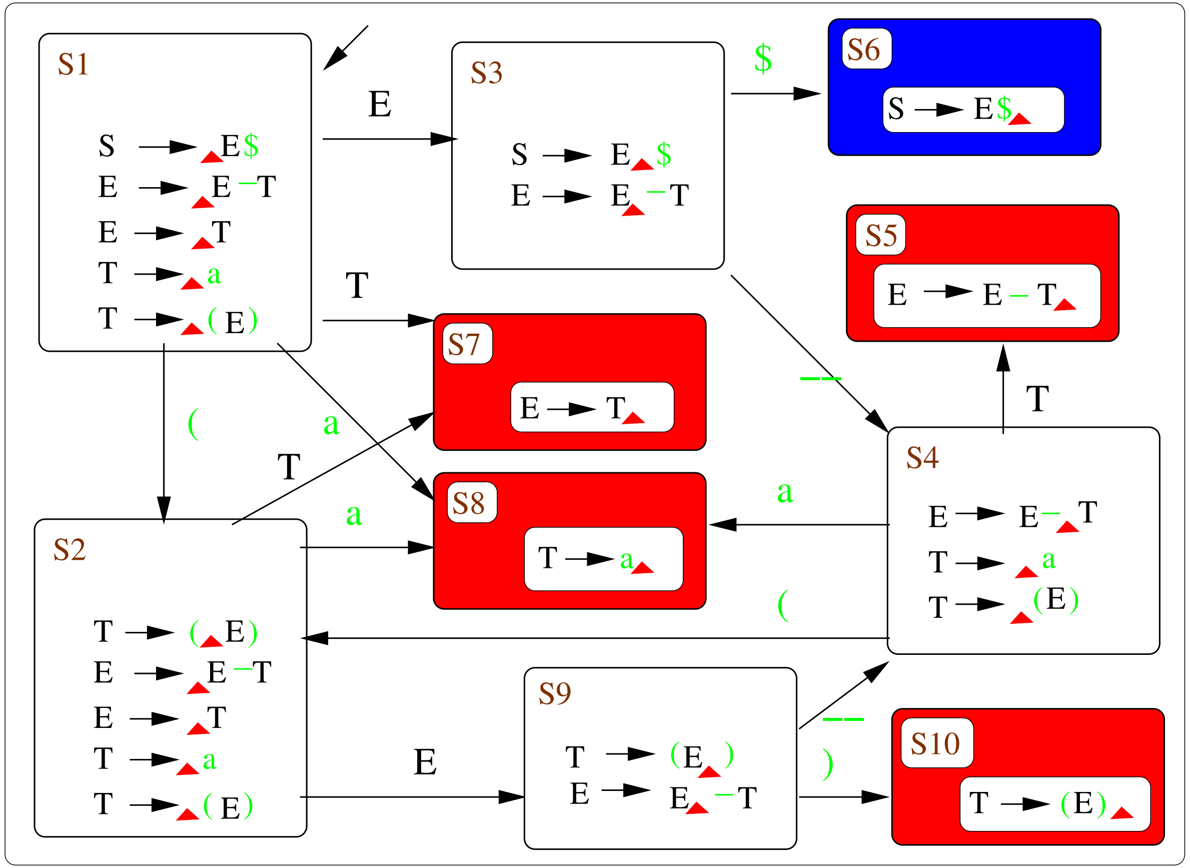
$$
\begin{aligned}
&\textsc{ClosureOfItems}(\{E \to E -_\blacktriangle T\}) \\
&= \{E \to E -_\blacktriangle T, T \to (_\blacktriangle E), T \to {}_\blacktriangle a\} \\
&= S4
\end{aligned}
$$

The closures of the other reachable sets of items are themselves.

- $S5 = \{E \to E - T_\blacktriangle\}$
- $S6 = \{S \to E\$_\blacktriangle\}$
- $S7 = \{E \to T_\blacktriangle\}$
- $S8 = \{T \to a_\blacktriangle\}$
- $S9 = \{T \to (E_\blacktriangle), E \to E_\blacktriangle - T\}$

Example: DFA

Parsing Table



Example: Parsing Table

DFA

States	Input					Nonterminals		
	a	(	)	\$	$-$	S	E	T
S1	S8	S2					G3	G7
S2	S8	S2					G9	G7
S3				ACC	S4			
S4	S8	S2						G5
S5	R2	R2	R2	R2	R2			
S6	R1	R1	R1	R1	R1			
S7	R3	R3	R3	R3	R3			
S8	R4	R4	R4	R4	R4			
S9			S10		S4			
S10	R5	R5	R5	R5	R5			

Note: All empty entries denote errors

Example 4.16 *Consider the following simple input viz. a\$. Here are the parsing steps.*

DFA *Parsing Table*

Stack	Input	Action
S1	a\$	Shift S8
S1 a S8	\$	Reduce Rule 4
S1 T	\$	Goto S7
S1 T S7	\$	Reduce Rule 3
S1 E	\$	Goto S3
S1 E S3	\$	Accept

Example 4.17 *Here is a slightly more complex input* $a-(a-a)\$$.

DFA *Parsing Table*

S1	$a-(a-a)\$$	Shift S8
S1 a S8	$-(a-a)\$$	Reduce Rule 4
S1 T	$-(a-a)\$$	Go to S7
S1 T S7	$-(a-a)\$$	Reduce Rule 3
S1 E	$-(a-a)\$$	Go to S3
S1 E S3	$-(a-a)\$$	Shift S4
S1 E S3 $-$ S4	$(a-a)\$$	Shift S2
S1 E S3 $-$ S4 (S2	$a-a)\$$	Shift S8
S1 E S3 $-$ S4 (S2 a S8	$-a)\$$	Reduce Rule 4
S1 E S3 $-$ S4 (S2 T	$-a)\$$	Go to S7
S1 E S3 $-$ S4 (S2 T S7	$-a)\$$	Reduce Rule 3
S1 E S3 $-$ S4 (S2 E	$-a)\$$	Go to S9

DFA Parsing Table

DFA Parsing Table

Stack	Input	Action
S1 E S3 $-$ S4 (S2 E S9	$-a)\$$	Shift S4
S1 E S3 $-$ S4 (S2 E S9 $-$ S4	$a)\$$	Shift S8
S1 E S3 $-$ S4 (S2 E S9 $-$ S4 a S8	$)\$$	Reduce Rule 4
S1 E S3 $-$ S4 (S2 E S9 $-$ S4 T	$)\$$	Go to S5
S1 E S3 $-$ S4 (S2 E S9 $-$ S4 T S5	$)\$$	Reduce Rule 2
S1 E S3 $-$ S4 (S2 E	$)\$$	Go to S9
S1 E S3 $-$ S4 (S2 E S9	$)\$$	Shift S10
S1 E S3 $-$ S4 (S2 E S9) S10	$\$$	Reduce Rule 5
S1 E S3 $-$ S4 T	$\$$	Go to S5
S1 E S3 $-$ S4 T S5	$\$$	Reduce Rule 2
S1 E	$\$$	Go to S3
S1 E S3	$\$$	Accept

Exercise 4.3

1. *Design a LR(0) parser for the grammar of palindromes. Identify whether there are any conflicts in the parsing table.*
2. *Design a LR(0) parser for the grammar of Matching brackets and identify any conflicts.*
3. *Design a context-free grammar for a language on the terminal symbols* `a` *and* `b` *such that every string has more* `a`*s than* `b`*s. Design a LR(0) parser for this grammar and find all the conflicts, if any.*
4. *Since every regular expression may also be represented by a context-free grammar design an LR(0) parser for comments in C.*

CFG = RLG + Bracket Matching

We use the idea that a context-free grammar is essentially a regular grammar with parentheses matching to arbitrary depths. Hence a DFA with some reductions introduced may work.

We modify the grammar to have a special terminal symbol called the end-marker (denoted by $\$$). Now consider the following simple grammar with a single right-associative binary operator ^ and bracket-matching.

We create a DFA of "items" which also have a special marker called the "cursor" ($\blacktriangle$).

LR(0) with Right-Association

Consider the following grammar

$$
\begin{array}{lrcl}
1. & S & \to & E\$ \\
2. & E & \to & P \,\hat{}\, E \\
3. & E & \to & P \\
4. & P & \to & a \\
5. & P & \to & (E)
\end{array}
$$

The following items make up the initial state S1 of the DFA

$$
\begin{array}{lrcl}
I1.1 & S & \to & {}_\blacktriangle E\$ \\
I2.1 & E & \to & {}_\blacktriangle P \,\hat{}\, E \\
I3.1 & E & \to & {}_\blacktriangle P \\
I4.1 & P & \to & {}_\blacktriangle a \\
I5.1 & P & \to & {}_\blacktriangle (E)
\end{array}
$$

Shift-Reduce Conflicts in LR(0)

There is a transition on the nonterminal P to the state S2 which is made up of the following items.

$$I2.2\;\; E \rightarrow P_{\blacktriangle}\,\hat{}\,E$$
$$I3.2\;\; E \rightarrow P_{\blacktriangle}$$

Then clearly the LR(0) parser suffers a shift-reduce conflict because

- item I2.2 indicates a shift action,
- item I3.2 produces a reduce action

This in contrast to the parsing table produced earlier where reduce actions took place regardless of the input symbol. Clearly now that principle will have to be modified.

The parsing table in this case would have a shift action if the input in state S2 is a ^ and a reduce action for all other input symbols.

FOLLOW Sets

We construct for each non-terminal symbol a set of terminal symbols that can *follow* this non-terminal in any rightmost derivation. In the previous grammar we have

$$follow(E) = \{\$,)\}$$
$$follow(P) = \{\hat{}\}$$

Depending upon the input symbol and whether it appears in the FOLLOW set of the non-terminal under question we resolve the shift-reduce conflict. This modification to LR(0) is called Simple LR (SLR) parsing method. However SLR is not powerful enough for many useful grammar constructions that are encountered in many programming languages.

Computing FIRST Sets

In order to compute FOLLOW sets we require FIRST sets of sentential forms to be constructed too.

1. first $(a) = \{a\}$ for every terminal symbol a.
2. $\varepsilon \in first(X)$ if $X \to \varepsilon \in P$.
3. If $X \to Y_1Y_2 \cdots Y_k \in P$ then $first(Y_1) \subseteq first(X)$
4. If $X \to Y_1Y_2 \cdots Y_k \in P$ then for each $i : i < k$ such that $Y_1Y_2 \cdots Y_i \Rightarrow \varepsilon$, $first(Y_{i+1}) \subseteq first(X)$.

Computing FOLLOW Sets

Once FIRST has been computed, computing FOLLOW for each non-terminal symbol is quite easy.

1. $\$ \in follow(S)$ where S is the start symbol of the augmented[a] grammar.
2. For each production rule of the form $A \rightarrow \alpha B \beta$, $first(\beta) - \{\varepsilon\} \subseteq follow(B)$.
3. For each production rule of the form $A \rightarrow \alpha B \beta$, if $\varepsilon \in first(\beta)$ then $follow(A) \subseteq follow(B)$.
4. For each production of the form $A \rightarrow \alpha B$, $follow(A) \subseteq follow(B)$.

[a] In an augmented grammar, the start symbol does not occur on the right hand side of any production

if-then-else vs. if-then

Most programming languages have two separate constructs `if-then` and `if-then-else`. We abbreviate the keywords and use the following symbols

Tokens	Symbols
`if`	`i`
`then`	`t`
`else`	`e`
booleans	b
other expressions	a

if-then-else vs. if-then (Contd.)

and construct the following two augmented grammars G_1 and G_2.

$1_1.\ S \rightarrow I\ \$$	$1_2.\ S \rightarrow I\ \$$
$2_1.\ I \rightarrow U$	$2_2.\ I \rightarrow \mathtt{i}\ b\ \mathtt{t}\ I\ E$
$3_1.\ I \rightarrow M$	$3_2.\ I \rightarrow a$
$4_1.\ U \rightarrow \mathtt{i}\ b\ \mathtt{t}\ I$	$4_2.\ E \rightarrow \mathtt{e}\ I$
$5_1.\ U \rightarrow \mathtt{i}\ b\ \mathtt{t}\ M\ \mathtt{e}\ U$	$5_2.\ E \rightarrow \varepsilon$
$6_1.\ M \rightarrow \mathtt{i}\ b\ \mathtt{t}\ M\ \mathtt{e}\ M$	
$7_1.\ M \rightarrow a$	

Exercise 4.4

1. *Prove that grammar G_2 is ambiguous.*
2. *Construct the LR(0) parsing tables for both G_1 and G_2 and find all shift-reduce conflicts in the parsing table.*
3. *Construct the FOLLOW sets in each case and try to resolve the conflicts.*
4. *Show that the following augmented grammar cannot be parsed (i.e. there are conflicts that cannot be resolved by FOLLOW sets) either by LR(0) or SLR parsers. (*Hint *First construct the LR(0) DFA).*

$$\begin{array}{llll} 1. & S & \rightarrow & E\$ \\ 2. & E & \rightarrow & L = R \\ 3. & E & \rightarrow & R \\ 4. & L & \rightarrow & *\,R \\ 5. & L & \rightarrow & a \\ 6. & R & \rightarrow & L \end{array}$$

Bindings, Attributes & Semantic Analysis

Context-sensitive Grammars

Definition 5.1 $G = \langle N, T, P, S \rangle$ *is called a* **context-sensitive grammar(CSG)** *if each production is of the form* $\alpha X \beta \longrightarrow \alpha\gamma\beta$, *where*

- $X \in N$ *is a nonterminal and*
- $\alpha, \beta, \gamma \in (N \cup T)^*$ *are sentential forms.*
- *The production is* **terminal** *if* $\alpha\gamma\beta$ *is a sentence*

Note:

- α and β are the context within which the non-terminal X can generate the sentential form γ.
- Every CFG is also a CSG with rules having empty contexts.
- The parsing problem for CSGs is known to be PSPACE-complete.

A Context-sensitive Language

The language $\{a^n b^n c^n | n > 0\}$ is not context-free but can be generated by the *context-sensitive grammar* $G = \langle N, \{a, b, c\}, P, S\rangle$ whose productions are

$$
\begin{array}{llllll}
S & \longrightarrow & aBC \mid aSBC & & & \\
aB & \longrightarrow & ab & bB & \longrightarrow & bb \\
bC & \longrightarrow & bc & cC & \longrightarrow & cc \\
CB & \longrightarrow & CZ & CZ & \longrightarrow & WZ \\
WZ & \longrightarrow & WC & WC & \longrightarrow & BC
\end{array}
$$

Context-sensitive Analysis: Preamble

The Big Picture

- Every programming language can be used to program any computable function, assuming of course, it has
  - unbounded memory, and
  - unbounded time
- Context-free grammars are used to specify the phrase structure of a language in a manner that is *free of all context*.

Semantic Analysis

1. Context-free grammars are not powerful enough to represent all computable functions.

   Example 5.2 *The language* $\{a^n b^n c^n | n > 0\}$ *is not context-free but can be generated by a context-sensitive grammar.*

2. Semantic analysis is an essential step to

   - producing the abstract syntax trees (AST)
   - generating IR-code, since it requires the computation of certain *bits and pieces of information* called attributes (which include information to be entered into the symbol table or useful for error-handling)
   - allocating memory for individual "objects" (variables, constants, structures, arrays etc.)

The Big Picture

The parser for a context-free grammar transforms the token stream into a **derivation tree** (which we also call a **concrete parse tree**)[3]. What we actually require in order to perform a computation is really an abstract syntax tree.

Example 5.3 *Consider the two sentences $a - a/b$ and $a - (a/b)$ which are both valid sentences generated by the grammar of our favourite example.*

The (possibly modified grammar) required for parsing

- treats all tokens uniformly since the phrase structure of the grammar is all-important during the parsing process,
- introduces bracketing and punctuation marks for
  - disambiguation and to override associativity when needed,
  - to facilitate easy parsing

[3]The term *parse tree* is a much abused term used to refer to anything from a derivation tree to an abstract syntax tree (AST).

But these symbols do not by themselves carry any semantic[4] information.

- also has many more non-terminal symbols that are required for parsing, but which carry no semantic significance
  - *either* for the end-user of the language
  - *or* for the later phases of the compilation process.

Both expressions in example 5.3 have the same meaning (semantics) if we assume that the operations are subtraction and division over integers respectively, and that division has higher precedence than subtraction. But the sentences are *syntactically* different and correspondingly have different parse trees (see fig. 5). Both the expressions may be represented by the following **abstract syntax tree (AST)** which gives the hierarchical structure of the expression.

Notice that in figure 6

- Every node in the AST is labelled by a token.
- The AST abstracts away from non-terminals which have significance only for the parsing of the expression and have no semantic significance whatsoever,
- The AST abstracts away from bracketing and punctuation mechanisms and provides a hierarchical structure containing only the *essential* operators and operands.
- The AST clearly distinguishes the operators (based on their arity) from the operands (which are leaves of the AST).

[4] *Semantic analysis* is another much abused term, often used by compiler writers to included even merely *context-sensitive* information.

Context-sensitive Analysis: 1

- There are aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - scope and visibility issues with respect to identifiers in a program.
  - type consistency between declaration and use.
  - correspondence between formal and actual parameters (example 5.2 is an abstraction where a^n represents a function or procedure declaration with n formal parameters and b^n and c^n represent two calls to the same procedure in which the number of actual parameters should equal n).
- Many of these attributes are *context-sensitive* in nature. They need to be computed and if necessary propagated during parsing from wherever they are available.

Context-sensitive Analysis: 2

- The parser of a programming language provides the *framework* within which the IR-code or even the target code is to be generated.
- The parser also provides a *structuring* mechanism that divides the task of code generation into bits and pieces determined by the individual nonterminals and production rules.
- The parser provides the *framework* from within which the semantic analysis (which includes the bits and pieces of information that are required for code generation) is performed

Binding

Binding is a central and fundamental concept in the definition of programming languages and their semantics. Programs written in a programming language deal with various entities – variables, subprograms, expressions, declarations, commands, modules, objects, object classes etc. These entities carry with them certain properties or values called *attributes*.

Definition 5.4 *The* **binding** *of a program entity to an attribute is simply the choice of the attribute from a set of possible attributes.*

Programming Language Entities

Programming languages vary widely in the various entities that they can deal with, in the number of attributes that are bound to each entity, the times at which these bindings take place (*binding time*) and the stability of these bindings (whether the bindings are fixed or modifiable).

Example 5.5

- *A* **variable** *has various attributes such as its name, its type, a storage area where its value is stored, a size depending on its type etc.*
  - *A variable in an imperative language also has a binding to its location and each of these locations has a binding to its value.*
  - *A variable in a pure functional setting needs to be bound to its value.*
  - *In addition a variable may be a formal parameter of some other entity such as a procedure or function and has certain parameter-passing conventions associated with it.*
- *A* **procedure** *or* **function** *has a name. formal parameters of certain types, return parameters of certain types and parameter-passing conventions associated with each formal parameter etc.*
- *A* **command** *has certain associated actions determined by its semantics.*

The values of the attributes of each entity need to be set before it may be used. Setting the values of these attributes is called **binding**. For each entity the attribute information is contained in a repository called a *descriptor*.

Binding times

The binding of attributes to entities may occur at various times during compilation, linking, loading and execution. Broadly speaking, *early binding* (also known as *static binding* and is performed before execution begins) ensures

- early detection and reporting of errors, rather than delaying them to execution time,
- greater run-time efficiency since the overheads of creating associations have been already dealt with statically (i.e. before actual execution). Hence *compiled* code tends to run faster than *interpreted* code.

On the other hand, *late binding*

- allows for greater flexibility. In particular interpreters often perform late bindings and therefore allow flexible (and interactive) code development. However,
- most programming languages that are not statically typed, tend to point out errors during run-time only if a given operation is not possible. Inadvertent type errors introduced in the program may often lead to unexpected results because of the lack of type-checking.

  Example 5.6 *Many implementations of Scheme and LiSP return an empty list when the tail of an empty list is required, and they tend to return a null when accessing the head of an empty list.*

Static or Early binding. These bindings occur before run-time. The term "static" refers to both binding that occurs before exection and also to the stability of the binding (i.e. it is not modifiable once the binding is done).

Language definition time. In most languages the control-flow constructs, the set of primitive types and the constructors available for creating complex types are chosen at language definition time.

Example 5.7 *The type "integer" is bound at the language definition time and refers (as closely as possible*[5]*) to its mathematical counterpart namely the algebra of integers and their associated operations and relations.*

Language implementation time. Most language manuals leave a variety of issues to the discretion of the language implementor.

Example 5.8 *The "integer" type refers to a* finite *set of values bound to a certain memory representation – e.g. byte, full-word, double-word etc. This automatically constrains the set of values that can be termed "integer" in the programming language.*

Compile time or translation time binding. Compilers choose the mapping of high-level constructs to machine code or IR code, including layout of statically defined data structures.

Example 5.9 *Often at compile time only relocatable addresses (i.e. addresses specified as an offset from a possibly unknown physical address in memory) are specified.*

- *Even for variables which have been statically declared and have a known fixed size, the actual physical address is usually available only at load time.*
- *In languages that support* recursion *there may be several activations of the same subprogram present simultaneously and hence many incarnations of the same variable are present simultaneously and each of them needs to be bound*

[5] The set of integers is actually infinite, however the set of integers representable on a machine with a finite word-length is likely to be finite. This also affects the operations on integers and their behaviour – e.g. overflow

to separate physical memory address. The actual binding of the each incarnation of the variable to its memory location may get fixed only at run-time depending the activation involved.

Program writing time. Programmers of course choose names, algorithms and data structures and describe certain high-level bindings (between names and data structures for example) at program writing time.

Example 5.10 *In some languages which distinguish between reserved words and mere keywords (e.g. Pascal) the type "integer" may be redefined in a user program and a different representation may be defined for it.*

Link time. Most modern compilers support *separate compilation* – compiling different modules of a program at different times. They depend on the availability of a library of standard routines. Program compilation is not complete until various names occurring in the program which depend upon certain modules are appropriately bound by the linker. The linker chooses the overall layout of the various modules with respect to each other and resolves *inter-module references* and references to the names within modules which may be exported to the program.

Load time. Load time refers to the point at which the operating system loads the program into memory so that it may be run. Most modern operating systems distinguish between virtual and physical addresses. Virtual addresses are chosen at link time. The binding of virtual addresses to physical addresses takes place at load time.

Dynamic or Late or runtime binding. Many bindings are performed during execution. These are usually modifiable at run-time (unlike static bindings).

Imperative variables. In most imperative languages variables are bound to a value at run-time and may be repeatedly modified.

Functional variables. In most pure functional languages, the value bound to a variable cannot be modified once a binding is established, *even though the location bound to a variable may be modified at run-time due to garbage collection and compaction.*

Entry into a sub-program or a block. Important classes of bindings take place at the time of entry into the sub-program or block.

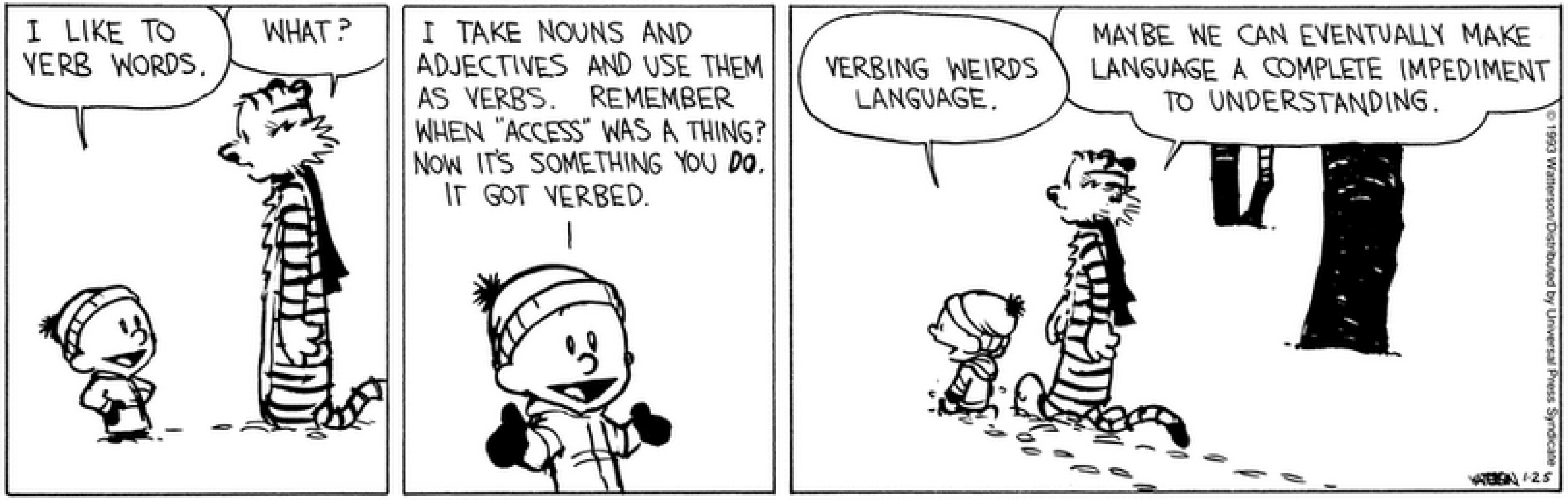
- binding of formal parameters to storage locations and
- binding of formal to actual parameters

At arbitrary points during execution. Some bindings may occur at any point during execution.

- Binding variables to values through assignment
- Binding of names to storage locations may change during garbage collection in languages (e.g. SML, LiSP, Java etc.) which support automatic garbage collection.

Introduction to Semantics

Calvin and Hobbes - Verbing Weirds Language - by Bill Watterson for January 25, 1993



Context-free grammars (actually EBNF) are used to describe the rules that define the grammatical structure of phrases and sentences in the language. However a manual for a programming language also needs to describe the meaning of each construct in the language both alone and in conjunction with other constructs. This is to enable users of the language to write correct programs and to be able to predict the effect of each construct. Implementors of the language need correct definitions of the meanings to be able to construct correct implementations of the language.

Syntax defines a well-formed program. Semantics defines the meaning of a syntactically correct program. However not all well-formed programs have well defined meanings. Thus semantics also separates meaningful programs from merely syntactically correct ones.

"Meaning" in the case of programming languages often refers to the execution behaviour of the program or the individual constructs. This is useful from an implementation point of view. From a user programmer's point of view It is possible to view a programming language as a precise description mechanism that is independent of execution behaviour and restrict meaning to the "effect" that a program or a construct has on some input (state).

While there are precise means of defining the syntax af the language, most language manuals describe the meanings of the constructs in natural language prose. This unfortunately is not very desirable as natural language tends to be too verbose, imprecise and very often ambiguous. On the other hand, if users and implementors have to be on the same page as regards the behaviour of programs and individual programming constructs a precise and unambiguous definition is required for this description. Typically a user programmer may misunderstand what a program or a construct will do when executed. Implementors may interpret the meaning differently and hence different implementaions of the language may yield different results on the same program.

While there are several formalisms for defining meanings of the constructs of a programming language, they all share the following characteristics in order to maintain a certain uniformity and applicability for any program written in the language

- Meanings should be *syntax-directed* i.e. meanings should be based on the syntactical definition of the language in the sense that it follows the hierarchy of the non-terminals in the grammar. The syntax (grammar) of the language therefore provides the framework for the semantics of the language.

- The meaning should be *compositional* i.e. the meaning of a compound construct should be expressed in terms of the meanings of the individual components in the construct. Hence it is important that the meanings of the most basic constructs be defined first so that the meanings of the compound constructs may be expressed in terms of the meanings of the individual components of the compound construct.

Disjoint Scopes

```
let
  val x = 10;
  fun fun1 y =
    let
      ...
    in
      ...
    end
  fun fun2 z =
    let
      ...
    in
      ...
    end
in
  fun1 (fun2 x)
end
```

Nested Scopes

```
let
  val x = 10;
  fun fun1  y =
       let
          val x = 15
        in
          x + y
        end
 in
   fun1 x
 end
```

Overlapping Scopes

```
let
  val x = 10;
  fun fun1  y =
                    . . .

                    . . .

                    . . .
in
                    . . .

  fun1 (fun2 x)
end
```

Spannning

```
let
  val x = 10;
  fun fun1 y =
        ...
        ...
  fun fun2 z =
        ...
        ...
in
  fun1 (fun2 x)
end
```

Scope & Names

- A name may occur either as being defined or as a use of a previously defined name
- The same name may be used to refer to different objects.
- The use of a name refers to the textually most recent definition in the innermost enclosing scope

diagram

Names & References: 0

```
let
   val x = 10; val z = 5;
   fun fun1  y =
            let
               val x = 15
            in
               x + y * z
            end
in
   fun1 x
end
```

Back to Scope & Names

Names & References: 1

```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 2

```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
        val x = 15
      in
        x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 3

```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
         val x = 15
      in
         x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 4

```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 5

```
let
  val x = 10; val z = 5;
  fun fun1 y =
    let
      val x = 15
    in
      x + y * z
    end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 6

```
let
  val x = 10; val z = 5;
  fun fun1 y =
    let
      val x = 15
    in
      x + y * z
    end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 7

```
let
  val x = 10; val x = x - 5;
  fun fun1  y =
                let
                      ...
                 in
                      ...
                end
  fun fun2  z =
                let
                      ...
                 in
                      ...
                end
in fun1 (fun2 x)
end
```

Back to Scope & Names

Names & References: 8

```
let
  val x = 10; val x = x - 5;
  fun fun1 y =
      let
        ...
      in
        ...
      end

  fun fun2 z =
      let
        ...
      in
        ...
      end
in fun1 (fun2 x)
end
```

Back to Scope & Names

Names & References: 9

```
let
  val x = 10; val x = x - 5;
  fun fun1 y =
      let
         ...
      in
         ...
      end
  fun fun2 z =
      let
         ...
      in
         ...
      end
in fun1 (fun2 x)
end
```

Back to Scope & Names

Definition of Names

Definitions are of the form

$qualifier\ \underline{name}\ \ldots = body$

- `val` $\underline{name}$ `=`
- `fun` $\underline{name}$ ($\underline{argnames}$) $=$
- `local` $definitions$
  `in` $definition$
  `end`

Use of Names

Names are used in expressions.
Expressions may occur

- by themselves – to be evaluated
- as the *body* of a definition
- as the *body* of a let-expression

```
let definitions
in  expression
end
```

use of local

Scope & local

```
local
          fun fun1     y =
                            ...

          fun fun2     z = ...
                              fun1

  in
          fun fun3   x   =    ...
                            fun2 ...
                              fun1 ...
  end
```

Symbol Table

"The name of the song is called 'Haddock's Eyes'."

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. *The name of the song really is, 'The Aged Aged Man'."*

Then I ought to have said 'That's what the song is called'?" Alice corrected herself.

"No you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, *you know!"*

"Well, what is *the song, then?" said Alice, who was by this time completely bewildered.*

"I was coming to that", the Knight said. "The song really is 'A-Sitting On a Gate': and the tune's my own invention.

Lewis Carroll, *Through the Looking-Glass*

Symbol Table:1

The Big picture

- The store house of context-sensitive and run-time information about every identifier in the source program.
- All accesses relating to an identifier require to first find the attributes of the identifier from the symbol table
- Usually organized as a hash table[a] – provides fast access.
- Compiler-generated temporaries may also be stored in the symbol table

[a]Sometimes other data-structures such as red-black trees are also used.

Symbol Table:2

The Big picture

Attributes stored in a symbol table for each identifier:

- type
- size
- scope/visibility information
- base address
- addresses to location of auxiliary symbol tables (in case of records, procedures, classes)
- address of the location containing the string which actually names the identifier and its length in the string pool

Symbol Table:3

- A symbol table exists through out the compilation (and run-time for debugging purposes).
- Major operations required of a symbol table:
  - insertion
  - search
  - deletions are purely logical (depending on scope and visibility) and not physical
- Keywords are often stored in the symbol table before the compilation process begins.

Symbol Table:4

The Big picture

Accesses to the symbol table at every stage of the compilation process,

Scanning: Insertion of new identifiers.

Parsing: Access to the symbol table to ensure that an operand exists (declaration before use).

Semantic analysis:

- Determination of types of identifiers from declarations
- type checking to ensure that operands are used in type-valid contexts.
- Checking scope, visibility violations.

Symbol Table:5

The Big picture

IR generation: . Memory allocation and relative[a] address calculation.

Optimization: All memory accesses through symbol table

Target code: Translation of relative addresses to absolute addresses in terms of word length, word boundary etc.

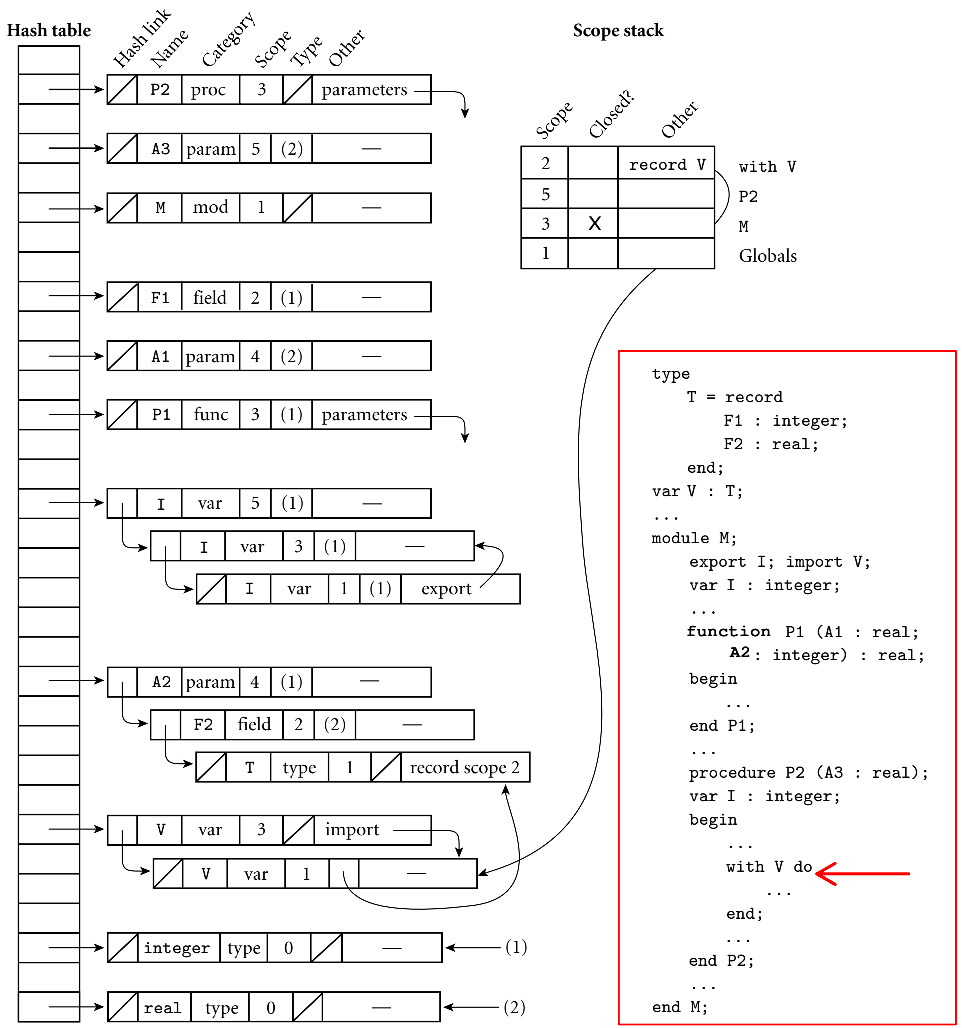
[a] i.e.relative to a base address that is known only at run-time

The hash table

Each name is hashed to an index of the hash table whose entry points to a *chain* of records where each record contains

- a possible link to the next record on the chain (in case of collisions)
- the name of the identifier
- category (e.g. module, procedure, function, block, record, formal parameter etc.)
- scope number of the identifier
- type information
- number of parameters (in case of functions, procedures, modules classes etc.)
- visibility information (derived from qualifiers such as **public**, **private**)

Corrected version of page from Michael Scott: Programming Language Pragmatics.




```
type
    T = record
        F1 : integer;
        F2 : real;
    end;
var V : T;
...
module M;
    export I; import V;
    var I : integer;
    ...
    function P1 (A1 : real;
        A2 : integer) : real;
    begin
        ...
    end P1;
    ...
    procedure P2 (A3 : real);
    var I : integer;
    begin
        ...
        with V do
            ...
        end;
        ...
    end P2;
    ...
end M;
```

Figure 3.19 **LeBlanc-Cook symbol table for an example program in a language like Modula-2.** The scope stack represents the referencing environment of the `with` statement in procedure `P2`. For the sake of clarity, the many pointers from type fields to the symbol table entries for `integer` and `real` are shown as parenthesized (1)s and (2)s, rather than as arrows.

Symbol Table: Scope Stack

Scope rules

- In addition to the hash table a scope stack is maintained for resolving non-local references.
- The new scope number is *push*ed onto the scope stack when the compiler enters a new scope and *pop*ped when exiting a scope.
- There could be unnamed scopes too (e.g. unnamed blocks with local declarations, **for**-loops where the counting variable is local to the loop etc).
- Each (static) scope may be assigned a number in sequential order as it is encountered in the program *text* starting with 0 assigned for the global scope.
- The scope number of a nested scope is always greater than that of its parent.

8. Runtime Structure

Run-time Structure

It is ever so. One of the poets, whose name I cannot recall, has a passage, which I am unable at the moment to remember, in one of his works, which for the time being has slipped my mind, which hits off admirably this age-old situation.

P. G. Wodehouse, *The Long Hole* in *The Golf Omnibus*

Run-time Environment

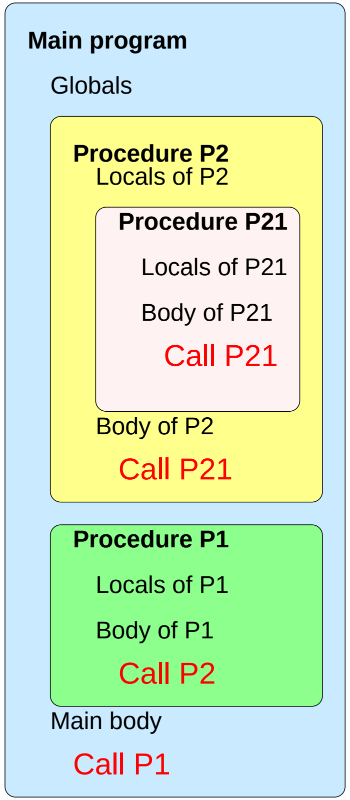
Memory for running a program is divided up as follows

Code Segment. This is where the *object* code of the program resides

Run-time Stack. Required in a *dynamic* memory management technique. Especially required in languages which support recursion. All data whose sizes can be determined *statically* before loading is stored in an appropriate stack-frame (activation record).

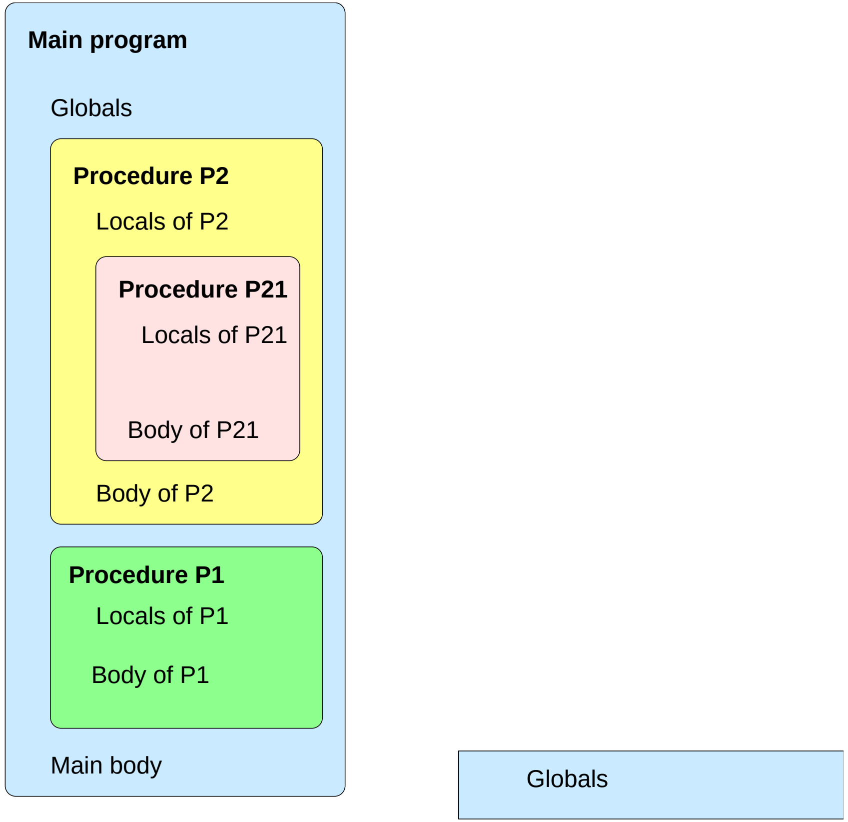
Heap. All data whose sizes are not determined statically and all data that is generated at run-time is stored in the heap.

A Calling Chain



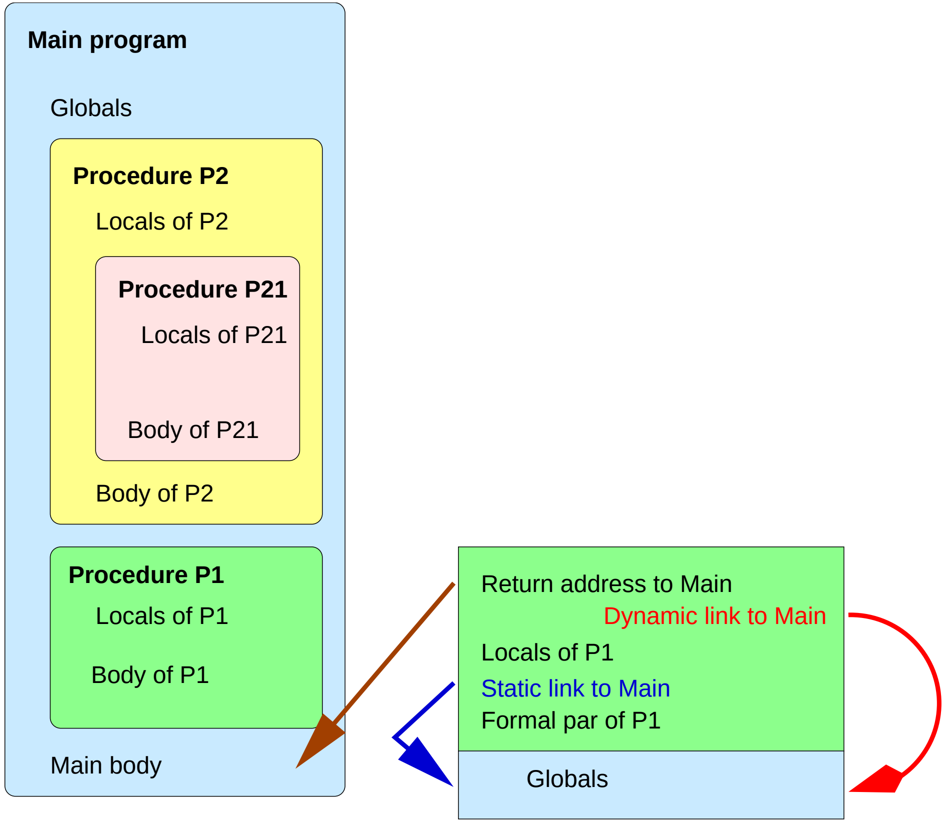
$$\text{Main} \rightarrow \text{P1} \rightarrow \text{P2} \rightarrow \text{P21} \rightarrow \text{P21}$$

Run-time Structure: 1



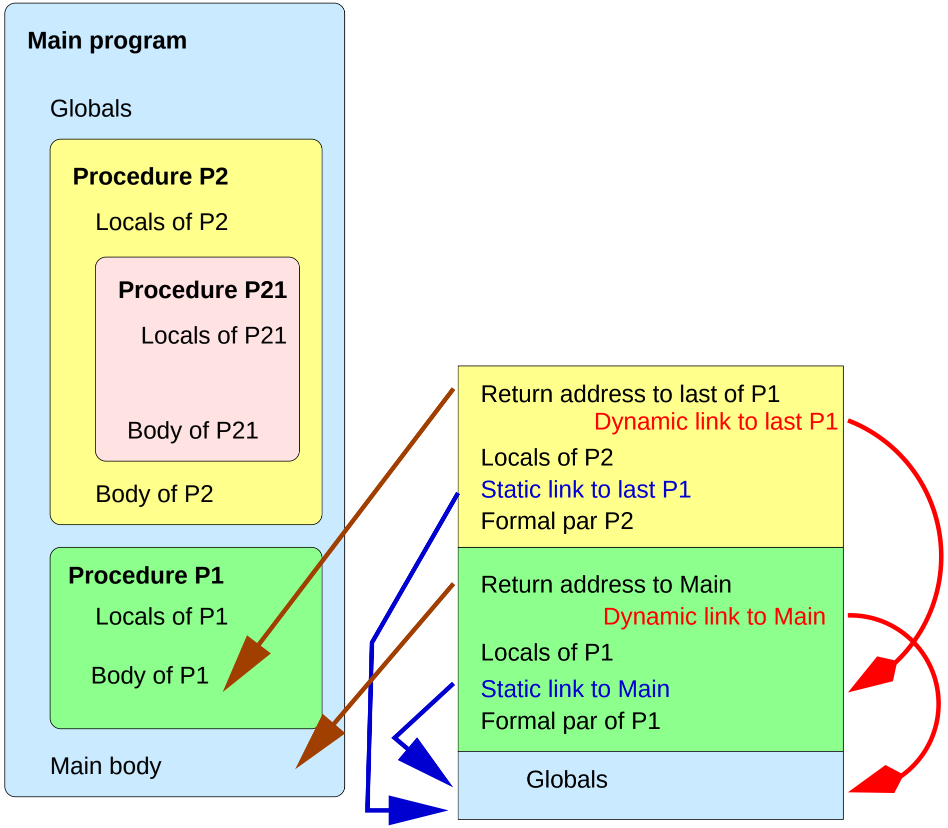
Main

Run-time Structure: 2



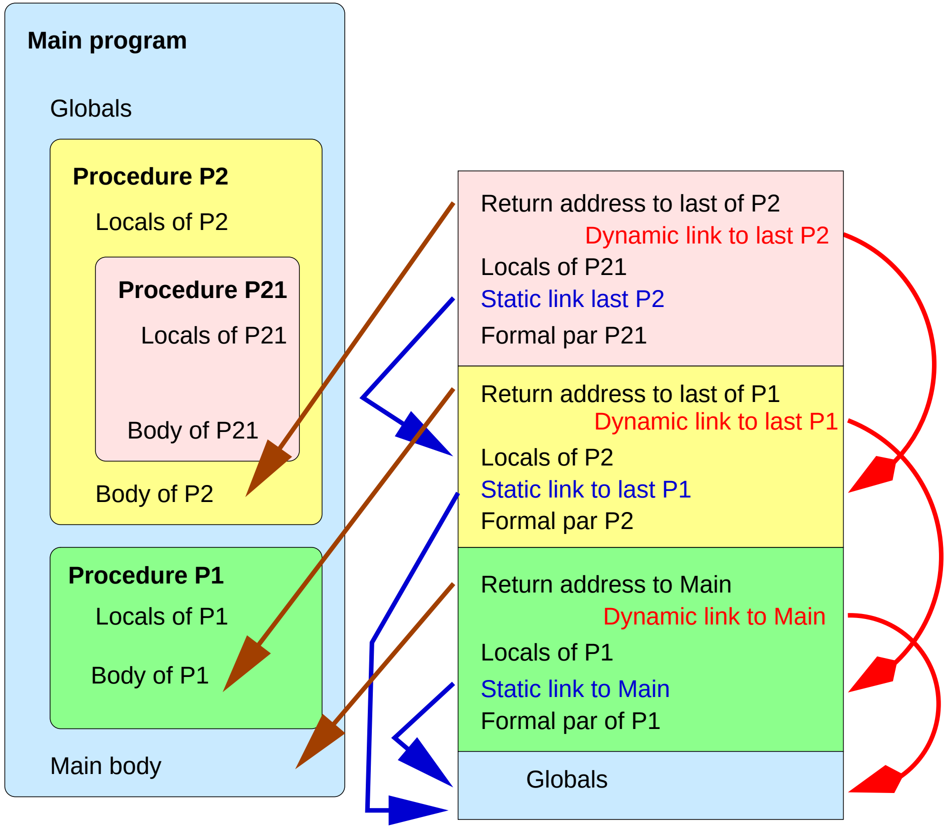
Main $\rightarrow$ P1

Run-time Structure: 3



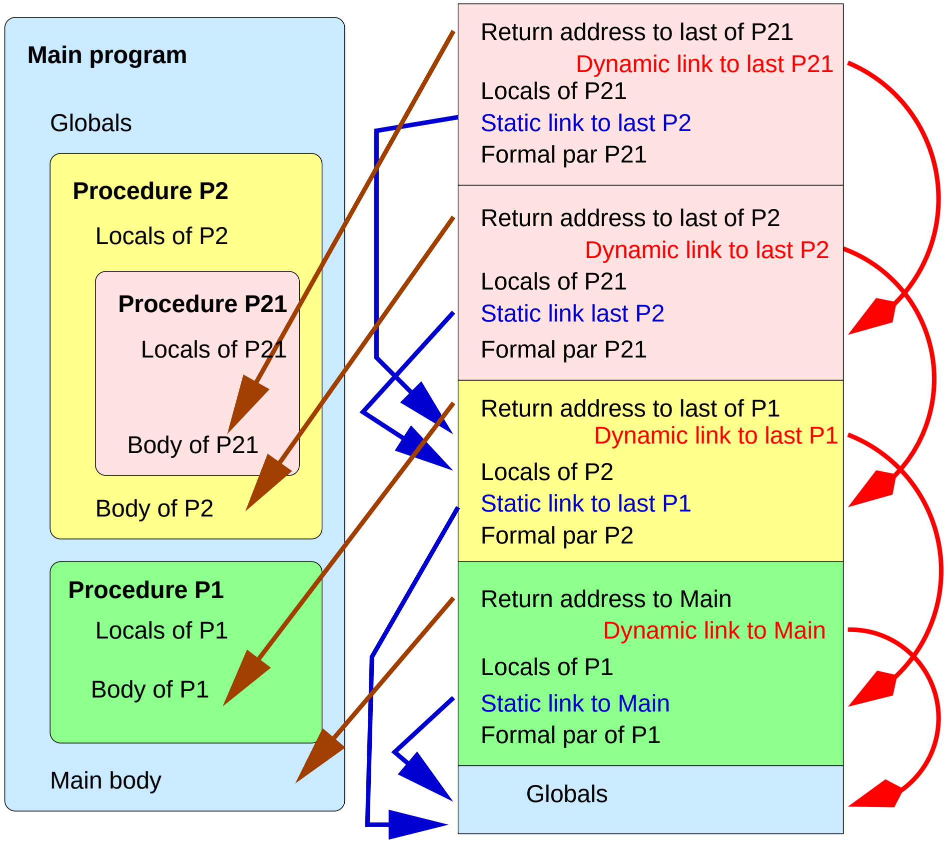
Main $\rightarrow$ P1 $\rightarrow$ P2

Run-time Structure: 4



$$\text{Main} \to \text{P1} \to \text{P2} \to \text{P21}$$

Run-time Structure: 5



Main $\rightarrow$ P1 $\rightarrow$ P2 $\rightarrow$ P21 $\rightarrow$ P21

Back to the Big Picture

Abstract Syntax Trees

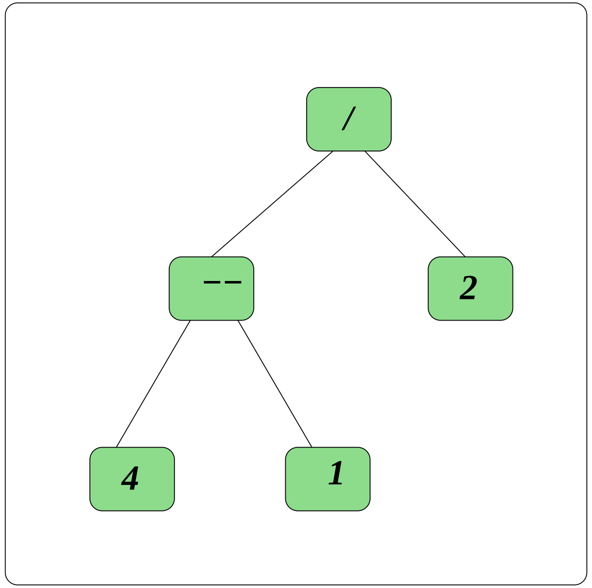
The construction of ASTs from concrete parse trees is an example of a transformation that can be performed using a syntax-directed definition that has no side-effects.
Hence we define it using an attribute grammar.

Definition 9.1 *An attribute grammar is a formal way to define semantic rules and context-sensitive aspects of the language. Each production of the grammar is associated with a set of values or semantic rules. These values and semantic rules are collectively referred to as attributes.*

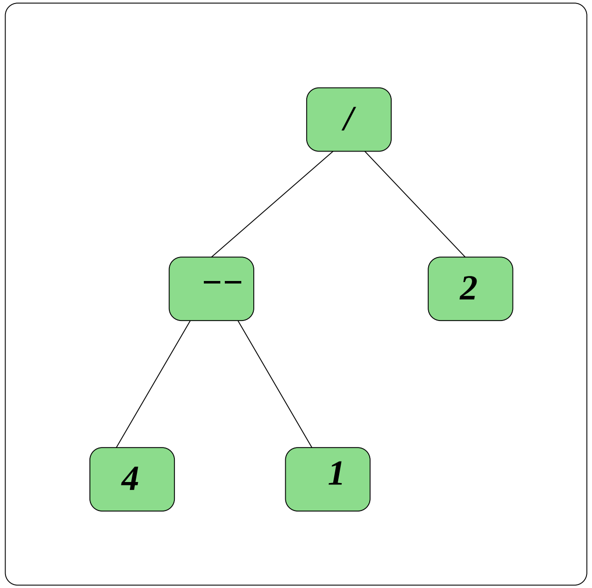
Abstract Syntax: 0

$$
\begin{aligned}
E &\rightarrow E - T \mid T \\
T &\rightarrow T / F \mid F \\
F &\rightarrow \mathbf{n} \mid (E)
\end{aligned}
$$

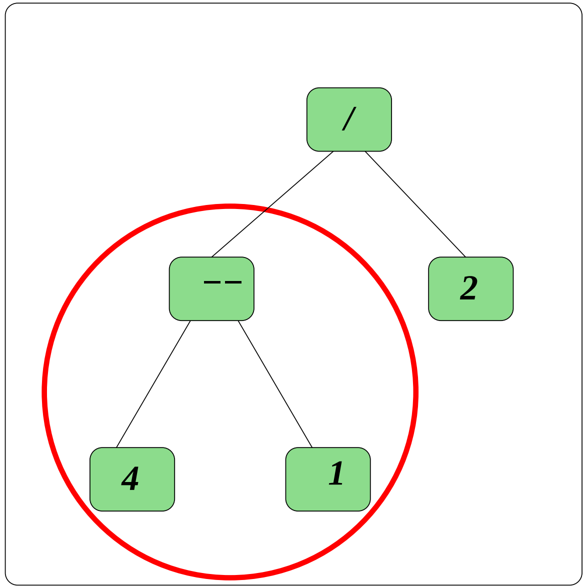
Suppose we want to evaluate an expression $(\mathbf{4} - \mathbf{1})/\mathbf{2}$. What we *actually want* is a tree that looks like this:



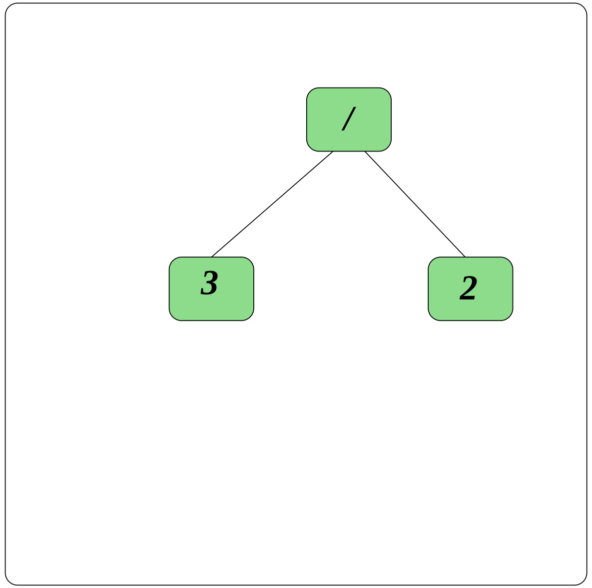
Evaluation: 0



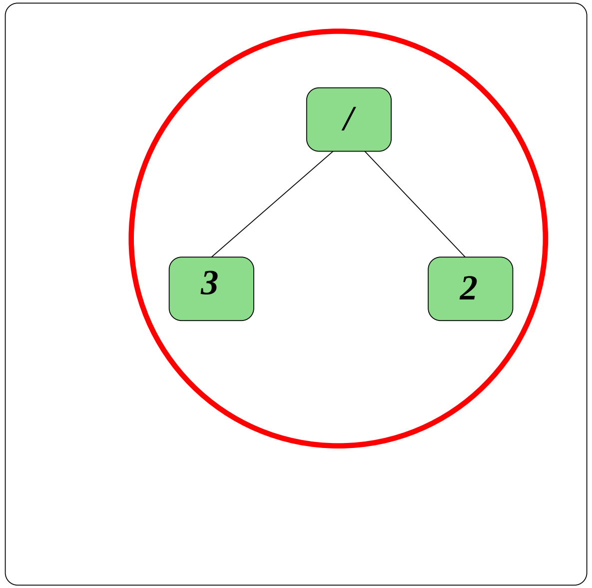
Evaluation: 1



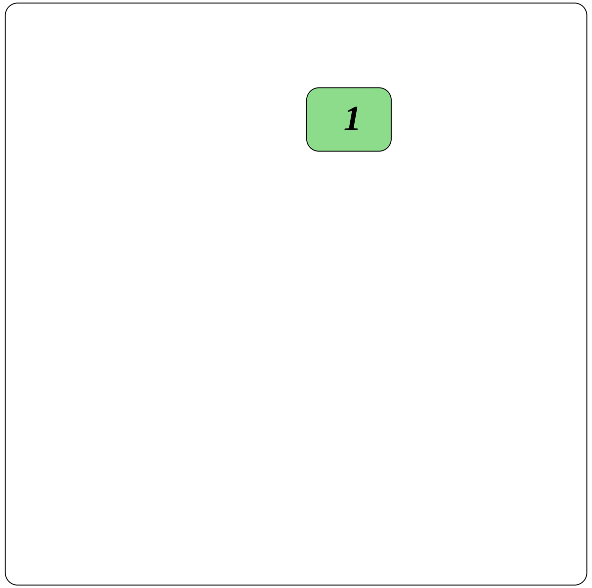
Evaluation: 2



Evaluation: 3



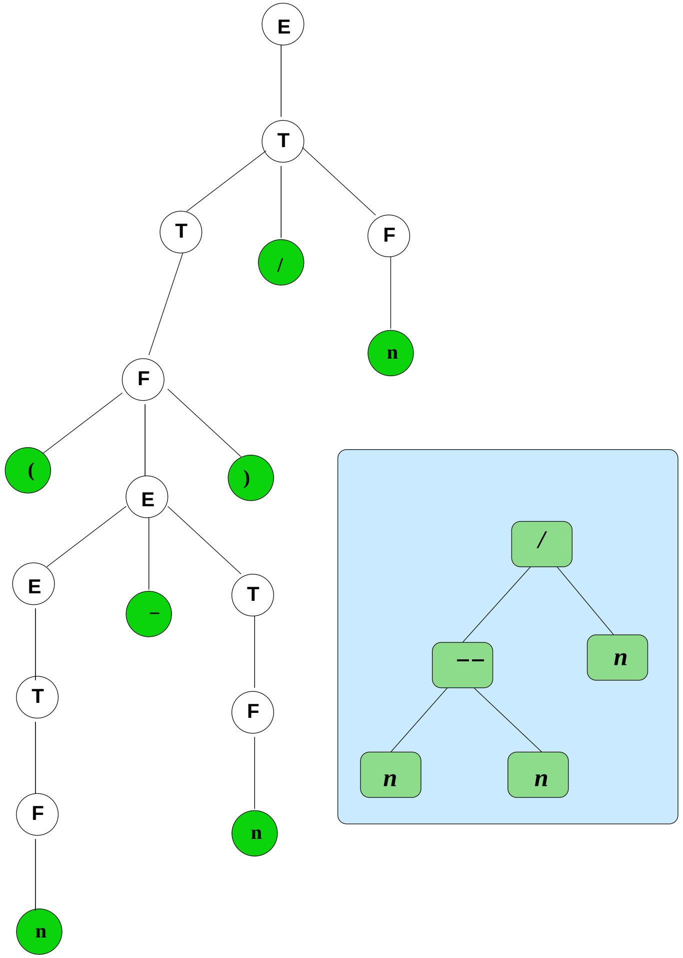
Evaluation: 4



But what we *actually get* during parsing is a tree that looks like ...

Abstract Syntax: 1

... *THIS!*



Abstract Syntax

Shift-reduce parsing produces a concrete syntax tree from the rightmost derivation. The syntax tree is concrete in the sense that

- It contains a lot of redundant symbols that are important or useful only during the parsing stage.
  - punctuation marks
  - brackets of various kinds
- It makes no distinction between operators, operands, and punctuation symbols

On the other hand the abstract syntax tree (AST) contains no punctuations and makes a clear distinction between an operand and an operator.

Abstract Syntax: Imperative Approach

We use attribute grammar rules to construct the abstract syntax tree (AST) from the parse tree.

But in order to do that we first require two procedures for tree construction.

makeLeaf(literal) : Creates a node with label **literal** and returns a pointer or a reference to it.

makeBinaryNode(opr, opd1, opd2) : Creates a node with label **opr** (with fields which point to opd1 and opd2) and returns a pointer or a reference to the newly created node.

Now we may associate a synthesized attribute called ptr with each terminal and nonterminal symbol which points to the root of the subtree created for it.

Abstract Syntax Trees: Imperative

$$
\begin{array}{llll}
E_0 & \to E_1 - T & \rhd & E_0.ptr := makeBinaryNode(-, E_1.ptr, T.ptr) \\
E & \to T & \rhd & E.ptr := T.ptr \\
T_0 & \to T_1 / F & \rhd & T_0.ptr := makeBinaryNode(/, T_1.ptr, F.ptr) \\
T & \to F & \rhd & T.ptr := F.ptr \\
F & \to (E) & \rhd & F.ptr := E.ptr \\
F & \to \mathbf{n} & \rhd & F.ptr := makeLeaf(\mathbf{n}.val)
\end{array}
$$

The Big Picture

Abstract Syntax: Functional Approach

We use attribute grammar rules to construct the abstract syntax tree (AST) functionally from the parse tree.
But in order to do that we first require two functions/constructors for tree construction.

makeLeaf(literal) : Creates a node with label **literal** and returns the AST.

makeBinaryNode(opr, opd1, opd2) : Creates a tree with root label **opr** (with sub-trees opd1 and opd2).

Now we may associate a synthesized attribute called ast with each terminal and nonterminal symbol which points to the root of the subtree created for it.

Abstract Syntax: Functional

$E_0 \rightarrow E_1 - T \quad \triangleright \quad E_0.ast := makeBinaryNode(-, E_1.ast, T.ast)$

$E \rightarrow T \quad \triangleright \quad E.ast := T.ast$

$T_0 \rightarrow T_1 / F \quad \triangleright \quad T_0.ast := makeBinaryNode(/, T_1.ast, F.ast)$

$T \rightarrow F \quad \triangleright \quad T.ast := F.ast$

$F \rightarrow (E) \quad \triangleright \quad F.ast := E.ast$

$F \rightarrow \mathbf{n} \quad \triangleright \quad F.ast := makeLeaf(\mathbf{n}.val)$

The Big Picture

Abstract Syntax: Alternative Functional

In languages like SML which support algebraic (abstract) datatypes, the functions **makeLeaf(literal)** and **makeBinaryNode(opr, opd1, opd2)** may be replaced by the constructors of an appropriate recursively defined datatype AST.

Syntax-directed Translation

Attributes

An attribute can represent anything we choose e.g.

- a string
- a number (e.g. size of an array or the number of formal parameters of a function)
- a type
- a memory location
- a procedure to be executed
- an error message to be displayed

The value of an attribute at a parse-tree node is defined by the semantic rule associated with the production used at that node.

The Structure of a Compiler

Divide and conquer. A large-scale structure and organization of a compiler or translator is defined by the structure of the parser in terms of the individual productions of the context-free grammar that is used in parsing.

Syntax-directed definitions. The problem of context-sensitive and semantic analysis is split up into the computation of individual attributes and semantic rules in such a way that each production is associated with the (partial) computation of one or more attributes.

Glue code. Finally it may require some "glue-code" to put together these computations to obtain the final compiler/translator. The glue-code may also be split into some that occurs in the beginning through global declarations/definitions and some which need to be performed in the end.

Syntax-Directed Definitions (SDD)

Syntax-Directed definitions are high-level specifications which specify the evaluation of

1. various attributes
2. various procedures such as
   - transformations
   - generating code
   - saving information
   - issuing error messages

They hide various implementation details and free the compiler writer from explicitly defining the order in which translation, transformations, and code generation take place.

Kinds of Attributes

There are two kinds of attributes that one can envisage.

Synthesized attributes A synthesized attribute is one whose value depends upon the values of its immediate children in the concrete parse tree.

A syntax-directed definition that uses only synthesized attributes is called an *S-attributed* definition. See example

Inherited attributes An inherited attribute is one whose value depends upon the values of the attributes of its parents or siblings in the parse tree.

Inherited attributes are convenient for expressing the dependence of a language construct on the *context* in which it appears.

What is Syntax-directed?

- A syntax-directed definition is a generalisation of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called synthesized and inherited attributes.
- The various attributes are computed by so-called semantic rules associated with each production of the grammar which allows the computation of the various attributes.
- These semantic rules are in general executed during

  bottom-up (SR) parsing at the stage when a reduction needs to be performed by the given rule and

  top-down (RDP) parsing in the procedure before the next call or return from the procedure. (see subsection 4.9)
- A parse tree showing the various attributes at each node is called an annotated parse tree.

Forms of SDDs

In a syntax-directed definition, each grammar production rule $X \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b = f(a_1, \ldots, a_k)$ where $a_1, \cdots, a_k$ are attributes belonging to X and/or the grammar symbols of α.

Definition 10.1 *Given a production $X \rightarrow \alpha$, an attribute a is*

synthesized: *a synthesized attribute of X (denoted $X.a$) or*

inherited: *an inherited attribute of one of the grammar symbols of α (denoted $B.a$ if a is an attribute of B).*

In each case the attribute a is said to depend upon the attributes $a_1, \cdots, a_k$.

Attribute Grammars

- An attribute grammar is a syntax-directed definition in which the functions in semantic rules can have no side-effects.
- The attribute grammar also specifies how the attributes are propagated through the grammar, by using *graph dependency* between the productions.
- In general *different occurrences* of the *same* non-terminal symbol in each production will be distinguished by appropriate subscripts when defining the semantic rules associated with the rule.

The following example illustrates the concept of a syntax-directed definition using synthesized attributes.

Attribute Grammars: Example

$$E_0 \to E_1 - T \quad \triangleright \quad E_0.val := E_1.val - T.val$$

$$E \to T \quad \triangleright \quad E.val := T.val$$

$$T_0 \to T_1 / F \quad \triangleright \quad T_0.val := T_1.val / F.val$$

$$T \to F \quad \triangleright \quad T.val := F.val$$

$$F \to (E) \quad \triangleright \quad F.val := E.val$$

$$F \to \mathbf{n} \quad \triangleright \quad F.val := \mathbf{n}.val$$

Note: The attribute $\mathbf{n}.val$ is the value of the numeral $\mathbf{n}$ computed during scanning (lexical analysis).

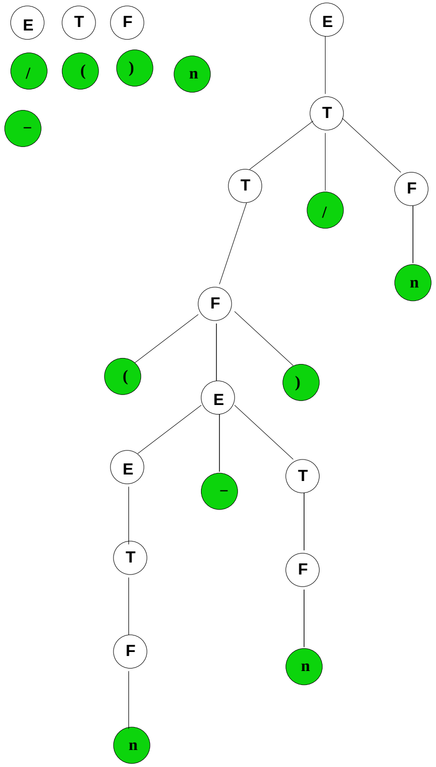
Attributes: Basic Assumptions

- Terminal symbols do not have any children in the concrete parse tree. Attributes of terminals supplied by the lexical analyser during scanning are assumed be *synthesized*. They could however have *inherited* attributes.
- The start symbol of the augmented grammar can have *only* synthesized attributes.
- In the case of LR parsing with its special start symbol, the start symbol *cannot have any* inherited attributes because
  1. it does not have any parent nodes in the parse tree and
  2. it does not occur on the right-hand side of any production.

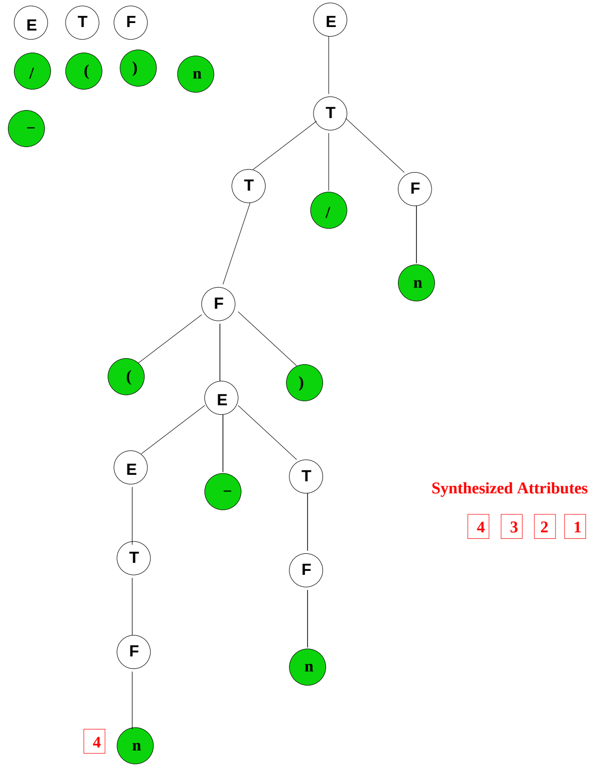
Synthesized Attributes

Evaluating the expression $(4 - 1)/2$ generated by the grammar for subtraction and division

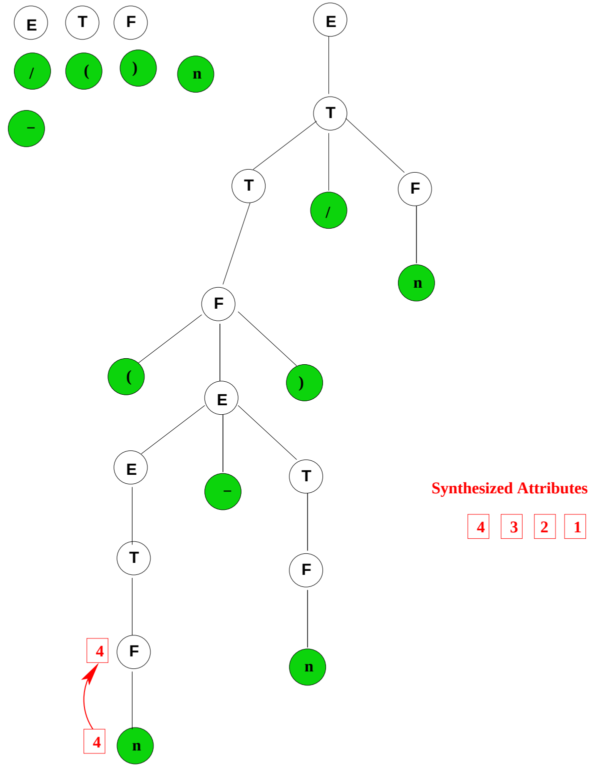
Synthesized Attributes: 0



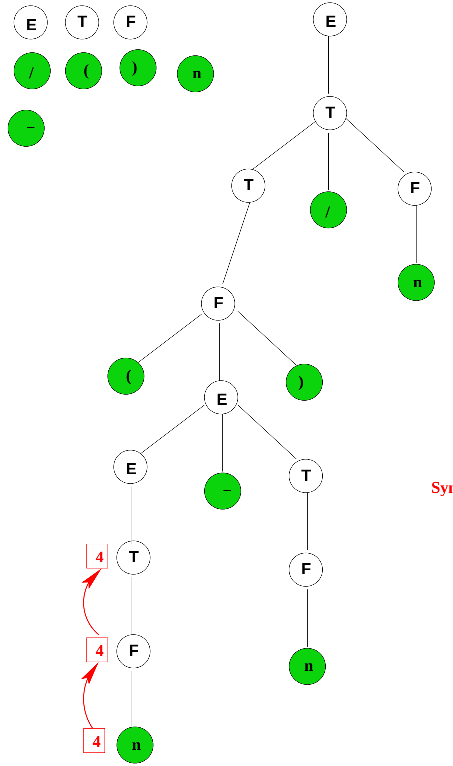
Synthesized Attributes: 1



Synthesized Attributes: 2



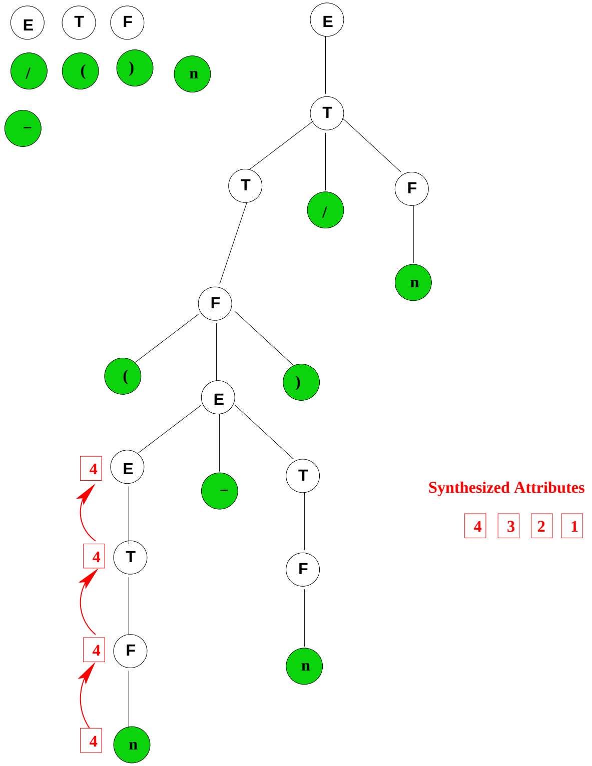
Synthesized Attributes: 3



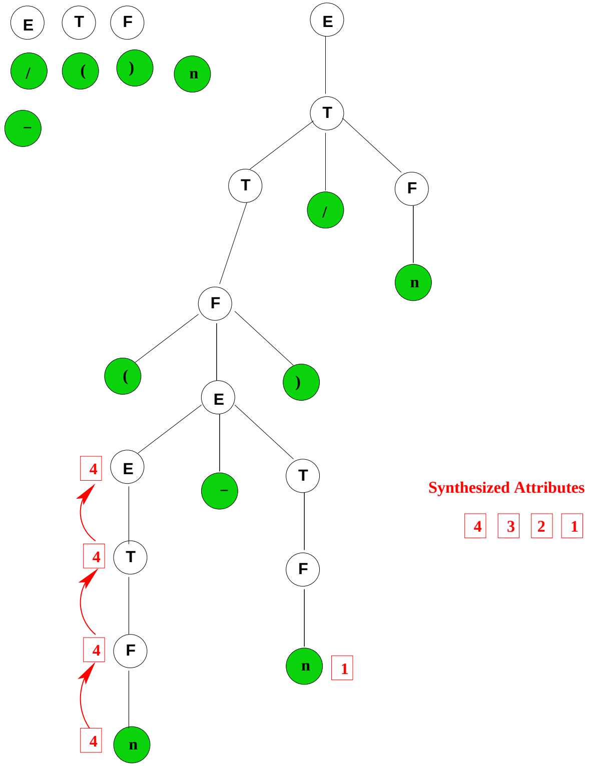
Synthesized Attributes

4 3 2 1

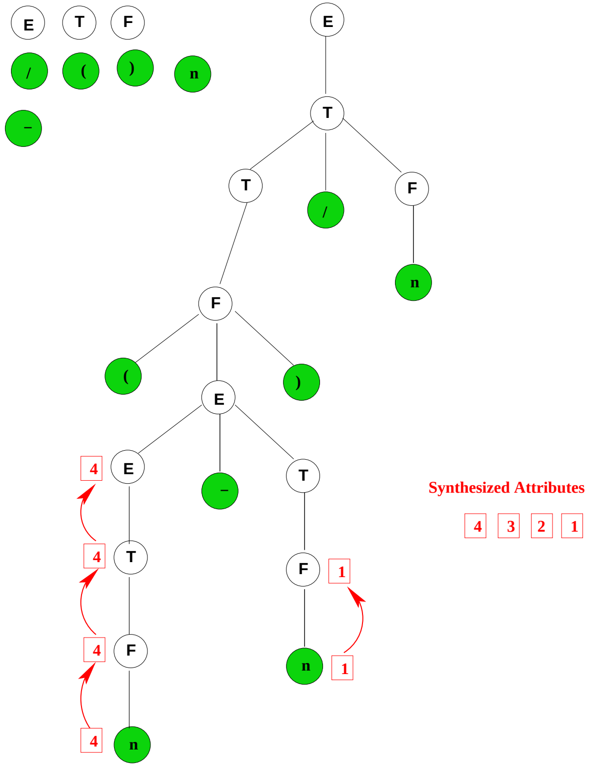
Synthesized Attributes: 4



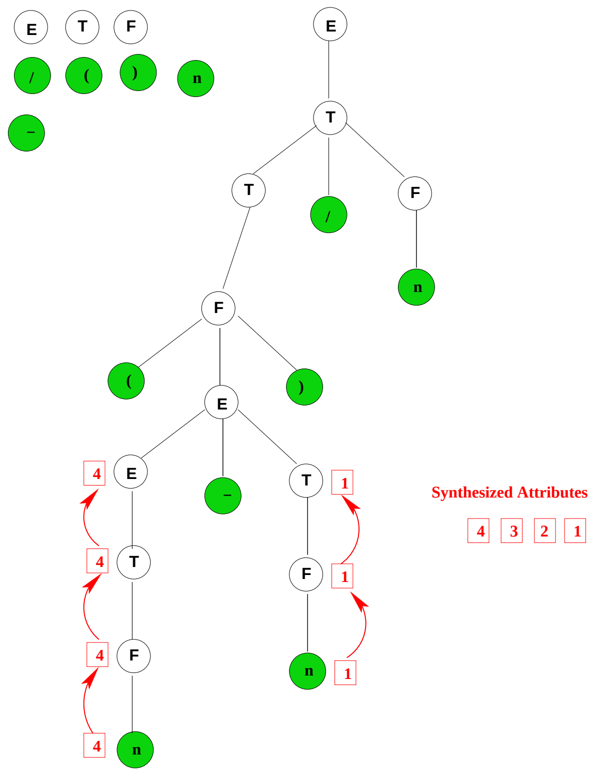
Synthesized Attributes: 5



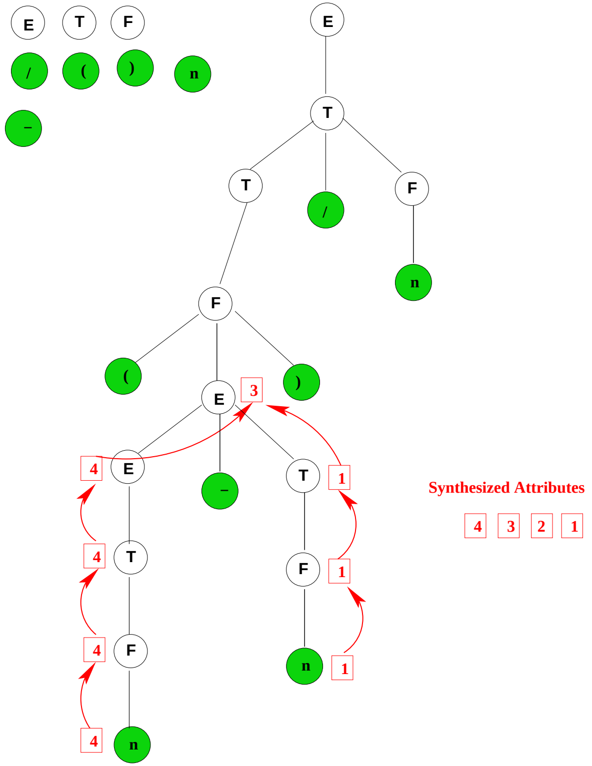
Synthesized Attributes: 6



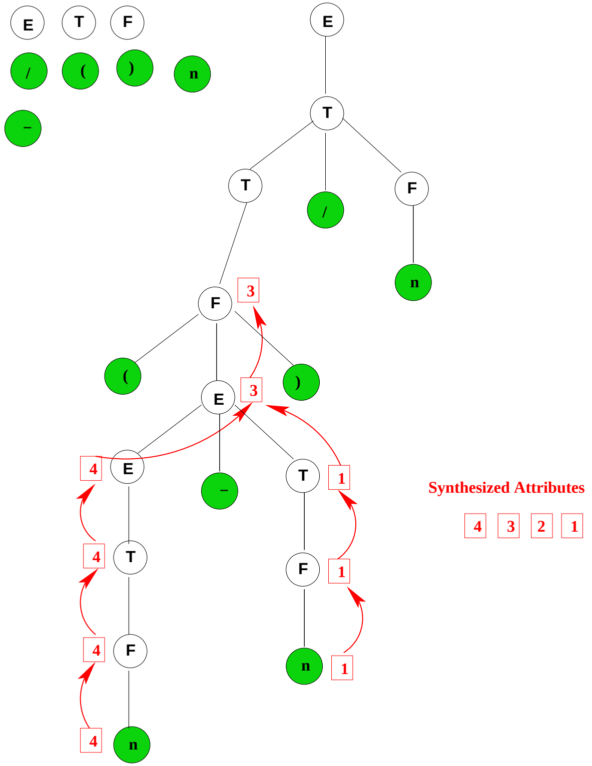
Synthesized Attributes: 7



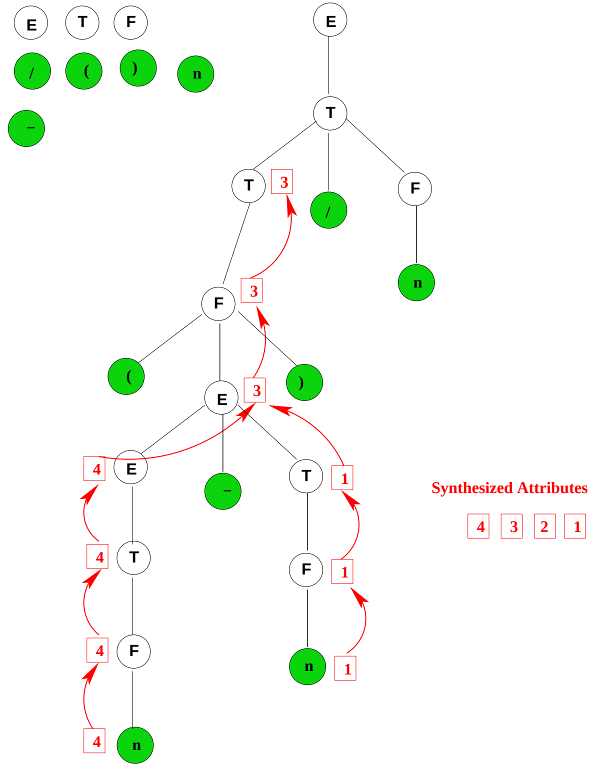
Synthesized Attributes: 8



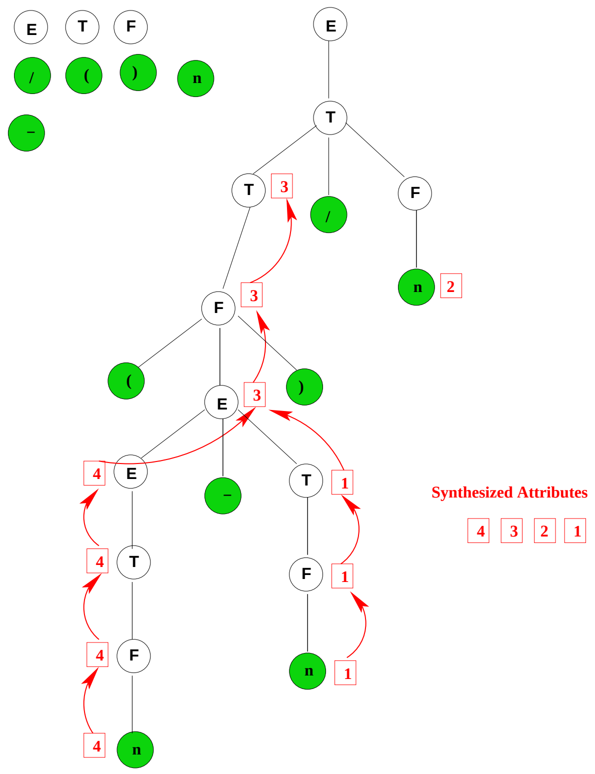
Synthesized Attributes: 9



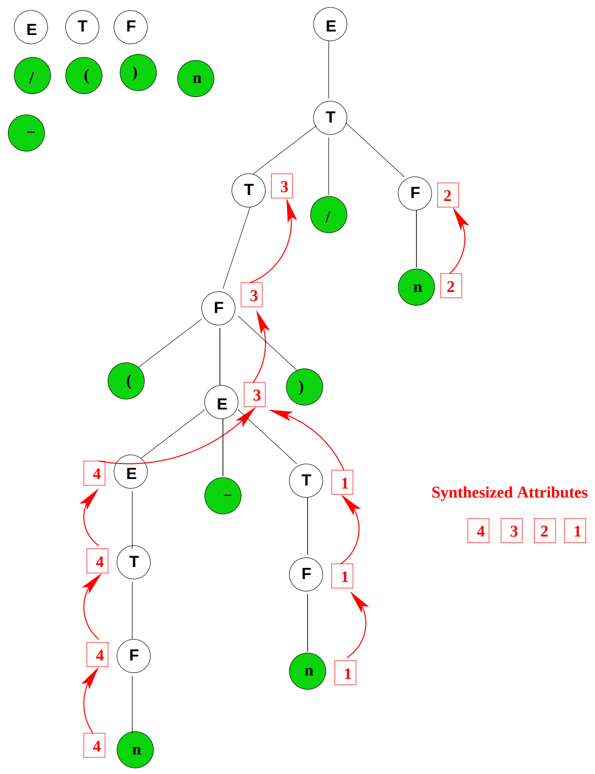
Synthesized Attributes: 10



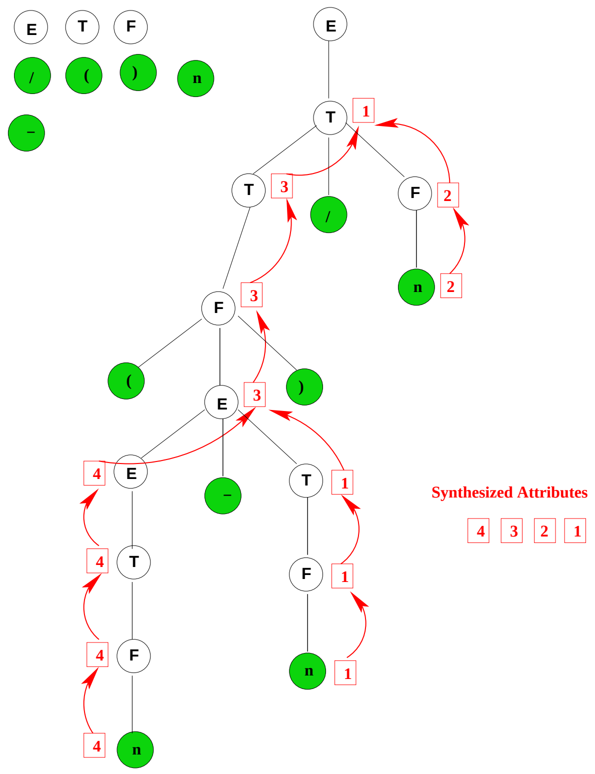
Synthesized Attributes: 11



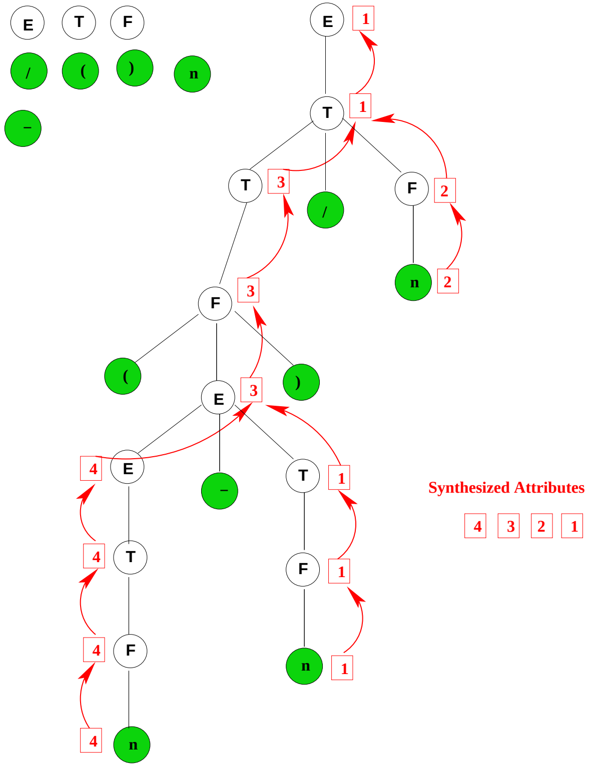
Synthesized Attributes: 12



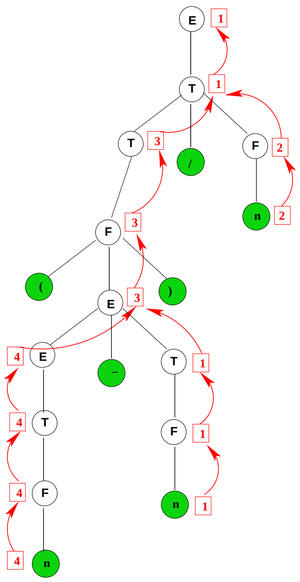
Synthesized Attributes: 13



Synthesized Attributes: 14



An Attribute Grammar



$$E_0 \to E_1 - T \quad \triangleright \quad E_0.val := \underline{sub}(E_1.val, T.val)$$

$$E \to T \quad \triangleright \quad E.val := T.val$$

$$T_0 \to T_1 / F \quad \triangleright \quad T_0.val := \underline{div}(T_1.val, F.val)$$

$$T \to F \quad \triangleright \quad T.val := F.val$$

$$F \to (E) \quad \triangleright \quad F.val := E.val$$

$$F \to \mathbf{n} \quad \triangleright \quad F.val := \mathbf{n}.val$$

Synthesized Attributes Evaluation: Bottom-up

During bottom-up parsing synthesized attributes are evaluated as follows:

Bottom-up Parsers

1. Keep an attribute value stack along with the parsing stack.
2. Just before applying a reduction of the form $Z \to Y_1 \ldots Y_k$ compute the attribute values of Z from the attribute values of $Y_1, \cdots, Y_k$ and place them in the same position on the attribute value stack corresponding to the one where the symbol Z will appear on the parsing stack as a result of the reduction.

Synthesized Attributes Evaluation: Top-down

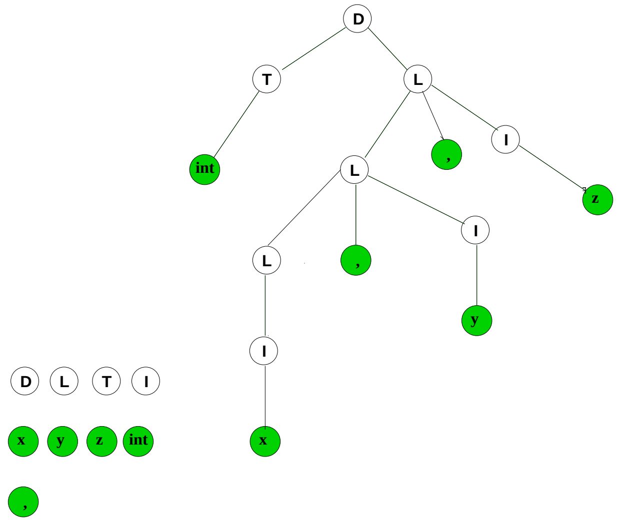
During top-down parsing synthesized attributes are evaluated as follows:

Top-down Parsers In any production of the form $Z \rightarrow Y_1 \ldots Y_k$, the parser makes recursive calls to procedures corresponding to the symbols $Y_1 \ldots Y_k$. In each case the attributes of the non-terminal symbols $Y_1 \ldots Y_k$ are computed and returned to the procedure for Z. Compute the synthesized attributes of Z from the attribute values returned from the recursive calls.

Inherited Attributes: 0

C-style declarations generating **int x, y, z.**

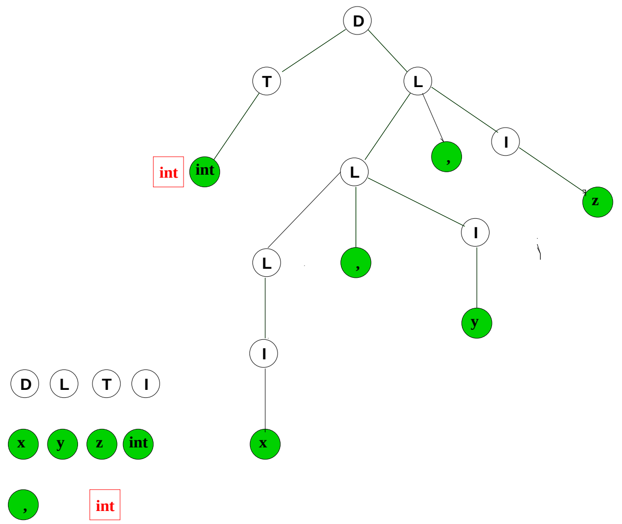
$$
\begin{array}{lll}
D \to T\ L & \qquad & T \to \mathbf{int} \mid \mathbf{float} \\
L \to L\mathbf{,}I \mid I & \qquad & I \to \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}
\end{array}
$$



Inherited Attributes: 1

C-style declarations generating **int x, y, z**.

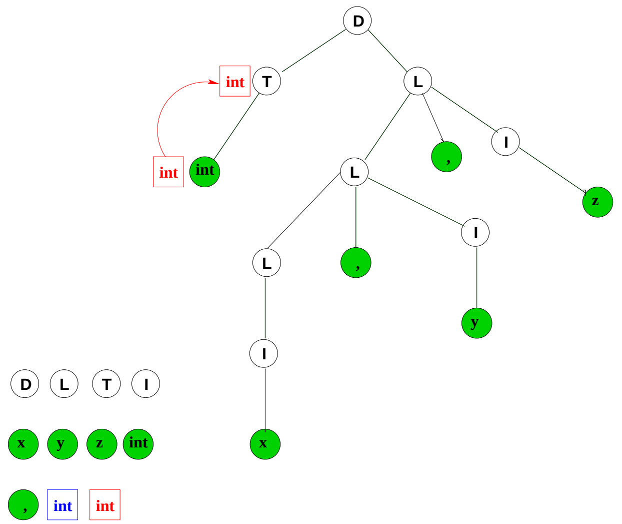
$$D \rightarrow T\ L \qquad\qquad T \rightarrow \textbf{int} \mid \textbf{float}$$
$$L \rightarrow L\textbf{,}I \mid I \qquad\qquad I \rightarrow \textbf{x} \mid \textbf{y} \mid \textbf{z}$$



Inherited Attributes: 2

C-style declarations generating **int x, y, z**.

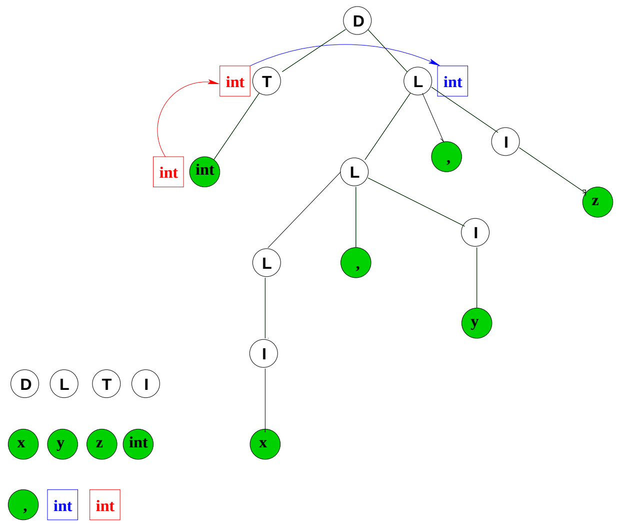
$$D \rightarrow T\ L \qquad\qquad T \rightarrow \textbf{int} \mid \textbf{float}$$
$$L \rightarrow L\textbf{,}I \mid I \qquad\qquad I \rightarrow \textbf{x} \mid \textbf{y} \mid \textbf{z}$$



Inherited Attributes: 3

C-style declarations generating **int x, y, z**.

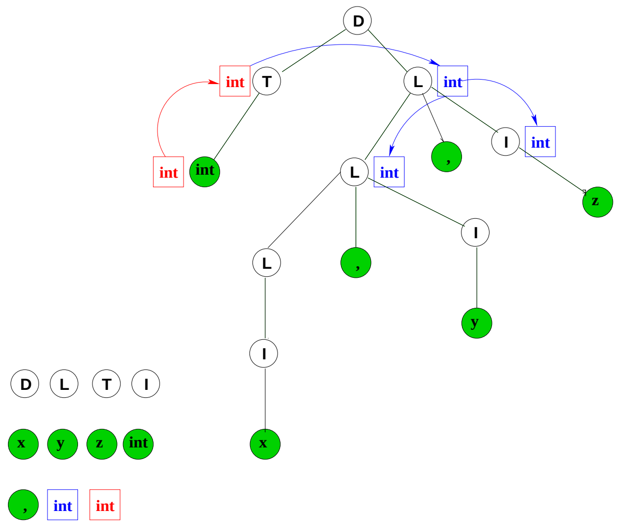
$$
\begin{array}{lllllll}
D & \rightarrow & T\ L & \qquad & T & \rightarrow & \textbf{int} \mid \textbf{float} \\
L & \rightarrow & L\textbf{,}I \mid I & \qquad & I & \rightarrow & \textbf{x} \mid \textbf{y} \mid \textbf{z}
\end{array}
$$



Inherited Attributes: 4

C-style declarations generating $\mathbf{int\ x, y, z}$.

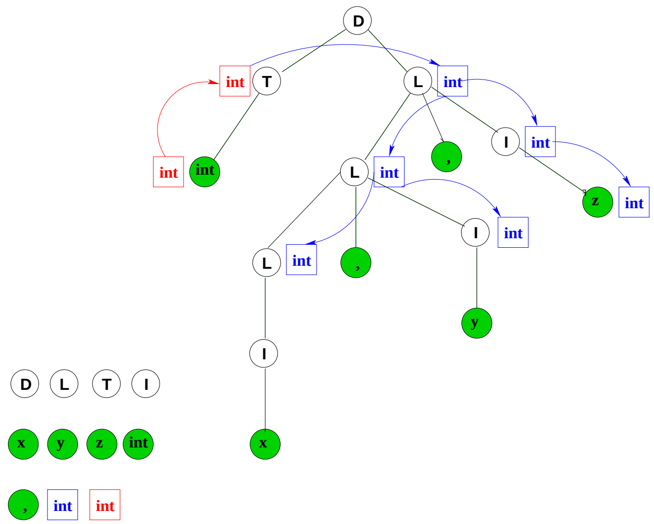
$$D \rightarrow T\ L \qquad\qquad T \rightarrow \mathbf{int} \mid \mathbf{float}$$
$$L \rightarrow L , I \mid I \qquad\qquad I \rightarrow \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}$$



Inherited Attributes: 5

C-style declarations generating **int x, y, z**.

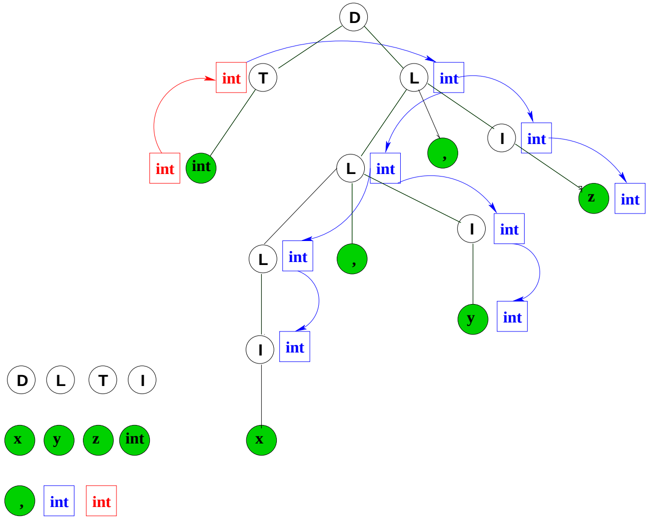
$$D \rightarrow T\ L \qquad\qquad T \rightarrow \textbf{int} \mid \textbf{float}$$
$$L \rightarrow L \textbf{,} I \mid I \qquad\qquad I \rightarrow \textbf{x} \mid \textbf{y} \mid \textbf{z}$$



Inherited Attributes: 6

C-style declarations generating **int x, y, z**.

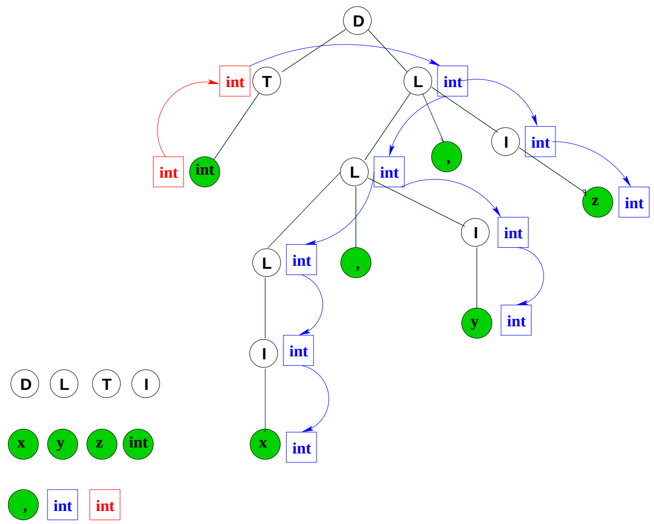
$$D \rightarrow T\ L \qquad\qquad T \rightarrow \textbf{int} \mid \textbf{float}$$
$$L \rightarrow L, I \mid I \qquad\qquad I \rightarrow \textbf{x} \mid \textbf{y} \mid \textbf{z}$$



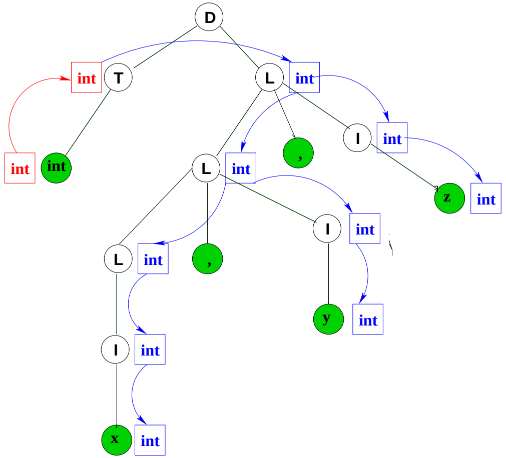
Inherited Attributes: 7

C-style declarations generating **int x, y, z**.

$$D \to T\ L \qquad\qquad T \to \mathbf{int} \mid \mathbf{float}$$
$$L \to L, I \mid I \qquad\qquad I \to \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}$$



Attribute Grammar: Inherited



$$D \rightarrow TL \quad \triangleright \; L.in := T.type$$

$$T \rightarrow \mathbf{int} \quad \triangleright \; T.type := \mathbf{int}.int$$

$$T \rightarrow \mathbf{float} \quad \triangleright \; T.type := \mathbf{float}.float$$

$$L_0 \rightarrow L_1,I \quad \triangleright \; L_1 := L_0.in$$

$$L \rightarrow I \quad \triangleright \; I.in := L.in$$

$$I \rightarrow \mathbf{id} \quad \triangleright \; \mathbf{id}.type := I.in$$

L-attributed Definitions

Definition 10.2 *A grammar is* L-attributed *if for each production of the form* $Y \to X_1 \dots X_k$, *each inherited attribute of the symbol* X_j, $1 \leq j \leq k$ *depends only on*

1. *the inherited attributes of the symbol* Y *and*
2. *the synthesized or inherited attributes of* $X_1, \cdots, X_{j-1}$.

Why L-attributedness? 1

$$Y \rightarrow X_1 \ldots X_k$$

Intuitively, if $X_j.inh$ is an inherited attribute then

- it cannot depend on any synthesized attribute $Y.syn$ of Y because it is possible that the computation of $Y.syn$ requires the value of $X_j.inh$ leading to circularity in the definition.

Why L-attributedness? 2

$$Y \rightarrow X_1 \dots X_k$$

Intuitively, if $X_j.inh$ is an inherited attribute then

- if the value of $X_j.inh$ depends upon the attributes of one or more of the symbols $X_{j+1}, \cdots, X_k$ then the computation of $X_j.inh$ cannot be performed just before the reduction by the rule $Y \rightarrow X_1 \dots X_k$ during parsing. Instead it may have to be postponed till the end of parsing.

Why L-attributedness? 3

$$Y \to X_1 \ldots X_k$$

Intuitively, if $X_j.inh$ is an inherited attribute then

- it could depend on the synthesized or inherited attributes of any of the symbols $X_1 \ldots X_{j-1}$ since they would already be available on the attribute value stack.

Why L-attributedness? 4

$$Y \rightarrow X_1 \ldots X_k$$

Intuitively, if $X_j.inh$ is an inherited attribute then

- it could depend upon the inherited attributes of Y because these inherited attributes can be computed from the attributes of the symbols lying below X_1 on the stack, provided these inherited attributes of Y are also L-attributed.

A Non L-attributed Definition

Our attribute grammar for C-style declarations is definitely L-attributed. However consider the following grammar for declarations in Pascal and ML.

$$
\begin{array}{llll}
D & \rightarrow L : T & \triangleright & L.in := T.type \\
T & \rightarrow \mathbf{int} & \triangleright & T.type := \mathbf{int}.int \\
T & \rightarrow \mathbf{real} & \triangleright & T.type := \mathbf{real}.real \\
L_0 & \rightarrow L_1 , I & \triangleright & L_1 := L_0.in \\
L & \rightarrow I & \triangleright & I.in := L.in \\
I & \rightarrow \mathbf{id} & \triangleright & \mathbf{id}.type := I.in
\end{array}
$$

In the first semantic rule the symbol $L.in$ is inherited from a symbol to its *right* viz. $T.type$ and hence is not L-attributed.

Evaluating Non-L-attributed Definitions 1

In many languages like ML which allow higher order functions as values, a definition not being L-attributed may not be of serious concern if the compiler is written in such a language.
But in most other languages it is serious enough to warrant changing the grammar of the language so as to replace inherited attributes by corresponding synthesized ones.

Evaluating Non-L-attributed Definitions 2

The language of the grammar of Pascal and ML declarations can be generated as follows (transforming the inherited attribute into a synthesised one).

$$
\begin{array}{lcll}
D & \to & \mathbf{id}L & \rhd\ addtype(\mathbf{id}, L.type) \\
L & \to & :T & \rhd\ L.in := T.type \\
L_0 & \to & ,\mathbf{id}\ L_1 & \rhd\ L_0.type := L_1.type; \\
 & & & \quad addtype(\mathbf{id}, L_1.type) \\
T & \to & \mathbf{int} & \rhd\ T.type := \mathbf{int}.int \\
T & \to & \mathbf{real} & \rhd\ T.type := \mathbf{real}.real
\end{array}
$$

Dependency Graphs

In general, the attributes required to be computed during parsing could be synthesized or inherited and further it is possible that some synthesized attributes of some symbols may depend on the inherited attributes of some other symbols. In such a scenario it is necessary to construct a dependency graph of the attributes of each node of the parse tree and check that it is acyclic.

Dependency Graph Construction

Algorithm 10.1

$\textsc{AttributeDependencyGraph}$

$$(T, A) \stackrel{df}{=}$$

$$\left\{\begin{array}{l}
\quad \textbf{Requires:} \text{ A parse tree } T \text{ and the list } A \text{ of attributes} \\
\quad \textbf{Yields:} \text{ An attribute dependency graph} \\
\textbf{for each } \text{node } n \text{ of } T \\
\quad \textbf{do } \left\{\begin{array}{l} \textbf{for each } \text{attribute } a \text{ of node } n \\ \quad \textbf{do } \text{Create an attribute node } n.a \end{array}\right. \\
\textbf{for each } \text{node } n \text{ of } T \\
\quad \textbf{do } \left\{\begin{array}{l} \textbf{for each } \text{semantic rule } a := f(b_1, \ldots, b_k) \\ \quad \textbf{do } \left\{\begin{array}{l} \textbf{for } i := 1 \textbf{ to } k \\ \quad \textbf{do } \text{Create a directed edge } b_i \rightarrow n.a \end{array}\right. \end{array}\right.
\end{array}\right.$$

Intermediate Representation

Intermediate Representation

Intermediate representations are important for reasons of portability i.e. platform (hardware and OS) independence.

- *(more or less)* independent of specific features of the high-level language.
  Example. Java byte-code which is the instruction set of the Java Virtual Machine (JVM).
- *(more or less)* independent of specific features of any particular target architecture (e.g. number of registers, memory size)
  - number of registers
  - memory size
  - word length

Typical Instruction set

IR Properties: Low vs high

1. It is fairly low-level containing instructions common to all target architectures and assembly languages.

   How low can you stoop? . . .

2. It contains some fairly high-level instructions that are common to most high-level programming languages.

   How high can you rise?

3. To ensure portability across architectures and OSs.

   Portability

4. To ensure type-safety

   Type safety

Typical Instruction set

IR: Representation?

- No commitment to word boundaries or byte boundaries
- No commitment to representation of
  - int vs. float,
  - float vs. double,
  - packed vs. unpacked,
  - strings – where and how?.

Back to IR Properties

IR: How low can you stoop?

- most arithmetic and logical operations, load and store instructions etc.
- so as to be interpreted easily,
- the interpreter is fairly small,
- execution speeds are high,
- to have fixed length instructions (where each operand position has a specific meaning).

Back to IR Properties

IR: How high can you rise?

- typed variables,
- temporary variables instead of registers.
- array-indexing,
- random access to record fields,
- parameter-passing,
- pointers and pointer management
- no limits on memory addresses

Back to IR Properties

IR Properties: Portability

1. How low can you stoop? ...
2. How high can you rise?
3. To ensure portability across architectures and OSs.
   - an unbounded number of variables and memory locations
   - no commitment to Representational Issues
4. Type safety

Back to IR Properties

IR Properties: Type Safety

1. How low can you stoop? ...
2. How high can you rise?
3. Portability
4. To ensure type-safety despite the hardware instruction set architectures.
   - Memory locations are also typed according to the data they may contain,
   - No commitment is made regarding word boundaries, and the structure of individual data items.

Back to IR Properties

Typical Instruction set

A typical instruction set: 1

Three address code: A suite of instructions. Each instruction has at most 3 operands.

- an opcode representing an operation with at most 2 operands
- two operands on which the binary operation is performed
- a target operand, which accumulates the result of the (binary) operation.

If an operation requires less than 3 operands then one or more of the operands is made null.

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

c.f. Java byte-code

A typical instruction set: 2.1

- Assignments (LOAD-STORE)
  - `x := y bop z`, where `bop` is a binary operation
  - `x := uop y`, where `uop` is a unary operation
  - `x := y`, load, store, copy or register transfer
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2.2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
  - `goto L` – Unconditional jump,
  - `x relop y goto L` – Conditional jump, where `relop` is a relational operator
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2.3

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
  - `call p n`, where `n` is the number of parameters
  - `return y`, return value from a procedures call
  - `param x`, parameter declaration
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2.4

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
  - `x := a[i]` – array indexing for *r-value*
  - `a[j] := y` – array indexing for *l-value*

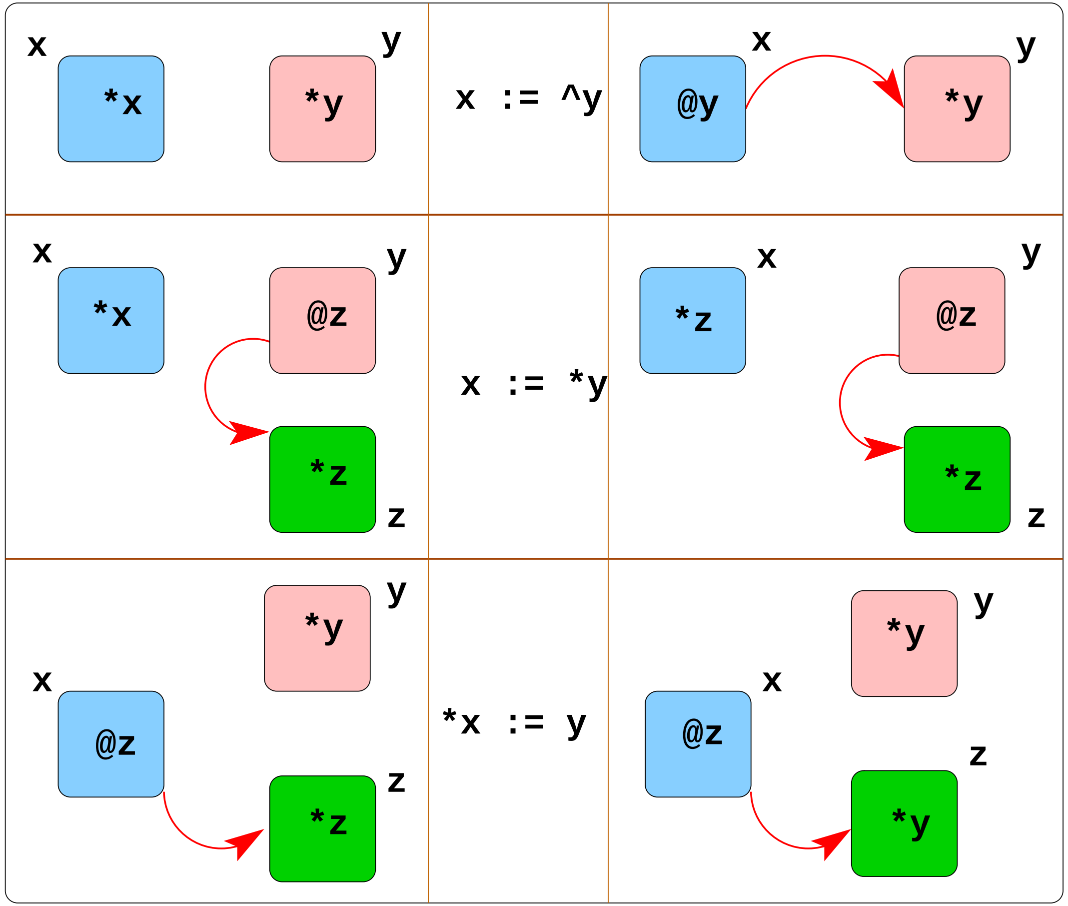
  Note: The two opcodes are different depending on whether *l-value* or *r-value* is desired. `x` and `y` are *always* simple variables
- Pointer Referencing and Dereferencing

A typical instruction set: 2.5

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
  - `x := ^y` – referencing: set `x` to point to `y`
  - `x := *y` – dereferencing: copy contents of location pointed to by `y` into `x`
  - `*x := y` – dereferencing: copy *r-value* of `y` into the location pointed to by `x`

Picture

Pointers



IR: Generation Basics

- Can be generated by recursive traversal of the abstract syntax tree.
- Can be generated by syntax-directed translation as follows:

  For every non-terminal symbol N in the grammar of the source language there exist two attributes

  N.place , which denotes the address of a temporary variable where the result of the execution of the generated code is stored

  N.code , which is the actual code segment generated.
- In addition a global counter for the instructions generated is maintained as part of the generation process.
- It is independent of the source language but can express target machine operations without committing to too much detail.

IR: Infrastructure 1

Given an abstract syntax tree T, with T also denoting its root node.

T.place address of temporary variable where result of execution of the T is stored.

$newtemp$ returns a *fresh* variable name and also installs it in the symbol table along with relevant information

T.code the actual sequence of instructions generated for the tree T.

$newlabel$ returns a *label* to mark an instruction in the generated code which may be the target of a jump.

$emit$ emits an instructions (regarded as a string).

IR: Infrastructure 2

Colour and font coding of IR code generation process.

- *Green*: Nodes of the Abstract Syntax Tree
- `Brown`: `Intermediate Representation` i.e. the language of the "virtual machine"
- *Red*: Variables and data structures of the *language* in which the IR code generator is written
- *Blue*: Names of relevant *procedures* used in IR code generation.
- *Black*: All other stuff.

IR: Expressions

$$
\begin{aligned}
E &\rightarrow id \qquad \triangleright\\
E.place &:= id.place;\\
E.code &:= emit()
\end{aligned}
$$

$$
\begin{aligned}
E_0 &\rightarrow E_1 - E_2 \qquad \triangleright\\
E_0.place &:= newtemp;\\
E_0.code &:= E_1.code;\\
&\quad E_2.code;\\
&\quad emit(E_0.place := E_1.place - E_2.place)
\end{aligned}
$$

The WHILE Language

Assume there is a language of expressions (with start symbol E) over which the statements are defined. For simplicity assume these are the only constructs of the language.

$$
\begin{array}{rlr}
S \rightarrow & id := E & \text{Assignment} \\
\mid & S; S & \text{Sequencing} \\
\mid & if\ E\ then\ S\ else\ S fi & \text{Conditional} \\
\mid & while\ E\ do\ S\ od & \text{Iteration}
\end{array}
$$

IR: Assignment and Sequencing

$$S \rightarrow id := E \quad \triangleright$$

$$\begin{aligned} S.code &:= E.code \\ &\quad emit(id.place{:=}E.place) \end{aligned}$$

$$S_0 \rightarrow S_1;\ S_2 \quad \triangleright$$

$$\begin{aligned} S_0.begin &:= S_1.begin; \\ S_0.after &:= S_2.after; \\ S_0.code &:= emit(S_0.begin:) \\ &\quad S_1.code \\ &\quad S_2.code \\ &\quad emit(S_0.after:) \end{aligned}$$

IR: Conditional

$$S_0 \rightarrow if\ E\ then\ S_1\ else\ S_2 fi \qquad \triangleright$$

$$\begin{aligned}
S_0.begin &:= newlabel;\\
S_0.after &:= S_2.after;\\
S_0.code &:= emit(S_0.begin:)\\
& \quad E.code;\\
& \quad emit(\texttt{if}\ E.place = \ \texttt{0 goto}\ S_2.begin);\\
& \quad S_1.code;\\
& \quad emit(\texttt{goto}\ S_0.after);\\
& \quad S_2.code;\\
& \quad emit(S_0.after:)
\end{aligned}$$

Selective Evaluation.

Notice that the evaluation/execution of the Conditional is such that only one arm of the conditional is evaluated/executed depending upon the truth value of the condition. This is perfectly consistent with the semantics of the conditional. It is also consistent with the functional semantics of the conditional construct in FL(X). Similar remarks also apply to iteration construct defined below.

IR: Iteration

$$S_0 \rightarrow \textit{while } E \textit{ do } S_1 \textit{ od} \qquad \triangleright$$

$$
\begin{array}{lcl}
S_0.begin & := & newlabel; \\
S_0.after & := & newlabel; \\
S_0.code & := & emit(S_0.begin:) \\
 & & E.code \\
 & & emit(\texttt{if } E.place = \texttt{ 0 goto } S_0.after); \\
 & & S_1.code; \\
 & & emit(\texttt{goto } S_0.begin); \\
 & & emit(S_0.after:)
\end{array}
$$

IR: Generation End

While generating the intermediate representation, it is sometimes necessary to generate jumps into code that has not been generated as yet (hence the address of the label is unknown). This usually happens while processing

- forward jumps
- short-circuit evaluation of boolean expressions

It is usual in such circumstances to either fill up the empty label entries in a second pass over the the code or through a process of *backpatching* (which is the maintenance of lists of jumps to the same instruction number), wherein the blank entries are filled in once the sequence number of the target instruction becomes known.

The Pure Untyped Lambda calculus

Curiously a systematic notation for functions is lacking in ordinary mathematics. The usual notation '$f(x)$' does not distinguish between the function itself and the value of this function for an undetermined value of the argument.

Haskell B Curry *Combinatory Logic vol 1*

Let us consider the nature of functions, higher-order functions (functionals) and the use of naming in mathematics, through some examples.

Example 12.1 *Let $y = x^2$ be the squaring function on the reals. Here it is commonly understood that x is the "independent" variable and y is the "dependent" variable when we look on it as plotting the function $f(x) = x^2$ on the $x - y$ axis.*

Example 12.2 *Often a function may be named and written as $f(x) = x^n$ to indicate that x is the independent variable and n is understood (somehow!) to be some constant. Here f, x and n are all names with different connotations. Similarly in the quadratic polynomial $ax^2 + bx + c$ it is somehow understood that a, b and c denote constants and that x is the independent variable. Implicitly by using the names like a, b and c we are endeavouring to convey the impression that we consider the class $\{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$ of all quadratic polynomials of the given form.*

Example 12.3 *As another example, consider the uni-variate polynomial $p(x) = x^2 + 2x + 3$. Is this polynomial the same as $p(y) = y^2 + 2y + 3$? Clearly they cannot be the same since the product $p(x).p(y)$ is a polynomial in two variables whereas $p(x).p(x)$ yields a uni-variate polynomial of degree 4. However,* *in the case of the function f in example 12.1 it does not matter whether we define the squaring function as $f(x) = x^2$ or as $f(y) = y^2$.*

Example 12.4 *The squaring function 12.1 is a continuous and differentiable real-valued function (in the variable x) and its derivative is $f'(x) = 2x$. Whether we regard f' as the name of a new function or we regard the $'$ as an operation on f which yields its derivative seems to make no difference.*

Example 12.5 *Referring again to the functions $f(x)$ and $f'(x)$ in example 12.4, it is commonly understood that $f'(0)$ refers to the value of the derivative of f at 0 which is also the value the function f' takes at 0. Now let us consider $f'(x+1)$. Going by the commonly understood notion, since $f'(x) = 2x$, we would have $f'(x+1) = 2(x+1)$. Then for $x = 0$ we have $f'(x+1) = f'(0+1) = f'(1) = 2 \times 1 = 2$. We could also think of it as the function $f'(g(0))$ where g is the function defined by $g(x) = x+1$, then $f'(g(0)) = 2g(0) = 2$ which yields the same result.*

The examples above give us some idea of why there is no systematic notation for functions which distinguishes between a function definition and the application of the same function to some argument. It simply did not matter!

However, this ambiguity in mathematical notation could lead to differing interpretationas and results in the context of mathematical theories involving higher-order functions (or "functionals" as they are often referred to). One common higher order function is the derivative (the differentiation operation) and another is the indefinite integral. Most mathematical texts emphasize the higher-order nature of a function by enclosing their arguments in (square) brackets. Hence if O is a functional which transforms a function $f(x)$ into a function $g(x)$, this fact is usually written $O[f(x)] = g(x)$.

Example 12.6 *Consider the functional E (on continuous real-valued functions of one real variable x) defined as follows.*

$$E[f(x)] = \begin{cases} f'(0) & \textit{if } x = 0 \\ \dfrac{f(x) - f(0)}{x} & \textit{if } x \neq 0 \end{cases}$$

The main question we ask now is "What does $E[f(x+1)]$ mean?"

It turns out that there are at least two ways of interpreting $E[f(x+1)]$ and unlike the case of example 12.5, the two interpretations actually yield different results!.

1. *We may interpret $E[f(x+1)]$ to mean that we first apply the transformation E to the function $f(x)$*

and then substitute $x+1$ *for* x *in the resulting expression. We then have the following.*

$$\begin{aligned} & E[f(x)] \\ = & \begin{cases} f'(0) & \text{if } x = 0 \\ \dfrac{f(x)-f(0)}{x} & \text{if } x \neq 0 \end{cases} \\ = & \begin{cases} 0 & \text{if } x = 0 \\ x & \text{if } x \neq 0 \end{cases} \\ = & \; x \end{aligned}$$

Since $E[f(x)] = x$, $E[f(x+1)] = x+1$.

2. *Since* $f(x+1) = f(g(x))$ *where* $g(x) = x+1$, *we may interpret* $E[f(x+1)]$ *as applying the operator* E *to the function* $h(x) = f(g(x))$. *Hence* $E[f(x+1)] = E[h(x)]$ *where* $h(x) = f(g(x)) = (x+1)^2 = x^2+2x+1$.

Noting that $h'(x) = 2x + 2$, $h(0) = 1$ *and* $h'(0) = 2$, *we get*

$$
\begin{aligned}
& E[h(x)] \\
= & \begin{cases} h'(0) & \textit{if } x = 0 \\ \dfrac{h(x) - h(0)}{x} & \textit{if } x \neq 0 \end{cases} \\
= & \begin{cases} 2 & \textit{if } x = 0 \\ x + 2 & \textit{if } x \neq 0 \end{cases} \\
= & \; x + 2
\end{aligned}
$$

The last example should clearly convince the reader that there is a need to disambiguate between a function definition and its application.

12.2. The λ-notation

In function definitions the independent variables are "bound" by a λ which acts as a pre-declaration of the name that is going to be used in the expression that defines a function.

The notation $f(x)$, which is interpreted to refer to "the value of function f at x", will be replaced by $(f\ x)$ to denote an application of a function f to the (known or unknown) value x.

In our notation of the untyped applied λ-calculus the functions and their applications in the examples in subsection 12.1 would be rewritten as follows.

Squaring . $\lambda\ x[x^2]$ is the squaring function.

Example 12.2 . $q \stackrel{df}{=} \lambda\ a\ b\ c\ x[ax^2 + bx + c]$ refers to any quadratic polynomial with coefficients unknown or symbolic. To obtain a particular member of this family such as $1x^2 + 2x + 3$, one would have to evaluate $(((q\ 1)\ 2)\ 3)$ which would yield $\lambda\ x[1x^2 + 2x + 3]$.

Example 12.3 . $p \stackrel{df}{=} \lambda\ x[x^2 + 2x + 3]$. Then $p(x)$ would be written as $(p\ x)$ i.e. as the function p applied to the argument x to yield the expression $x^2 + 2x + 3$. Likewise $p(y)$ would be $(p\ y)$ which would yield $y^2 + 2y + 3$. The products $(p\ x).(p\ x)$ and $(p\ x).(p\ y)$ are indeed different and distinct.

Example 12.5 Let us denote the operation of obtaining the derivative of a real-valued function f of one independent variable x by the simple symbol D (instead of the more confusing $\frac{d}{dx}$). Then for any function f, $(D\ f)$ would yield the derivative. In particular $(D\ \lambda\ x[x^2]) = \lambda\ x[2x]$ and the value of the derivative at 0 would be obtained by the application $(\lambda\ x[2x]\ 0)$ which would yield 0. Likewise the value of the derivative at $x+1$ would be expressed as the application $(\lambda\ x[2x]\ (x+1))$. Thus for any function f the value of its derivative at $x+1$ is simply the application $((D\ f)\ (x+1))$.

The function $g(x) = x+1$ would be defined as $g \stackrel{df}{=} \lambda\ x[x+1]$ and $(g\ x) = x+1$. Thus the alternative definition of the derivative of f at $x+1$ is simply the application $((D\ f)\ (g\ x))$.

Example 12.6 The two interpretations of the expression $E[f(x+1)]$ are respectively the following.

1. $((E\ f)\ (x+1))$ and
2. $((E\ h)\ x)$ where $h \stackrel{df}{=} \lambda\ x[(f\ (g\ x))]$

Pure Untyped λ-Calculus: Syntax

The language Λ of pure untyped λ-terms is the smallest set of terms built up from an infinite set V of *variables* and closed under the following productions

$$
\begin{array}{rcll}
L, M, N & ::= & x & \text{Variable} \\
& \mid & \lambda x[L] & \text{Abstraction} \\
& \mid & (L \;\; M) & \text{Application}
\end{array}
$$

where $x \in V$.

- A *Variable* denotes a possible binding in the external environment.
- An *Abstraction* denotes a function which takes a formal parameter.
- An *Application* denotes the application of a function to an actual parameter.

The language Λ

- The language Λ is "pure" (as opposed to being "applied") in the sense that it is minimal and symbolic and does not involve any operators other *abstraction* and *application*.
- When used in the context of some algebraic system (e.g. the algebra of integers or reals) it becomes applied. Hence the example of using the λ-notation in the differential calculus is one of an applied λ-calculus.
- It is purely symbolic and no types have been specified which put restrictions on the use of variables in contexts. We will look at typing much later.

Free Variables

Definition 12.7 *For any term $N \in \Lambda$ the set of* **free variables** *and the set of all variables are defined by induction on the structure of terms.*

N	$FV(N)$	$Var(N)$
x	$\{x\}$	$\{x\}$
$\lambda x[L]$	$FV(L) - \{x\}$	$Var(L) \cup \{x\}$
$(L \quad M)$	$FV(L) \cup FV(M)$	$Var(L) \cup Var(M)$

Bound Variables

- The set of **bound variables** $BV(N) = Var(N) - FV(N)$.
- The same variable name may be used with different bindings in a single term (e.g. $(\lambda x[x] \;\; \lambda x[(x \;\; y)])$)
- The brackets "$[$" and "$]$" delimit the scope of the bound variable x in the term $\lambda x[L]$.
- The usual rules of static scope apply to λ-terms.

Closed Terms and Combinators

Definition 12.8

- $\Lambda_0 \subseteq \Lambda$ *is the set of closed* λ*-terms (i.e. terms with no free variables).*
- *A* λ *abstraction with no free variables is called a* **combinator**[a]*.*

The λ-terms corresponding to D (section 12.2) and E (section 12.2) must be combinators too.

[a]Combinators represent function definitions

Notational Conventions

To minimize use of brackets and parentheses unambiguously

1. $\lambda x_1 x_2 \ldots x_m[L]$ denotes $\lambda x_1[\lambda x_2[\ldots \lambda x_m[L] \cdots]]$ i.e. L is the scope of each of the variables $x_1, x_2, \ldots x_m$.
2. $(L_1\ L_2\ \cdots L_m)$ denotes $(\cdots(L_1\ L_2)\ \cdots L_m)$ i.e. application is *left-associative*.

Substitution

Definition 12.9 *For any terms L, M and N and any variable x, the* **substitution** *of the term N for a variable x is defined as follows:*

$$
\begin{array}{lll}
\{N/x\}x & \equiv N & \\
\{N/x\}y & \equiv y & \text{if } y \not\equiv x \\
\{N/x\}\lambda x[L] & \equiv \lambda x[L] & \\
\{N/x\}\lambda y[L] & \equiv \lambda y[\{N/x\}L] & \text{if } y \not\equiv x \text{ and } y \notin FV(N) \\
\{N/x\}\lambda y[L] & \equiv \lambda z[\{N/x\}\{z/y\}L] & \text{if } y \not\equiv x \text{ and } y \in FV(N) \text{ and} \\
 & & z \text{ is 'fresh'} \\
\{N/x\}(L\ \ M) & \equiv (\{N/x\}L\ \ \{N/x\}M) &
\end{array}
$$

Lemma 12.10 *If L and N are pure λ-terms and x is a variable symbol then $\{N/x\}L$ is a pure λ-term.*

Proof: By induction on the structure of the λ-term L. QED

Notes on Substitution

- In the above definition it is necessary to ensure that the free variables of N continue to remain free after substitution i.e. none of the free variables of N should be "captured" as a result of the substitution.
- The phrase "z is 'fresh' " may be taken to mean $z \notin FV(N) \cup Var(L)$.
- Λ is closed under the syntactic operation of substitution.
- Substitution is the only operation required for function application in the pure λ-calculus.

Compatibility

Definition 12.11 *A binary relation* $\rho \subseteq \Lambda \times \Lambda$ *is said to be* **compatible** *if* $L\ \rho\ M$ *implies*

1. *for all variables* x, $\lambda x[L]\ \rho\ \lambda x[M]$ *and*

2. *for all terms* N, $(L\ \ N)\ \rho\ (M\ \ N)$ *and* $(N\ \ L)\ \rho\ (N\ \ M)$.

Compatible Closure

Definition 12.12 *The* **compatible closure** *of a relation* $\rho \subseteq \Lambda \times \Lambda$ *is the smallest (under the* $\subseteq$ *ordering) relation* $\rho^c \subseteq \Lambda \times \Lambda$ *such that*

$$\rho \ \frac{L \ \rho \ M}{L \ \rho^c \ M}$$

$$\rho\mathbf{Abs} \ \frac{L \ \rho^c \ M}{\lambda x[L] \ \rho^c \ \lambda x[M]}$$

$$\rho\mathbf{AppL} \ \frac{L \ \rho^c \ M}{(L \ \ N) \ \rho^c \ (M \ \ N)}$$

$$\rho\mathbf{AppR} \ \frac{L \ \rho^c \ M}{(N \ \ L) \ \rho^c \ (N \ \ M)}$$

Compatible Closure: Properties

Lemma 12.13

1. $\rho^c \supseteq \rho$.

2. The compatible closure of any relation is compatible.

3. If ρ *is compatible then* $\rho^c = \rho$.

Example 12.14

1. $\equiv_\alpha$ *is a compatible relation*

2. $\rightarrow^1_\beta$ *is by definition a compatible relation.*

α-equivalence

Definition 12.15 **(α-equivalence)** $\equiv_\alpha \subseteq \Lambda \times \Lambda$ *is the compatible closure of the relation* $\{(\lambda x[L] \equiv_\alpha \lambda y[\{y/x\}L]) \mid y \notin FV(L)\}$.

- α-equivalence implies that the the *name(s)* of the bound (called "independent" in normal mathematics) variable(s) in a function definition is unimportant[a]. Hence $\lambda x[x^2] \equiv_\alpha \lambda y[y^2]$[b].
- As long as distinct bound variable names do not clash within the same or nested scopes (where they need to be kept visible)[c] one can substitute the other.
- Condition $y \notin FV(L)$ is necessary to ensure that a "free" y is not captured by the new "bound" variable y.

[a]it corresponds exactly to uniformly replacing a variable name in a local context in a program by another variable name throughout the block provided there is no clash of variable names.

[b]See also **??**

[c]Whenever they need to be 'collapsed' e.g. when we need the value of $f(x, x)$ as an instance of a function $f(x, y)$, we need to explicitly apply f to the argument pair (x, x).

Function application: Basic β-Reduction

Definition 12.16

- *Any (sub-)term of the form* $(\lambda x[L]\ \ M)$ *is called a* *β-redex*
- *Basic β-reduction is the relation on* Λ

$$\rightarrow_\beta \stackrel{df}{=} \{((\lambda x[L]\ \ M), \{M/x\}L') \mid L' \equiv_\alpha L, L', L, M \in \Lambda\}$$

- *It is usually represented by the axiom*

$$\boxed{(\lambda x[L]\ \ M) \rightarrow_\beta \{M/x\}L'} \tag{6}$$

where $L' \equiv_\alpha L$.

Function application: 1-step β-Reduction

Definition 12.17 *A 1-step β-reduction $\to_\beta^1$ is the smallest relation (under the $\subseteq$ ordering) on Λ such that*

$$\beta_1 \quad \frac{L \to_\beta M}{L \to_\beta^1 M} \qquad\qquad \beta_1\mathbf{Abs} \quad \frac{L \to_\beta^1 M}{\lambda x[L] \to_\beta^1 \lambda x[M]}$$

$$\beta_1\mathbf{AppL} \quad \frac{L \to_\beta^1 M}{(L \;\; N) \to_\beta^1 (M \;\; N)} \qquad\qquad \beta_1\mathbf{AppR} \quad \frac{L \to_\beta^1 M}{(N \;\; L) \to_\beta^1 (N \;\; M)}$$

- $\to_\beta^1$ is the compatible closure of basic β-reduction to all contexts.
- We will often omit the superscript ^{1} as understood.

Definition 12.18

- *For all integers $n \geq 0$, n-step β-reduction $\rightarrow^n_\beta$ is defined by induction on 1-step β-reduction*

$$\beta_{\mathbf{n}}\mathbf{Basis}\quad \frac{}{L \rightarrow^0_\beta L}$$

$$\beta_{\mathbf{n}}\mathbf{Induction}\quad \frac{L \rightarrow^m_\beta M \rightarrow^1_\beta N}{L \rightarrow^{m+1}_\beta N}\ (m \geq 0)$$

- *β-reduction $\rightarrow^*_\beta$ is the reflexive-transitive closure of 1-step β-reduction. That is,*

$$\beta_*\quad \frac{L \rightarrow^n_\beta M}{L \rightarrow^*_\beta M}\ (n \geq 0)$$

Computations and Normal Forms

- Loosely speaking, by a normal form we mean a term that cannot be "simplified" further. In some sense it is like a "final answer".
- We use β-reduction as the only way to "compute" final answers by simplification.
- There may be more than one β-redex in a term – this may lead to different ways of computing the final answer.

Main Question: Do all the different ways of computing yield the same answer?

Function Calls

Let

$$f = \lambda x[x^2 + 1]$$

$$g = \lambda y[3.y + 2]$$

Consider two different evaluations of the function call $(f\ (g\ 4))$

$$\begin{array}{rl|rl}
 & \textit{Call-by-value} & & \textit{Call-by-name/text} \\
 & (f\ (g\ 4)) & & (f\ (g\ 4)) \\
= & (f\ (3.4 + 2)) & = & (g\ 4)^2 + 1 \\
= & (f\ 14) & = & (3.4 + 2)^2 + 1 \\
 & & = & (12 + 2)^2 + 1 \\
= & 14^2 + 1 & = & 14^2 + 1 \\
= & 196 + 1 & = & 196 + 1 \\
= & 197 & = & 197
\end{array}$$

Function Composition

- Let $F \equiv x^2 + 1$ be an expression involving one independent variable x and let $f \stackrel{df}{=} \lambda x[F]$
- Let $G \equiv 3.y + 2$ be an expression involving one independent variable y and let $g \stackrel{df}{=} \lambda y[G]$.
- Let $h = \lambda f[\lambda g[\lambda z[(f\ (g\ z))]]$. Then h is the composition of f and g i.e. $h = (f\ o\ g)$.

The function call $(f\ (g\ a))$ for some value a is $(h\ a)$ which is exactly $((f\ o\ g)\ a)$. Hence There are at least two different ways of evaluating the composition of functions.

Evaluating Function Composition

Call-by-value.

1. *First evaluate* $(g\ a) = \{a/y\}G$ *yielding a value* b.
2. *Then evaluate* $(f\ b) = \{b/x\}F$ *yielding a value* c.

Call-by-text.

1. *First evaluate* $(f\ (g\ y)) = \{(g\ y)/x\}F = \{G/x\}F$ *yielding expression* H *which contains only* y *as independent variable.* This expression represents a function $h \stackrel{df}{=} \lambda y[H]$.
2. *Evaluate* $(h\ a) = \{a/y\}H$ *yielding a value* d.

Main Question: Is $c = d$ always?

Untyped λ-Calculus: Normalization

Definition 12.19

- *A term is called a β-normal form (β-nf) if it has no β-redexes.*
- *A term is weakly normalising (β-WN) if it can reduce to a β-normal form.*
- *A term L is strongly normalising (β-SN) if it has no infinite reduction sequence $L \to^1_\beta L_1 \to^1_\beta L_2 \to^1_\beta \cdots$*

Intuitively speaking a β-normal form is one that cannot be "reduced" further.

Example 12.20

1. $\mathsf{K} \stackrel{df}{=} \lambda x\ y[x]$ a projection *function.*

2. $\mathsf{I} \stackrel{df}{=} \lambda x[x]$, *the* identity *function.*

3. $\mathsf{S} \stackrel{df}{=} \lambda x\ y\ z[((x\ \ z)\ \ (y\ \ z))]$, a generalized composition *function*

4. $\omega \stackrel{df}{=} \lambda x[(x\ \ x)]$

are all β*-nfs.*

Examples of Strong Normalization

Example 12.21

1. $((\mathsf{K}\ \ \omega)\ \ \omega)$ *is* $strongly\ normalising$ *because it reduces to the normal form* ω *in two* β*-reduction steps.*

$$((\mathsf{K}\ \ \omega)\ \ \omega) \rightarrow^1_\beta (\lambda y[\omega]\ \ \omega) \rightarrow^1_\beta \omega$$

2. *Consider the term* $((\mathsf{S}\ \ \mathsf{K})\ \ \mathsf{K})$. *Its reduction sequences go as follows:*

$$((\mathsf{S}\ \ \mathsf{K})\ \ \mathsf{K}) \rightarrow^1_\beta \lambda z[((\mathsf{K}\ \ z)\ \ (\mathsf{K}\ \ z))] \rightarrow^1_\beta \lambda z[z] \equiv \mathsf{I}$$

Example 12.22

1. $\Omega \stackrel{df}{=} (\omega\ \ \omega)$ *has no* β*-nf. Hence it is neither* *weakly nor strongly* *normalising.*
2. $(\mathsf{K}\ \ (\omega\ \ \omega))$ *cannot reduce to any normal form because it has no finite reduction sequences. All its reductions are of the form*

$$(\mathsf{K}\ \ (\omega\ \ \omega)) \rightarrow^1_\beta (\mathsf{K}\ \ (\omega\ \ \omega)) \rightarrow^1_\beta (\mathsf{K}\ \ (\omega\ \ \omega)) \rightarrow^1_\beta \cdots$$

or at some point it could transform to

$$(\mathsf{K}\ \ (\omega\ \ \omega)) \rightarrow^1_\beta \lambda y[(\omega\ \ \omega)] \rightarrow^1_\beta \lambda y[(\omega\ \ \omega)] \rightarrow^1_\beta \cdots$$

3. $((\mathsf{K}\ \ \omega)\ \ \Omega)$ *is* $weakly$ *normalising because it can reduce to the normal form* ω *but it is* not $strongly$ *normalising because it also has an infinite reduction sequence*

$$((\mathsf{K}\ \ \omega)\ \ \Omega) \rightarrow^1_\beta ((\mathsf{K}\ \ \omega)\ \ \Omega) \rightarrow^1_\beta \cdots$$

Parameter Passing Mechanisms

- Call-by-name/text defines a *Leftmost-outermost-computation*, i.e. the leftmost-outermost β-redex is chosen for application.
- Call-by-value defines a *Leftmost-innermost-computation*, i.e. the leftmost-innermost β-redex is chosen for application.

To study these computation rules with regard to computing β-normal forms we consider the following examples.

Example 12.23 *Let*

- $L \rightarrow^{l}_{\beta} P \not\rightarrow_{\beta}$ *yield a normal form* P *in* l *steps of* β*-reduction,*
- $M \rightarrow^{m}_{\beta} Q \not\rightarrow_{\beta}$ *yield a normal form* Q *in* m *steps and*
- $N \rightarrow^{n}_{\beta} R \not\rightarrow_{\beta}$ *yield a normal form* R *in* n *steps*

Deterministic Computation Mechanisms: K

Example 12.24 *For the term* $((\mathsf{K}\ \ L)\ \ \Omega)$ *we have the following reduction sequences.*

Call-by-name/text. *Choose the* $\mathit{leftmost\text{-}outermost}$ β*-redex*

- $((\mathsf{K}\ \ L)\ \ \Omega) \rightarrow^1_\beta (\lambda y[L]\ \ \Omega) \rightarrow^1_\beta L \rightarrow^l_\beta P$

Call-by-value. *Choose the* $\mathit{leftmost\text{-}innermost}$ β*-redex*

- $((\mathsf{K}\ \ L)\ \ \Omega) \rightarrow^l_\beta ((\mathsf{K}\ \ P)\ \ \Omega) \rightarrow^1_\beta (\lambda y[P]\ \ \Omega) \rightarrow^*_\beta (\lambda y[P]\ \ \Omega) \rightarrow^*_\beta \cdots$

Here Call-by-value fails to produce the normal form even when it exists.

Example 12.25 *For the term* $(((\mathsf{S}\ \ L)\ \ M)\ \ N)$ *we have the following reduction sequences if* $((P\ \ R)\ \ (Q\ \ R))$ *is in* β *normal form.*

Call-by-name/text. *Choose the* *leftmost-outermost* β*-redex*

- $(((\mathsf{S}\ \ L)\ \ M)\ \ N) \to_\beta^3 ((L\ \ N)\ \ (M\ \ N)) \to_\beta^l ((P\ \ N)\ \ (M\ \ N)) \to_\beta^n ((P\ \ R)\ \ (M\ \ N)) \to_\beta^m ((P\ \ R)\ \ (Q\ \ N)) \to_\beta^n ((P\ \ R)\ \ (Q\ \ R))$

Call-by-value. *Choose the* *leftmost-innermost* β*-redex*

- $(((\mathsf{S}\ \ L)\ \ M)\ \ N) \to_\beta^l (((\mathsf{S}\ \ P)\ \ M)\ \ N) \to_\beta^m (((\mathsf{S}\ \ P)\ \ Q)\ \ N) \to_\beta^n (((\mathsf{S}\ \ P)\ \ Q)\ \ R) \to_\beta^3 ((P\ \ R)\ \ (Q\ \ R))$

Call-by-value takes fewer steps to reduce to the normal form because there is no duplication of the argument N.

Contrariwise

Example 12.26 *However consider the term* $((\mathsf{K}\ \ L)\ \ M)$.

Call-by-name/text. *In* $l+2$ *steps we get the normal form.*

- $((\mathsf{K}\ \ L)\ \ M) \rightarrow^1_\beta (\lambda y[L]\ \ M) \rightarrow^1_\beta L \rightarrow^l_\beta P$

Call-by-value. *We get the normal form in* $l+m+2$ *steps.*

- $((\mathsf{K}\ \ L)\ \ M) \rightarrow^l_\beta ((\mathsf{K}\ \ P)\ \ M) \rightarrow^m_\beta ((\mathsf{K}\ \ P)\ \ Q) \rightarrow^1_\beta (\lambda y[P]\ \ Q) \rightarrow^1_\beta L \rightarrow^l_\beta P$

Here Call-by-value takes an extra m *steps reducing an argument* M *that has no influence on the computation!*

Some Morals, Some Practice

In general,

- Call-by-value (example 12.24) may fail to terminate even if there is a possibility of termination.
- If Call-by-value terminates, then Call-by-name will also terminate. More precisely, if a normal form exists then Call-by-name will definitely find it.
- However, Call-by-value when it does terminate may terminate faster (example 12.25) than Call-by-name/text *provided all arguments need to be evaluated in both cases*.
- It is also easier to implement Call-by-value rather than Call-by-name under static scope rules in the presence of non-local references.

β-nf: Characterisation

The following theorem is easy to prove.

Theorem 12.27 *The class β-nf $\subseteq \Lambda$ is the smallest class such that*

- $V \subseteq \beta$*-nf i.e. all variables are in* β*-nf,*
- *if* $L_1, \ldots, L_m \in \beta$*-nf then for any variable* x, $(x\ L_1\ \ldots\ L_m) \in \beta$*-nf and*
- *if* $L \in \beta$*-nf then* $\lambda x[L] \in \beta$*-nf.*

∎

Notions of Reduction

Reduction

For any function such as $p = \lambda x[3.x.x + 4.x + 1]$,

$$(p \;\; 2) = 3.2.2 + 4.2 + 1 = 21$$

However there is something *asymmetric* about the identity,

- While $(p \;\; 2)$ deterministically produces $3.2.2 + 4.2 + 1$ which in turn
- simplifies deterministically to 21,

Reduction Induced Equality

- It is not possible to deterministically infer that 21 came from $(p\ \ 2)$. It would be more accurate to refer to this sequence as a *reduction sequence* and capture the asymmetry as follows:

$$(p\ \ 2) \rightsquigarrow 3.2.2 + 4.2 + 1 \rightsquigarrow 21$$

- And yet they are *behaviourally* equivalent and mutually substitutable in all contexts (*referentially transparent*).

Reduction Vs. Equality

1. Reduction (specifically β-reduction) captures this asymmetry.
2. Since reduction produces behaviourally $equal$ terms we have the following notion of equality.

Untyped λ-Calculus: β-Equality

Definition 13.1 *β-equality or β-conversion (denoted $=_\beta$) is the smallest equivalence relation containing β-reduction ($\longrightarrow^*_\beta$).*

The following are equivalent definitions.

1. $=_\beta$ is the reflexive-symmetric-transitive closure of 1-step β-reduction.
2. $=_\beta$ is the smallest relation defined by the following rules.

$$=_\beta \textbf{Basis}\ \frac{L \longrightarrow^*_\beta M}{L =_\beta M} \qquad =_\beta \textbf{Reflexivity}\ \frac{}{L =_\beta L}$$

$$=_\beta \textbf{Symmetry}\ \frac{L =_\beta M}{M =_\beta L} \qquad =_\beta \textbf{Transitivity}\ \frac{L =_\beta M,\ \ M =_\beta N}{L =_\beta N}$$

Compatibility of Beta-reduction and Beta-Equality

Theorem 13.2 *β-reduction $\rightarrow_\beta^*$ and β-equality $=_\beta$ are both compatible relations.*

$\square$

Lemma 13.3 (Substitution lemma). *If $L \rightarrow_\beta^* M$ (resp. $L =_\beta M$) then*

1. $x \notin FV(L)$ implies $x \notin FV(M)$,

2. $\{N/x\}L \rightarrow_\beta^* \{N/x\}M$ (resp. $\{N/x\}L =_\beta \{N/x\}M$),

3. $\{L/x\}N \rightarrow_\beta^* \{M/x\}N$ (resp. $\{L/x\}N =_\beta \{M/x\}N$).

Proof of theorem 13.2

Proof: ($\to_\beta^*$) Assume $L \to_\beta^* M$. By definition of β-reduction $L \to_\beta^n M$ for some $n \geq 0$. The proof proceeds by induction on n

Basis. $n = 0$. Then $L \equiv M$ and there is nothing to prove.

Induction Hypothesis (IH).

The proof holds for all k, $0 \leq k \leq m$ *for some* $m \geq 0$.

Induction Step. For $n = m+1$, let $L \equiv L_0 \to_\beta^m L_m \to_\beta^1 M$. Then by the induction hypothesis and the compatibility of $\to_\beta^1$ we have

$$\begin{array}{llll|l} & & & & \text{By definition of } \to_\beta^n \\ \text{for all } x \in V, & \lambda x[L] \to_\beta^m \lambda x[L_m], & \lambda x[L_m] \to_\beta^1 \lambda x[M] & & \lambda x[L] \to_\beta^n \lambda x[M], \\ \text{for all } N \in \Lambda, & (L\ \ N) \to_\beta^m (L_m\ \ N), & (L_m\ \ N) \to_\beta^1 (M\ \ N) & & (L\ \ N) \to_\beta^n (M\ \ N) \\ \text{for all } N \in \Lambda, & (N\ \ L) \to_\beta^m (N\ \ L_m), & (N\ \ L_m) \to_\beta^1 (N\ \ M) & & (N\ \ L) \to_\beta^n (N\ \ M) \end{array}$$

End ($\to_\beta^*$)

($=_\beta$) Assume $L =_\beta M$. We proceed by induction on the length of the proof of $L =_\beta M$ using the definition of β-equality.

Basis. $n = 1$. Then either $L \equiv M$ or $L \to_\beta^* M$. The case of reflexivity is trivial and the case of $L \to_\beta^* M$ follows from the previous proof.

Induction Hypothesis (IH).

> *For all terms L and M, such that the proof of $L =_\beta M$ requires less than n steps for $n > 1$, the compatibility result holds.*

Induction Step. Suppose the proof requires n steps and the last step is obtained by use of either $=_\beta$ **Symmetry** or $=_\beta$ **Transitivity** on some previous steps.

Case ($=_\beta$ **Symmetry).** Then the $(n-1)$-st step proved $M =_\beta L$. By the induction hypothesis and then by applying $=_\beta$ **Symmetry** to each case we get

$$\begin{array}{lc|l} & & \text{By } =_\beta \textbf{ Symmetry} \\ \text{for all variables } x, & \lambda x[M] =_\beta \lambda x[L] & \lambda x[L] =_\beta \lambda x[M] \\ \text{for all terms } N, & (M\ \ N) =_\beta (L\ \ N) & (L\ \ N) =_\beta (M\ \ N) \\ \text{for all terms } N, & (N\ \ M) =_\beta (N\ \ L) & (N\ \ M) =_\beta (N\ \ L) \end{array}$$

Case ($=_\beta$ **Transitivity).** Suppose $L =_\beta M$ was inferred in the n-th step from two previous steps which proved $L =_\beta P$ and $P =_\beta M$ for some term P. Then again by induction hypothesis and then applying $=_\beta$ **Transitivity** we get

$$\begin{array}{lcc|l} & & & \text{By } =_\beta \textbf{ Transitivity} \\ \text{for all variables } x, & \lambda x[L] =_\beta \lambda x[P], & \lambda x[P] =_\beta \lambda x[M] & \lambda x[L] =_\beta \lambda x[M] \\ \text{for all terms } N, & (L\ \ N) =_\beta (P\ \ N), & (P\ \ N) =_\beta (M\ \ N) & (L\ \ N) =_\beta (M\ \ N) \\ \text{for all terms } N, & (N\ \ L) =_\beta (N\ \ P), & (N\ \ P) =_\beta (N\ \ M) & (N\ \ L) =_\beta (N\ \ P) \end{array}$$

End ($=_\beta$)

QED

Eta reduction

Given any term M and a variable $x \notin FV(M)$, the syntax allows us to construct the term $\lambda x[(M \;\; x)]$ such that for every term N we have

$$(\lambda x[(M \;\; x)] \;\; N) \rightarrow^1_\beta (M \;\; N)$$

In other words,

$$\boxed{(\lambda x[(M \;\; x)] \;\; N) =_\beta (M \;\; N) \text{ for all terms } N}$$

We say that the two terms $\lambda x[(M \;\; x)]$ and M are **extensionally** equivalent i.e. they are *syntactically distinct* but there is no way to distinguish between their *behaviours*.

So we define basic η-reduction as the relation

$$\boxed{\lambda x[(L \;\; x)] \rightarrow_\eta L \text{ provided } x \notin FV(L)} \qquad (7)$$

Eta-Reduction and Eta-Equality

The following notions are then defined similar to the corresponding notions for β-reduction.

- 1-step η-reduction $\rightarrow^1_\eta$ is the closure of basic η-reduction to all contexts,
- $\rightarrow^n_\eta$ is defined by induction on 1-step η-reduction
- η-reduction $\rightarrow^*_\eta$ is the reflexive-transitive closure of 1-step η-reduction.
- the notions of strong and weak η normal forms η-nf.
- the notion of η-equality or η-conversion denoted by $=_\eta$.

Example 13.4 *Consider Curry's* $paradoxical$ *combinator*

$$\mathsf{Y_C} \stackrel{df}{=} \lambda f[(\mathsf{C}\ \ \mathsf{C})] \text{ where } \mathsf{C} \stackrel{df}{=} \lambda x[(f\ \ (x\ \ x))]$$

For any term L *we have*

$$\begin{aligned}
(\mathsf{Y_C}\ \ L) &\rightarrow^1_\beta (\lambda x[(L\ \ (x\ \ x))]\ \ \lambda x[(L\ \ (x\ \ x))]) \\
&\equiv_\alpha (\lambda y[(L\ \ (y\ \ y))]\ \ \lambda x[(L\ \ (x\ \ x))]) \\
&\rightarrow^1_\beta (L\ \ \underbrace{(\lambda x[(L\ \ (x\ \ x))]\ \ \lambda x[(L\ \ (x\ \ x))])}) \\
&=_\beta (L\ \ \overbrace{(\mathsf{Y_C}\ \ \mathsf{L})})
\end{aligned}$$

Hence $\boxed{(\mathsf{Y_C}\ \ L) =_\beta (L\ \ (\mathsf{Y_C}\ \ L))}$. *However* $(L\ \ (\mathsf{Y_C}\ \ L))$ *will never* β*-reduce to* $(\mathsf{Y_C}\ \ L)$.

Exercise 13.1

1. *Prove that η-reduction and η-equality are both compatible relations.*
2. *Prove that η-reduction is strongly normalising.*
3. *Define basic $\beta\eta$-reduction as the application of either (6) or (7). Now prove that $\rightarrow^1_{\beta\eta}$, $\rightarrow^*_{\beta\eta}$ and $=_{\beta\eta}$ are all compatible relations.*

13.1. Recursion and the Y combinator

Since the lambda calculus only has variables and expressions and there is no place for names themselves (we use names such as K and S for our convenience in discourse, but the language itself allows only (untyped) variables and is meant to define functions anonymously as expressions in the language). In such a situation, recursion poses a problem in the language.

Recursion in most programming languages requires the use of an identifier which names an expression that contains a call to the very name of the function that it is supposed to define. This is at variance with the aim of the lambda calculus wherein the only names belong to variables and even functions may be defined anonymously as mere expressions.

This notion of recursive definitions may be generalised to a system of mutually recursive definitions.

The name of a recursive function, acts as a place holder in the body of the definition (which in turn has the name acting as a place holder for a copy of the body of the definition and so on ad infinitum). However no language can have sentences of infinite length.

The combinator $\mathsf{Y_C}$ helps in providing copies of any lambda term L whenever demanded in a more disciplined fashion. This helps in the modelling of recursive definitions anonymously. What the $\mathsf{Y_C}$ combinator provides is a mechanism for recursion "unfolding" which is precisely our understanding of how recursion should work. Hence it is easy to see from $\boxed{(\mathsf{Y_C}\ \ L) =_\beta (L\ \ (\mathsf{Y_C}\ \ L))}$ that

$$(\mathsf{Y_C}\ \ L) =_\beta (L\ \ (\mathsf{Y_C}\ \ L)) =_\beta (L\ \ (L\ \ (\mathsf{Y_C}\ \ L))) =_\beta (L\ \ (L\ \ (L\ \ (\mathsf{Y_C}\ \ L)))) =_\beta \cdots \tag{8}$$

Many other researchers have defined other combinators which mimic the behaviour of the combinator $\mathsf{Y_C}$. Of particular interest is Turing's combinator $\mathsf{Y_T} \stackrel{df}{=} (\mathsf{T}\ \ \mathsf{T})$ where $\mathsf{T} \stackrel{df}{=} \lambda x\ y[(y\ \ ((x\ \ x)\ \ y))]$. Notice that

$$
\begin{array}{rl}
 & (\mathsf{T}\ \ \mathsf{T}) \\
\equiv & (\lambda x\ y[(y\ \ ((x\ \ x)\ \ y))]\ \ \mathsf{T}) \\
\to^1_\beta & \lambda y[(y\ \ ((\mathsf{T}\ \ \mathsf{T})\ \ y))] \\
\equiv & \lambda y[(y\ \ (\mathsf{Y_T}\ \ \mathsf{y}))]
\end{array}
$$

from which, by compatible closure, for any term L we get

$$
\begin{array}{rl}
 & (\mathsf{Y_T}\ \ L) \\
\equiv & ((\mathsf{T}\ \ \mathsf{T})\ \ L) \\
\to^*_\beta & (\lambda y[(y\ \ (\mathsf{Y_T}\ \ y))]\ \ L) \\
\to^1_\beta & (L\ \ (\mathsf{Y_T}\ \ L))
\end{array}
$$

Thus $\mathsf{Y_T}$ is also a recursion unfolding combinator yielding

$$
(\mathsf{Y_T}\ \ L) =_\beta (L\ \ (\mathsf{Y_T}\ \ L)) =_\beta (L\ \ (L\ \ (\mathsf{Y_T}\ \ L))) =_\beta (L\ \ (L\ \ (L\ \ (\mathsf{Y_T}\ \ L)))) =_\beta \cdots
$$

Recursion and The Fixed point theorem

Theorem 13.5 *For every (untyped) λ-term L, there exists a fixed point F_L such that $F_L =_\beta (L\ F_L)$.*

Proof: Assume x and y are not free in L. Then

$$\begin{array}{rcl} F_L & \equiv_\alpha & (\lambda\ x[(L\ (x\ x))]\ \lambda\ x[(L\ (x\ x))]) \\ & \equiv_\alpha & (\lambda\ x[(L\ (x\ x))]\ \lambda\ y[(L\ (y\ y))]) \\ & \longrightarrow^1_\beta & (L\ (\lambda\ y[(L\ (y\ y))]\ \lambda\ y[(L\ (y\ y))])) \\ & \equiv_\alpha & (L\ F_L) \end{array}$$

Hence $F_L =_\beta (L\ F_L)$. QED

By abstracting out L we get a function that can generate fixed-points.

Corollary 13.6 $\mathsf{Y_C}$ *is a fixed-point combinator which generates a fixed point for every lambda term i.e* $(\mathsf{Y_C}\ L) \longrightarrow^1_\beta F_L$.

14. Representing Data in the Untyped Lambda Calculus

The Boolean Constants

$$\mathsf{True} \stackrel{df}{=} \lambda x[\lambda y[x]] \qquad \text{(True)}$$

$$\mathsf{False} \stackrel{df}{=} \lambda x[\lambda y[y]] \qquad \text{(False)}$$

Negation

$$\mathsf{Not} \stackrel{df}{=} \lambda x[((x \;\; \mathsf{False}) \;\; \mathsf{True})] \qquad \text{(not)}$$

The Conditional

$$\mathsf{Ite} \stackrel{df}{=} \lambda x\ y\ z[(x\ y\ \ z)] \tag{ite}$$

Exercise 14.1

1. *Prove that*

$$(\mathsf{Not}\ \ \mathsf{True}) =_{\beta\eta} \mathsf{False} \quad (9)$$

$$(\mathsf{Not}\ \ \mathsf{False}) =_{\beta\eta} \mathsf{True} \quad (10)$$

2. *Prove that*

$$(\mathsf{Ite}\ \ \mathsf{True}\ L\ \ M) =_{\beta\eta} L \quad (11)$$

$$(\mathsf{Ite}\ \ \mathsf{False}\ L\ \ M) =_{\beta\eta} M \quad (12)$$

$$(13)$$

3. *We know from Theorem 7.7 that the boolean constants and the conditional form a functionally complete (adequate) set for propositional logic. Use the conditional combinator* Ite *and the constant combinators* True *and* False *to express the following boolean operators upto $\beta\eta$-equivalence.*
   - Not. *Verify that it is α-equivalent to (not).*
   - And*: conjunction*
   - Or*: disjunction*

- Xor: *exclusive OR*

4. *Prove the de Morgan laws for the boolean combinators, using only $\beta\eta$-reductions.*
5. *Does* ((And K) I) *have a $\beta\eta$-normal form?*

The Church Numerals There are many ways to represent the natural numbers as lambda expressions. Here we present Church's original encoding of the naturals in the λ-calculus. We represent a natural n as a combinator n.

$$\mathsf{0} \stackrel{df}{=} \lambda f\ x[x] \qquad \text{(numeral-0)}$$

$$\mathsf{1} \stackrel{df}{=} \lambda f\ x[(f\ \ x)] \qquad \text{(numeral-1)}$$

$$\ldots$$

$$\mathsf{n}+\mathsf{1} \stackrel{df}{=} \lambda f\ x[(f\ \ (f^n\ \ x))] \qquad \text{(numeral-n+1)}$$

$$\ldots$$

where $(f^n\ \ x)$ denotes the n-fold application of f to x. That is, $(f^n\ \ x) = \underbrace{(f\ (f\ \ \ldots\ \ (f\ x)\ldots))}_{f\ \text{applied}\ n\ \text{times}}$.

"Arithmagic"

For any function g and Church numeral n, $(\mathsf{n}\ g)$ β-reduces to $\lambda x[(g^n\ \ x)]$ which is the n-fold application of g.

We follow the operators of Peano arithmetic and the postulates of first order arithmetic (as treated in any course in first order logic) and obtain "magically"[6] the following combinators for the basic operations of arithmetic and checking for 0.

$$\mathsf{IsZero} \stackrel{df}{=} \lambda n[(n\ \lambda x[\mathsf{False}]\ \mathsf{True})] \tag{IsZero}$$

$$\mathsf{Succ} \stackrel{df}{=} \lambda n\ f\ x[((n\ \ f)\ \ (f\ \ x))] \tag{Succ}$$

$$\mathsf{Add} \stackrel{df}{=} \lambda m\ n\ f\ x[((m\ \ f)\ \ (n\ f\ x))] \tag{Add}$$

$$\mathsf{Mult} \stackrel{df}{=} \lambda m\ n\ f[(m\ \ (n\ \ f))] \tag{Mult}$$

$$\mathsf{Pwr} \stackrel{df}{=} \lambda m\ n[(n\ \ m)] \tag{Pwr}$$

The only way to convince oneself that the above are correct, is to verify that they do produce the expected results.

[6]There are geniuses out there somewhere who manage to come up with these things. Don't ask me how they thought of them!

Exercise 14.2

1. *The successor function may also be defined as* $\mathsf{Succ}' \stackrel{df}{=} \lambda n\ f\ x[(f\ \ (n\ f\ \ x))]$. *Show that the following hold when* Succ *is replaced by* Succ'
2. *Prove the following.*
   (a) $(\mathsf{Succ}\ 0) =_{\beta\eta} 1$
   (b) $(\mathsf{Succ}\ \mathsf{n}) =_{\beta\eta} \mathsf{n}+1$
   (c) $(\mathsf{IsZero}\ 0) =_{\beta\eta} \mathsf{True}$
   (d) $(\mathsf{IsZero}\ (\mathsf{Succ}\ \ \mathsf{n})) =_{\beta\eta} \mathsf{False}$
   (e) $(\mathsf{Add}\ 0\ \mathsf{n}) =_{\beta\eta} \mathsf{n}$
   (f) $(\mathsf{Add}\ \mathsf{m}\ 0) =_{\beta\eta} \mathsf{m}$
   (g) $(\mathsf{Add}\ \mathsf{m}\ \mathsf{n}) =_{\beta\eta} \mathsf{p}$ *where* p *denotes the combinator for* $p = m + n$
3. *Try to reduce* $(\mathsf{Add}\ \mathsf{K}\ \mathsf{S})$ *to its* β*-normal form. Can you interpret the resulting lambda term as representing some meaningful function?*
4. *What identities should* Mult *and* Pwr *satisfy? Do they do indeed satisfy the inductive definitions of multiplication and powering of natural numbers respectively. In particular, what is* $(\mathsf{Pwr}\ 0\ 0)$*?*

Ordered Pairs and Tuples

$$\mathsf{Pair} \stackrel{df}{=} \lambda x\ y\ p[(p\ x\ y)] \tag{14}$$

$$\mathsf{Fst} \stackrel{df}{=} \lambda p[(p\ \mathsf{True})] \tag{15}$$

$$\mathsf{Snd} \stackrel{df}{=} \lambda p[(p\ \mathsf{False})] \tag{16}$$

We may define an n-tuple inductively as a pair consisting of the first element of the n-tuple and an $n-1$ tuple of the other $n-1$ elements. Let $\langle L, M\rangle$ represent a pair. We then have for any $n > 2$

$$\langle L_1, \ldots, L_n\rangle = (\mathsf{Pair}\ L_1\ \langle L_2, \ldots, L_n\rangle)$$

Recursively defined data structures – Lists

Note the isomorphism between lists of length n and n-tuples for each $n \geq 2$ (ordered pairs are 2-tuples). We use this facility to define lists of length $n \geq 0$ by first defining the empty list as being the same as False.

$$\mathsf{Nil} \stackrel{df}{=} \lambda x\ y[y] \tag{17}$$

$$\mathsf{List} \stackrel{df}{=} \lambda h\ t[\mathsf{Pair}\ h\ t] \tag{18}$$

$$\mathsf{Hd} \stackrel{df}{=} \lambda l[(l\ \mathsf{True})] \tag{19}$$

$$\mathsf{Tl} \stackrel{df}{=} \lambda l[(l\ \mathsf{False})] \tag{20}$$

Exercise 14.3

1. *Let* $P \stackrel{df}{=} (\mathsf{Pair}\ L\ M)$. *Verify that* $(\mathsf{Pair}\ (\mathsf{Fst}\ P)\ (\mathsf{Snd}\ P)) =_{\beta\eta} P$.
2. *Let* $\mathsf{Sfst} \stackrel{df}{=} (\mathsf{Fst}\ \mathsf{S})$ *and* $\mathsf{Ssnd} \stackrel{df}{=} (\mathsf{Snd}\ \mathsf{S})$.
   (a) *Compute the* $\beta\eta$ *normal form of* $(\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd})$*? Is it* $\beta\eta$*-equal to* S*?*
   (b) *Now compute the* $\beta\eta$ *normal forms of* $(\mathsf{Fst}\ \ (\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd}))$ *and* $(\mathsf{Snd}\ \ (\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd}))$. *What are their* $\beta\eta$ *normal forms?*
   (c) *What can you conclude from the above?*
3. *For any* k, $0 \leq k < n$, *define combinators which extract the* k*-th component of an* n*-tuple.*
4. (a) *Define a combinator* $\mathsf{Bintree}$ *that constructs binary trees from* λ*-terms with node labels drawn from the Church numerals.*
   (b) *Define combinators* Root, Lst *and* Rst *which yield respectively the root, the left subtree and the right subtree of a binary tree.*
   (c) *Prove that for any such binary tree* B *expressed as a* λ*-term,* $(\mathsf{Bintree}\ (\mathsf{Root}\ \ B)\ (\mathsf{Lst}\ \ B)\ (\mathsf{Rst}\ \ B)) =_{\beta\eta} B$.

Confluence: Definitions

Reduction Relations

Definition 15.1 *For any binary relation ρ on Λ*

1. ρ^1 *is the compatible closure of* ρ
2. ρ^+ *is the transitive closure of* ρ^1
3. ρ^* *is the reflexive-transitive-closure of* ρ^1 *and is a preorder*
4. $((\rho^1) \cup (\rho^1)^{-1})^*$ *(denoted* $=_\rho$*) is the reflexive-symmetric-transitive closure of* ρ^1 *and is an equivalence relation.*
5. $=_\rho$ *is also called the equivalence* **generated** *by* ρ*.*

Reduction Relations: Arrow Notation

We will often use $\longrightarrow$ (suitably decorated) in infix notation as a reduction relation instead of ρ. Then

- $\longrightarrow^1$ denotes the *compatible* closure of $\longrightarrow$,
- $\longrightarrow^+$ denotes the *transitive* closure of $\longrightarrow$,
- $\longrightarrow^*$ denotes the *reflexive-transitive* closure of $\longrightarrow$, and
- $\stackrel{*}{\longleftrightarrow}$ denotes the *equivalence generated* by $\longrightarrow$,

The Diamond Property

Definition 15.2 *Let ρ be any relation on terms. ρ has the* **diamond property** *if for all L, M, N,*

$$\begin{tikzcd} & M & & M \arrow[dr, "\rho"] & \\ L \arrow[ur, "\rho"] \arrow[dr, "\rho"'] & & \Rightarrow \exists P : & & P \\ & N & & N \arrow[ur, "\rho"'] & \end{tikzcd}$$

The Diamond Property: Arrow Notation

We often use a decorated version of the symbol $\longrightarrow$ for a reduction relation and depict the diamond property as

$$\begin{tikzcd} & M \arrow[dr] & \\ L \arrow[ur] \arrow[dr] & & \Rightarrow \exists\ P \\ & N \arrow[ur] & \end{tikzcd}$$

Reduction Relations: Termination

Let $\longrightarrow$ be a reduction relation, $\longrightarrow^*$ the least preorder containing $\longrightarrow$ and $\stackrel{*}{\longleftrightarrow}$ the least equivalence relation containing $\longrightarrow^*$. Then

Definition 15.3 $\longrightarrow$ *is* $\mathbf{terminating}$ *iff there is no infinite sequence of the form*

$$L_0 \longrightarrow L_1 \longrightarrow \cdots$$

Lemma 15.4 $\longrightarrow_\eta$ *is a terminating reduction relation.*

Proof: By induction on the structure of terms. QED

15.1. Why confluence?

We are mostly interested in β-reduction which is not guaranteed to terminate. We already know that there are several terms which are only weakly normalising (β-WN). This means that there are several possible reduction sequences, some of which may yield β-normal forms while the others may yield infinite computations. Hence in order to obtain normal forms for such terms we need to schedule the β-reductions carefully to be guaranteed a normal form. The matter would be further complicated if there are multiple unrelated normal forms.

Each β-reduction step may reveal fresh β-redexes. This in turn raises the disquieting possibility that each termination sequence may yield a different β-normal form. If such is indeed the case, then it raises fundamental questions on the use of β-reduction (or function application) as a notion of reduction. If β-reduction is to be considered fundamental to the notion of computation then all β-reduction sequences that terminate in β-nfs must yield the same β-nf upto α-equivalence.

Hence our interest in the notion of confluence. Since the issue of confluence of β-reduction is rather complicated we approach it in terms of inductively easier notions such as *local confluence*, and *semi-confluence* which finally lead up to *confluence* and the Church-Rosser property.

Reduction: Local Confluence

Definition 15.5 $\longrightarrow$ *is* **locally confluent** *if for all* L, M, N,

$$N \longleftarrow L \longrightarrow M \Rightarrow \exists P : N \longrightarrow^* P \ {}^*\!\!\longleftarrow M$$

which we denote by

$$\begin{tikzcd}
 & M \arrow[dr, "*"] & \\
L \arrow[ur] \arrow[dr] & & \Rightarrow \exists \ P \\
 & N \arrow[ur, "*"'] &
\end{tikzcd}$$

Reduction: Semi-confluence

Definition 15.6 $\longrightarrow$ *is* **semi-confluent** *if for all* L, M, N,

$$N \longleftarrow L \longrightarrow^{*} M \Rightarrow \exists P : N \longrightarrow^{*} P \;{}^{*}\!\!\longleftarrow M$$

which we denote by

$$\begin{tikzcd}
 & M \arrow[dr, "*"] & \\
L \arrow[ur] \arrow[dr, "*"'] & & \Rightarrow \exists\ P \\
 & N \arrow[ur, "*"'] &
\end{tikzcd}$$

Reduction: Confluence

Definition 15.7 $\longrightarrow$ *is* confluent *if for all* L, M, N,

$$N \;{}^*\!\!\longleftarrow L \longrightarrow^* M \Rightarrow \exists P : N \longrightarrow^* P \;{}^*\!\!\longleftarrow M$$

which we denote as

$$\begin{tikzcd}
 & M \arrow[dr, "*"] & \\
L \arrow[ur, "*"] \arrow[dr, "*"'] & & \Rightarrow \exists \; P \\
 & N \arrow[ur, "*"'] &
\end{tikzcd}$$

Fact 15.8 *Any confluent relation is also semi-confluent.*

Equivalence Characterization

Lemma 15.9

1. $\stackrel{*}{\longleftrightarrow}$ *is the least equivalence containing* $\longrightarrow$.

2. $\stackrel{*}{\longleftrightarrow}$ *is the least equivalence containing* $\longrightarrow^*$.

3. $L \stackrel{*}{\longleftrightarrow} M$ *if and only if there exists a finite sequence* $L \equiv M_0, M_1, \ldots M_m \equiv M$, $m \geq 0$ *such that for each* i, $0 \leq i < m$, $M_i \longrightarrow M_{i+1}$ *or* $M_{i+1} \longrightarrow M_i$. *We represent this fact more succinctly as*

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_\alpha M \qquad (21)$$

Proof of lemma 15.9

Proof:

1. Just prove that $\stackrel{*}{\longleftrightarrow}$ is a subset of every equivalence that contains $\longrightarrow$.
2. Use induction on the length of proofs to prove this part
3. For the last part it is easy to see that the existence of the "chain equation" (21) implies $L \stackrel{*}{\longleftrightarrow} M$ by transitivity. For the other part use induction on the length of the proof.

QED

Reduction: The Church-Rosser Property (CRP)

Definition 15.10 $\longrightarrow$ *is* **Church-Rosser** *if for all* L, M,

$$L \stackrel{*}{\longleftrightarrow} M \Rightarrow \exists P : L \longrightarrow^{*} P \;{}^{*}\!\!\longleftarrow M$$

which we denote by

$$\begin{array}{ccccc} L & & \stackrel{*}{\longleftrightarrow} & & M \\ & \searrow_{*} & \Downarrow & \swarrow_{*} & \\ & & \exists P & & \end{array}$$

To answer the main question we need to prove that β-reduction is Church-Rosser.

β-reduction and CRP

We already know that

- some terms may only be weakly normalising
- weakly normalising terms have both terminating and non-terminating computations.
- But if the CRP holds then all terminating computations will yield the same β-nf (upto $\equiv_\alpha$).

The Church-Rosser Property

Confluence and Church-Rosser

Lemma 15.11 *Every confluent relation is also semi-confluent*

$\blacksquare$

Theorem 15.12 *The following statements are equivalent for any reduction relation* $\longrightarrow$.

1. $\longrightarrow$ *is Church-Rosser.*
2. $\longrightarrow$ *is confluent.*

Proof of theorem 15.12

Proof: $(1 \Rightarrow 2)$ Assume $\longrightarrow$ is Church-Rosser and let

$$N \;{}^*\!\!\longleftarrow L \longrightarrow^* M$$

Clearly then $N \stackrel{*}{\longleftrightarrow} M$. If $\longrightarrow$ is Church-Rosser then

$$\exists P : N \longrightarrow^* P \;{}^*\!\!\longleftarrow M$$

which implies that it is confluent.

$(2 \Rightarrow 1)$ Assume $\longrightarrow$ is confluent and let $L \stackrel{*}{\longleftrightarrow} M$. We proceed by induction on the length of the chain (21).

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_\alpha M$$

Basis. $m = 0$. This case is trivial since for any P, $L \longrightarrow^* P$ iff $M \longrightarrow^* P$

Induction Hypothesis (IH).

> *The claim is true for all chains of length k, $0 \leq k < m$.*

Induction Step. Assume the chain is of length $m = k + 1$. i.e.

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_k \longrightarrow / \longleftarrow M_{k+1} \equiv_\alpha M$$

Case $M_k \longrightarrow M$. Then by the induction hypothesis and semi-confluence we have

$$\begin{tikzcd}
L \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & \arrow[d, phantom, "\Downarrow"] & M_k \arrow[dl, "*"] \arrow[dr] & \\
& \exists Q \arrow[dr, "*"'] & \arrow[d, phantom, "\Downarrow"] & M \arrow[dl, "*"] \\
& & \exists P &
\end{tikzcd}$$

which proves the claim.

Case $M_k \longleftarrow M$. Then the claim follows from the induction hypothesis and the following diagram

$$\begin{tikzcd}
L \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & \arrow[d, phantom, "\Downarrow"] & M_k \arrow[dl, "*"] & M \arrow[l] \\
& \exists P & &
\end{tikzcd}$$

QED

Lemma 15.13 *If a terminating relation is locally confluent then it is semi-confluent.*

Proof: Assume $L \longrightarrow M$ and $L \longrightarrow^* N$. We need to show that there exists P such that $M \longrightarrow^* P$ and $N \longrightarrow^* P$. We prove this by induction on the length of $L \longrightarrow^* N$. If $L \equiv_\alpha N$ then $P \equiv_\alpha M$, otherwise assume $L \longrightarrow N_1 \longrightarrow \cdots \longrightarrow N_n = N$ for some $n > 0$. By the local confluence we have there exists P_1 such that $M \longrightarrow^* P_1$. By successively applying the induction hypothesis we get terms $P_2, \ldots, P_n$ such that $P_{j-1} \longrightarrow^* P_j$ and $N_j \longrightarrow^* P_j$ for each j, $1 \leq j \leq m$. In effect we complete the following rectangle

$$\begin{array}{ccccccccc}
L & \longrightarrow & N_1 & \longrightarrow & N_2 & \longrightarrow & \cdots & \longrightarrow & N_n \equiv M \\
\downarrow & & \downarrow & & \downarrow & & \cdots & & \downarrow \\
M & \longrightarrow & P_1 & \longrightarrow & P_2 & \longrightarrow & \cdots & \longrightarrow & P_n
\end{array}$$

QED

From lemma 15.13 and theorem 15.12 we have the following theorem.

Theorem 15.14 *If a terminating relation is locally confluent then it is confluent.*

Proof:

$\longrightarrow$ on Λ is given to be terminating and locally confluent. We need to show that it is confluent. That is for any L, we are given that

1. there is no infinite sequence of reductions of L, i.e. every maximal sequence of reductions of L is of length n for some $n \geq 0$.
2. $$N_1 \;{}^1\!\!\longleftarrow L \longrightarrow^1 M_1 \Rightarrow \exists P : M_1 \longrightarrow^* P \;{}^*\!\!\longleftarrow N_1 \tag{22}$$

We need to show for any term L that

$$N \;{}^*\!\!\longleftarrow L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S \;{}^*\!\!\longleftarrow N \tag{23}$$

Let L be any term. Consider the graph $G(L) = \langle \Gamma(L), \longrightarrow^1 \rangle$ such that $\Gamma(L) = \{M \mid L \longrightarrow^* M\}$. Since $\longrightarrow$ is a terminating reduction

Fact 15.15 *The graph $G(L)$ is acyclic for any term L.*

If $G(L)$ is not acyclic, there must be a cycle of length $k > 0$ such that $M_0 \longrightarrow^1 M_1 \longrightarrow^1 \cdots \longrightarrow^1 M_{k-1} \longrightarrow^1 M_0$ which implies there is also an infinite reduction sequence of the form $L \longrightarrow^* M_0 \longrightarrow^k M_0 \longrightarrow^k \cdots$ which is impossible.

Since there are only a finite number of sub-terms of L that may be reduced under $\longrightarrow$, for each L there is a maximum number $p \geq 0$, which is the length of the longest reduction sequence.

Fact 15.16 *For every $M \in \Gamma(L)$,*

1. *$G(M)$ is a sub-graph of $G(L)$ and*
2. *For every $M \in \Gamma(L) - \{L\}$, the length of the longest reduction sequence of M is less than p.*

Proof: We proceed by induction on p.

Basis. $p = 0$. Then $\Gamma(L) = \{L\}$ and there are no reductions possible, so it is trivially confluent.

Induction Hypothesis (IH).

For any L whose longest reduction sequence is of length k, $0 \leq k < p$, property (23) holds.

Induction Step. Assume L is a term whose longest reduction sequence is of length $p > 0$. Also assume $N \;{}^{*}\!\!\longleftarrow L \longrightarrow^{*} M$ i.e. $\exists m, n \geq 0 : N \;{}^{n}\!\!\longleftarrow L \longrightarrow^{m} M$.

Case $m = 0$. If $m = 0$ then $M \equiv_\alpha L$ and hence $S \equiv_\alpha N$.

Case $n = 0$. Then $N \equiv_\alpha L$ and we have $S \equiv_\alpha M$.

Case $m, n > 0$. Then consider M_1 and N_1 such that

$$N \;{}^{*}\!\!\longleftarrow N_1 \;{}^{1}\!\!\longleftarrow L \longrightarrow^{1} M_1 \longrightarrow^{*} M \tag{24}$$

See figure (7). By (22), $\exists P : M_1 \longrightarrow^{*} P \;{}^{*}\!\!\longleftarrow N_1$. Clearly $M_1, N_1, P \in \Gamma(L) - \{L\}$. Hence by fact 15.16, $G(M_1)$, $G(N_1)$ and $G(P)$ are all sub-graphs of $G(L)$ and all their reduction sequences are of length smaller than p. Hence by induction hypothesis, we get

$$P \;{}^{*}\!\!\longleftarrow M_1 \longrightarrow^{*} M \Rightarrow \exists Q : M \longrightarrow^{*} Q \;{}^{*}\!\!\longleftarrow P \tag{25}$$

and

$$N \,{}^*\!\longleftarrow N_1 \longrightarrow^* P \Rightarrow \exists R : P \longrightarrow^* R \,{}^*\!\longleftarrow N \tag{26}$$

But by (25) and (26) and the induction hypothesis we have

$$R \,{}^*\!\longleftarrow P \longrightarrow^* Q \Rightarrow \exists S : Q \longrightarrow^* S \,{}^*\!\longleftarrow R \tag{27}$$

Combining (27) with (24), (25) and (26) we get

$$N \,{}^*\!\longleftarrow L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S \,{}^*\!\longleftarrow N \tag{28}$$

QED

QED

Theorem 15.17 *If a terminating relation is locally confluent then it is Church-Rosser.*

Proof: Follows from theorem 15.14 and theorem 15.12 QED

The Church-Rosser Property for β-reduction

Parallel Beta Reduction

Definition 15.18 *The* **parallel-**β *or* $||\beta$ *reduction is the smallest relation for which the following rules hold.*

$$||\beta_1 \quad \frac{}{L \longrightarrow^1_{||\beta} L}$$

$$||\beta_1\mathbf{App} \quad \frac{L \longrightarrow^1_{||\beta} L', M \longrightarrow^1_{||\beta} M'}{(L \;\; M) \longrightarrow^1_{||\beta} (L' \;\; M')}$$

$$||\beta_1\mathbf{Abs1} \quad \frac{L \longrightarrow^1_{||\beta} L'}{\lambda x[L] \longrightarrow^1_{||\beta} \lambda x[L']}$$

$$||\beta_1\mathbf{Abs2} \quad \frac{L \longrightarrow^1_{||\beta} L', M \longrightarrow^1_{||\beta} M'}{(\lambda x[L] \;\; M) \longrightarrow^1_{||\beta} \{M'/x\}L'}$$

Lemma 15.19

1. $L \longrightarrow^1_\beta L' \Rightarrow L \longrightarrow^1_{||\beta} L'$.
2. $L \longrightarrow^1_{||\beta} L' \Rightarrow L \longrightarrow^*_\beta L'$.
3. $\longrightarrow^*_{||\beta} \quad = \quad \longrightarrow^*_\beta$ *and is the smallest preorder containing* $\longrightarrow^1_{||\beta}$.
4. *If* $L \longrightarrow^1_\beta L'$ *and* $M \longrightarrow^1_{||\beta} M'$ *then* $\{M/x\}L \longrightarrow^1_{||\beta} \{M'/x\}L'$.

Proof: By induction on the structure of terms or by induction on the number of steps in any proof. QED

Theorem 15.20 $\longrightarrow^1_{||\beta}$ *has the diamond property.*

Proof of theorem 15.20

Proof: We need to prove for all L

$$N \;{}^{1}_{||\beta}\!\!\longleftarrow L \longrightarrow^{1}_{||\beta} M \Rightarrow \exists P : N \longrightarrow^{1}_{||\beta} P \;{}^{1}_{||\beta}\!\!\longleftarrow M$$

We prove this by induction on the structure of L and a case analysis of the rule applied in definition 15.18.

Case $L \equiv x \in V$. Then $L \equiv M \equiv N \equiv P$.

Before dealing with the other inductive cases we dispose of some trivial sub-cases that arise in some or all of them.

Case $L \equiv_\alpha M$. Choose $P \equiv_\alpha N$ to complete the diamond.

Case $L \equiv_\alpha N$. Then choose $P \equiv_\alpha M$.

Case $M \equiv_\alpha N$. Then there is nothing to prove.

In the sequel we assume $N \not\equiv_\alpha L \not\equiv_\alpha M \not\equiv_\alpha N$ and proceed by induction on the structure of L.

Case $L \equiv \lambda x[L_1]$. Then clearly M and N were both obtained in proofs whose last step was an application of rule $||\beta_1 Abs1$ and so $M \equiv \lambda x[M_1]$ and $N \equiv \lambda x[N_1]$ for some M_1 and N_1 respectively and hence $N_1 \;{}^{1}_{||\beta}\!\!\longleftarrow L_1 \longrightarrow^{1}_{||\beta} M_1$. By the induction hypothesis we have

$$\exists P_1 : N_1 \longrightarrow^{1}_{||\beta} P_1 \;{}^{1}_{||\beta}\!\!\longleftarrow M_1$$

Hence by choosing $P \equiv \lambda x[P_1]$ we obtain the required result.

Case $L \equiv (L_1 \;\; L_2)$ and L_1 is not an abstraction.

The rule $||\beta_1 App$ is the only rule that must have been applicable in the last step of the proofs of $N \;{}^{1}_{||\beta}\!\!\longleftarrow L \longrightarrow^{1}_{||\beta} M$. Clearly then there exist M_1, M_2, N_1, N_2 such that $N_1 \;{}^{1}_{||\beta}\!\!\longleftarrow L_1 \longrightarrow^{1}_{||\beta} M_1$ and $N_2 \;{}^{1}_{||\beta}\!\!\longleftarrow L_2 \longrightarrow^{1}_{||\beta} M_2$. Again by the induction hypothesis, we have

$$\exists P_1 : N_1 \longrightarrow^{1}_{||\beta} P_1 \;{}^{1}_{||\beta}\!\!\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^1_{||\beta} P_2 \ {}^1_{||\beta}\!\longleftarrow M_2$$

By choosing $P \equiv (P_1 \ \ P_2)$ we obtain the desired result.

Case $L \equiv (\lambda x[L_1] \ \ L_2)$.

Here we have four sub-cases depending upon whether each of M and N were obtained by an application of $||\beta_1 App$ or $||\beta_1 Abs2$. Of these the sub-case when both M and N were obtained by applying $||\beta_1 App$ is easy and similar to the previous case. That leaves us with three subscases.

Sub-case: Both M and N were obtained by applying rule $||\beta_1 Abs2$.

Then we have

$$\{N_2/x\}N_1 \equiv N \ {}^1_{||\beta}\!\longleftarrow L \equiv (\lambda x[L_1] \ \ L_2) \longrightarrow^1_{||\beta} M \equiv \{M_2/x\}M_1$$

for some M_1, M_2, N_1, N_2 such that

$$N_1 \ {}^1_{||\beta}\!\longleftarrow L_1 \longrightarrow^1_{||\beta} M_1$$

and

$$N_2 \ {}^1_{||\beta}\!\longleftarrow L_2 \longrightarrow^1_{||\beta} M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \longrightarrow^1_{||\beta} P_1 \ {}^1_{||\beta}\!\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^1_{||\beta} P_2 \ {}^1_{||\beta}\!\longleftarrow M_2$$

and the last part of lemma 15.19 we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow^1_{||\beta} P \ {}^1_{||\beta}\!\longleftarrow M$$

completing the proof.

Sub-case: M was obtained by applying rule $||\beta_1 Abs2$ and N by $||\beta_1 App$.

Then we have the form

$$(\lambda x[N_1] \;\; N_2) \equiv N \;{}^{1}_{||\beta}\!\!\longleftarrow L \equiv (\lambda x[L_1] \;\; L_2) \longrightarrow^{1}_{||\beta} M \equiv \{M_2/x\}M_1$$

where again

$$N_1 \;{}^{1}_{||\beta}\!\!\longleftarrow L_1 \longrightarrow^{1}_{||\beta} M_1$$

and

$$N_2 \;{}^{1}_{||\beta}\!\!\longleftarrow L_2 \longrightarrow^{1}_{||\beta} M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \longrightarrow^{1}_{||\beta} P_1 \;{}^{1}_{||\beta}\!\!\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^{1}_{||\beta} P_2 \;{}^{1}_{||\beta}\!\!\longleftarrow M_2$$

and finally we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow^{1}_{||\beta} P \;{}^{1}_{||\beta}\!\!\longleftarrow M$$

completing the proof.

Sub-case: M was obtained by applying rule $||\beta_1 App$ and N by $||\beta_1 Abs2$.

Similar to the previous sub-case.

QED

Beta and Parallel Beta: Confluence

To show that $\longrightarrow^1_\beta$ is Church-Rosser, it suffices to prove that it is confluent.

Theorem 15.21 $\longrightarrow^1_{||\beta}$ *is confluent.*

■

Corollary 15.22 $\longrightarrow^1_\beta$ *is confluent.*

Proof: Since $\longrightarrow^*_\beta \;=\; \longrightarrow^*_{||\beta}$ by lemma 15.19 it follows from theorem 15.21 that $\longrightarrow^1_\beta$ is confluent. QED

Proof of theorem 15.21.

Proof: We need to show that for all L, M, N,

$$N \;{}^{*}_{||\beta}\!\!\longleftarrow L \longrightarrow^{*}_{||\beta} M \Rightarrow \exists P : N \longrightarrow^{*}_{||\beta} P \;{}^{*}_{||\beta}\!\!\longleftarrow M$$

We prove this by induction on the length of the sequences

$$L \longrightarrow^{1}_{||\beta} M_1 \longrightarrow^{1}_{||\beta} M_2 \longrightarrow^{1}_{||\beta} \cdots \longrightarrow^{1}_{||\beta} M_m \equiv M$$

and

$$L \longrightarrow^{1}_{||\beta} N_1 \longrightarrow^{1}_{||\beta} N_2 \longrightarrow^{1}_{||\beta} \cdots \longrightarrow^{1}_{||\beta} N_n \equiv N$$

where $m, n \geq 0$. More specifically we prove this by induction on the pairs of integers (j, i) bounded by (n, m), where $(j, i) < (j', i')$ if and only if either $j < j'$ or $(j = j')$ and $i < i'$. The interesting cases are those where both $m, n > 0$. So we repeatedly apply theorem 15.20 to complete the rectangle

$$\begin{array}{ccccccccc}
L & \longrightarrow^{1}_{||\beta} & M_1 & \longrightarrow^{1}_{||\beta} & M_2 & \longrightarrow^{1}_{||\beta} & \cdots & \longrightarrow^{1}_{||\beta} & M_m \equiv M \\
{}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & \cdots & & {}_{||\beta}\!\downarrow^{1} \\
N_1 & \longrightarrow^{1}_{||\beta} & P_{11} & \longrightarrow^{1}_{||\beta} & P_{12} & \longrightarrow^{1}_{||\beta} & \cdots & \longrightarrow^{1}_{||\beta} & P_{1m} \\
{}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & \cdots & & {}_{||\beta}\!\downarrow^{1} \\
\vdots & & \vdots & & \vdots & & \cdots & & \vdots \\
{}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & {}_{||\beta}\!\downarrow^{1} & & \cdots & & {}_{||\beta}\!\downarrow^{1} \\
N_n & \longrightarrow^{1}_{||\beta} & P_{n1} & \longrightarrow^{1}_{||\beta} & P_{n2} & \longrightarrow^{1}_{||\beta} & \cdots & \longrightarrow^{1}_{||\beta} & P_{nm} \equiv P
\end{array}$$

QED ■

The Church-Rosser Property and β-nf

Corollary 15.23 $\longrightarrow^1_\beta$ *is Church-Rosser.*

Proof: Follows from corollary 15.22 and theorem 15.12. QED

Corollary 15.24 *If a term reduces to a β-normal form then the normal form is unique (upto $\equiv_\alpha$).*

Proof: If $N_1 \;{}^{*}_{\beta}\!\!\longleftarrow L \longrightarrow^*_\beta N_2$ and both N_1 N_2 are β-nfs, then by the corollary 15.22 they must both be β-reducible to a third element N_3 which is impossible if both N_1 and N_2 are β-nfs. Hence β-nfs are unique (upto $\equiv_\alpha$) whenever they exist. QED

Finding β-nf

The following shows that if a normal form exists, then there is a β-reduction sequence which will find it.

Corollary 15.25 *If* $L =_\beta N \in \beta$*-nf then* $L \to_\beta^* N$.

Proof: By the Church-Rosser property both L and N reduce to a common form M. But since N is in normal form $M \equiv_\alpha N$. QED

16. An Applied Lambda-Calculus

16.1. FL with recursion

An Applied Lambda-Calculus With Types

16.2. Motivation and Organization

In the sequel we will define, by stages a simple higher order programming language.

Stage 0. A simple expression language to represent integers and booleans. Initially we define a representation for integers and booleans purely symbolically (29) as a data type with constant constructors.

Stage 1. FL(X) – a simple expression (functional programming) language with variables that allows expressions to be defined on the two types of data – integers and booleans.

Static Semantics. By allowing more than one type of data we also show that there is a need for a type-checking discipline since several meaningless constructs may be generated by the grammar. We specify the type-checking (type-inferencing) system for this simple language as the static semantics of the language.

Functional Semantics. For the well-typed terms we also define the *intended meanings* of these expressions, by defining a functional semantics.

Operational (Reduction) Semantics. We show that we can capture the intended meanings of well-typed expressions by a dynamic semantics which specifies symbolically a notion of reduction (δ-rules (51) to (60)).

Relating Functional and Operational Semantics. The integer values and boolean values are denoted symbolically by δ-normal forms. Lemma 16.3 shows that the intended values of integers and boolean values are obtained as

δ-normal forms and in combination with the property of confluence (see exercise 16.1 problem 3) it follows that all integer and boolean values have unique normal form representations in the expression language.

Subject Reduction. In problem 16.1.5 we encourage the reader to show that types are preserved under δ-reductions (5) i.e. the type of an expression cannot change arbitrarily during reduction (program execution) – an important static property that a dynamic semantics should obey.

Referential Transparency. Further, in problem (6) the reader is encouraged to show that the language enjoys the property of referential transparency viz. that each variable name in an expression may be substituted by its value while preserving the meaning of the expression – a dynamic property that any functional programming language should obey.

Stage 2. Λ+FL(X). However, the language FL(X) lacks the elementary facilities for user-defined functions. Add to that the lack of expressiveness to define even the most common useful integer operations such as addition, subtraction and multiplication. We rectify this by defining λ-abstraction and application to terms of the language. The new extended language Λ+FL(X) allows us to define some (non-recursive) operators and functions over the terms of the language FL(X). BY this addition, β-reduction has been added to the language as well. The language Λ+FL(X) is expressive enough to define some of the common boolean operators and the most common order relations on integers. These have been made possible due to the inclusion of the ternary if-the-else construct(or) `ITE` and construct(or)s for checking for 0 (`IZ`) and positive integer values (`GTZ`).

Stage 3. The addition of the λ-abstraction and application on top of FL(X) has the drawback that functions and function applications do not have the same status as expressions. To bring function definition and application down to the expression level it is necessary to allow an intermingling of the two. Hence we "flatten" the language to produce a genuinely applied λ-calculus with a $\beta\delta$ reduction mechanism. The result is the language $\Lambda_{FL}(X)$.

Stage 4. $\Lambda_{FL}(X)$ allows the full power of the λ-calculus to be incorporated into the language. Hence it allows higher-order functions as well. However, the power of recursion is not achieved in a type-safe manner because no paradoxical combinator can be made type-safe. Hence even to program some elementary inductive functions like addition, a recursion operator is absolutely required. This yields the language $\Lambda_{RecFL}(X)$.

A Simple Language of Terms: FL0

Let X be an infinite collection of variables (names). Consider the language (actually a collection of abstract syntax trees) of terms $T_\Omega(X)$ defined by the following constructors (along with their intended meanings). T_Ω denotes the variable-free subset of $T_\Omega(X)$ and is called the set of **ground terms**.

Construct	Arity	Informal Meaning
`Z`	0	The number **0**
`T`	0	The truth value **true**
`F`	0	The truth value **false**
`P`	1	The **predecessor** function on numbers
`S`	1	The **successor** function on numbers
`ITE`	3	The **if-then-else** construct (on numbers and truth values)
`IZ`	1	The **is-zero** predicate on numbers
`GTZ`	1	The **greater-than-zero** predicate on numbers

FL(X): Language, Datatype or Instruction Set?

The set of terms $T_\Omega(X)$, where X is an infinite collection of variable names (that are disjoint from all other symbols in the language) may be defined by the BNF:

$$t ::= x \in X \mid \mathtt{Z} \mid (\mathtt{P}\ \ t) \mid (\mathtt{S}\ \ t) \mid \mathtt{T} \mid \mathtt{F} \mid (\mathtt{ITE}\ \ \langle t, t_1, t_0\rangle) \mid (\mathtt{IZ}\ \ t) \mid (\mathtt{GTZ}\ \ t) \tag{29}$$

- It could be thought of as a user-defined data-type
- It could be thought of as the instruction-set of a particularly simple hardware machine.
- It could be thought of as a simple functional programming language without recursion.
- It is a language with two simple types of data: integers and booleans
- Notice that the constructor $(\mathtt{ITE}\ \ \langle t, t_1, t_0\rangle)$ is overloaded.

Extending the language

To make this simple language safe we require

Type-checking: to ensure that arbitrary expressions are not mixed in ways they are not "intended" to be used. For example

- t cannot be a boolean expression in $(\texttt{S}\ \ t)$, $(\texttt{P}\ \ t)$, $(\texttt{IZ}\ \ t)$ and $(\texttt{GTZ}\ \ t)$
- $(\texttt{ITE}\ \ \langle t, t_1, t_0\rangle)$ may be used as a conditional expression for both integers and booleans, but t needs to be a boolean and either both t_1 and t_0 are integer expressions or both are boolean expressions.

Functions: To be a useful programming language we need to be able to define functions.

Recursion: to be able to define complex functions in a well-typed fashion. Recursion should also be well-typed

Typing FL0 Expressions

We have only two types of objects in FL0 – integers and booleans which we represent by $\underline{\mathtt{int}}$ and $\underline{\mathtt{bool}}$ respectively. We then have the following elementary typing annotations for the expressions, which may be obtained by pattern matching.

Basis. $\mathtt{Z} : \underline{\mathtt{int}}, \quad \mathtt{T} : \underline{\mathtt{bool}}, \quad \mathtt{F} : \underline{\mathtt{bool}}$

Int. $\mathtt{S} : \underline{\mathtt{int}} \to \underline{\mathtt{int}}, \quad \mathtt{P} : \underline{\mathtt{int}} \to \underline{\mathtt{int}}$

Bool. $\mathtt{IZ} : \underline{\mathtt{int}} \to \underline{\mathtt{bool}}, \quad \mathtt{GTZ} : \underline{\mathtt{int}} \to \underline{\mathtt{bool}}$

boolCond. $\mathtt{ITEB} : \underline{\mathtt{bool}} * \underline{\mathtt{bool}} * \underline{\mathtt{bool}} \to \underline{\mathtt{bool}}$

intCond. $\mathtt{ITEI} : \underline{\mathtt{bool}} * \underline{\mathtt{int}} * \underline{\mathtt{int}} \to \underline{\mathtt{int}}$

16.3. Static Semantics of FL(X)

While **dynamic** semantics refers to the run-time behaviour of a program, the **static** semantics refers to all the context-sensitive information about a program that needs to be gathered during the compilation process, to enable the generation of code both for execution as well as error-reporting and handling. Most of this information about variable symbols is stored in the symbol table and is accessed during the code-generation process for both memory allocation and the actual generation of target code.

The purposes of both code-generation and memory allocation aspects are more or less (i.e. except for scope and the absolute addresses of the data-objects during execution) covered by determining the types of the various objects in a program (data objects, functions, procedures etc.). The type of a scalar data item implicitly defines the amount of storage it requires. For example, an integer variable needs perhaps one word of storage and a floating point variable requires two-words of storage, booleans require just a bit (but in the case of byte-addressable or word-addressable machines machines it may be more efficient to assign a byte or word of storage to it). Similarly characters may require a byte of storage and strings require storage that is proportional to their length. All complex data items such as records and arrays being built of the scalar components require correspondingly proportional amounts of storage in the run-time stack. For each of these the compiler creates a so-called *data descriptor* and stores it in the symbol table and refers to it while generating code. The control units viz. expressions, commands, functions and procedures would require storage (in the code-segment) proportional to the length of the code that is generated for each of them; and the parameters they invoke correspondingly require data descriptors to define the storage requirements for the parameters. Further in the process of compiling a procedure or a function the types of input and output parameters in the definition (declaration) should correspond exactly with the types of the actual parameters in each call (otherwise a compile-time error needs to be generated).

Much of the above process can all be captured by the simple process of assigning types to each data and control unit in a program. Hence most compilers (with static scoping rules) actually perform *static* or *compile-time* type-checking.

16.3.1. Type-checking FL(X) terms

While trying to type FL0 expressions we have had to introduce two new type operators viz. $*$ and $\rightarrow$ which allow us to precisely capture the types of expressions involving constructors such as `S`, `P`, `IZ`, `GTZ`, `ITE` etc. which we intend to view as functions of appropriate arity on appropriate types of arguments. These type operators will be required for specifying the types of other (user-defined) functions as well. Hence it makes sense for us to define a *formal* language of type expressions (with type variables!) to enable us define types of polymorphic operations (which in the particular case of FL(X) is restricted to overloading the `ITE` constructor). As we shall see later, this expression language of types may be defined by the grammar

$$\sigma, \tau ::= \underline{\mathtt{int}} \mid \underline{\mathtt{bool}} \mid {}'\mathtt{a} \in TV \mid (\sigma * \tau) \mid (\sigma \rightarrow \tau)$$

where $'\mathtt{a} \in TV$ is a type variable and all type variables are distinct from program variables in X.

What we have specified earlier are the typing axioms for the *constant* expressions (without variables). For the purpose of typing expressions involving (free) variables we require assumptions to be made about the types of the variables occurring in an expression. In most programming languages these assumptions come from the declarations of variables. For instance, the successor constructor S should be applied only to expressions which yield integer values. Hence for any expression $t \in FL(X)$, $(\mathtt{S}\ \ t)$ would be *well-typed* only if $t : \underline{\mathtt{int}}$ and further $(\mathtt{S}\ \ t) : \underline{\mathtt{int}}$. Similarly, given three expressions t, t_1, t_0, the expression $(\mathtt{ITE}\ \ \langle t, t_1, t_0\rangle)$ *type-checks* i.e. it is well-typed only if $t : \underline{\mathtt{bool}}$ and the types of t_1 and t_0 are the same – either both $\underline{\mathtt{bool}}$ or both $\underline{\mathtt{int}}$.

16.3.2. The Typing Rules

In a language of expressions that requires the type of each variable to be declared beforehand, the list of (free) variables and their types may be available as a *type environment* Γ and the rules that we give are *type-checking* rules. The rules for type-checking any expression $t \in T_\Omega(X)$ extend the earlier specification by induction on the structure of expressions. More precisely, the earlier specification form the basis of an induction by structure of expressions. The 0-ary constructors and variables form the basis for the structural induction rules and have the following axioms and their types are independent of the type environment. The type-checking rules for expressions form the induction step of the type-checking algorithm and go as follows. These rules also assign types to each individual sub-expression along the way. We begin with the unary constructors and conclude with the conditional operator.

Alternatively, in the absence of declarations, we could derive them as *constraints* on the types of variables (as we shall see later). It is then necessary to also use the concept of type variables as distinct from program variables.

Static Semantics of FL(X): Type-checking

$$\textbf{Ft}\ \frac{}{\Gamma \vdash \mathtt{F} : \underline{\mathtt{bool}}} \qquad \textbf{Tt}\ \frac{}{\Gamma \vdash \mathtt{T} : \underline{\mathtt{bool}}} \qquad \textbf{Zt}\ \frac{}{\Gamma \vdash \mathtt{Z} : \underline{\mathtt{int}}} \qquad \textbf{Var}\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\textbf{St}\ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{S}\ \ t) : \underline{\mathtt{int}}} \qquad \textbf{Pt}\ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{P}\ \ t) : \underline{\mathtt{int}}}$$

$$\textbf{IZt}\ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{IZ}\ \ t) : \underline{\mathtt{bool}}} \qquad \textbf{GTZt}\ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{GTZ}\ \ t) : \underline{\mathtt{bool}}}$$

$$\textbf{ITEIt}\ \frac{\Gamma \vdash t : \underline{\mathtt{bool}} \quad \Gamma \vdash t_1 : \underline{\mathtt{int}} \quad \Gamma \vdash t_0 : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{ITE}\ \ \langle t, t_1, t_0 \rangle) : \underline{\mathtt{int}}} \qquad \textbf{ITEBt}\ \frac{\Gamma \vdash t : \underline{\mathtt{bool}} \quad \Gamma \vdash t_1 : \underline{\mathtt{bool}} \quad \Gamma \vdash t_0 : \underline{\mathtt{bool}}}{\Gamma \vdash (\mathtt{ITE}\ \ \langle t, t_1, t_0 \rangle) : \underline{\mathtt{bool}}}$$

Well-typed Terms

Definition 16.1 *A term $t \in T_\Omega(X)$ in a type environment Γ is well-typed if there exists a proof of either $\Gamma \vdash t : \texttt{int}$ or $\Gamma \vdash t : \texttt{bool}$ (not both).*

As we have seen before there are terms that are not well-typed. We consider only the subset $WT_\Omega(X) \subset T_\Omega(X)$ while describing the dynamic semantics. While T_Ω is variable-free subset of $T_\Omega(X)$, $WT_\Omega \subset WT_\Omega(X)$ is the variable-free subset of $WT_\Omega(X)$.

Dynamic Semantics of FL(X)

The dynamic semantics or the run-time behaviour of FL(X) expressions may be specified in several ways.

Functional semantics. The language *designer* could specify the intended meanings of the constants, constructors and operators in terms that are useful to the user programmer as functions (as an extension of the informal meaning specified earlier), or

Operational semantics The *implementor* of the language could specify the run-time behaviour of expressions through an abstract algorithm.

But any *implementation* should also be consistent with the intended meanings specified by the *designer*

Functional Semantics of FL(X):0

Boolean constants . The constructors T and F are interpreted as the boolean constants $true$ and $false$ respectively.

Zero . Z is interpreted as the *constant* number 0

Positive integers . Each k-fold application, $k > 0$, of the constructor S to Z viz. $\underbrace{(\mathtt{S}\ \ldots(\mathtt{S}\ \mathtt{Z})\ldots)}_{k-fold}$ (abbreviated to $(\mathtt{S}^k\ \ \mathtt{Z})$ for convenience) is interpreted as the positive integer k.

Negative integers . Similarly, each k-fold application, $k > 0$, of the constructor P to Z viz. $\underbrace{(\mathtt{P}\ \ldots(\mathtt{P}\ \mathtt{Z})\ldots)}_{k-fold}$ (abbreviated as $(\mathtt{P}^k\ \ \mathtt{Z})$) is interpreted as $-k$.

Functional Semantics of FL(X):1

- Let $\mathbb{Z}$ and $\mathbb{B}$ denote the sets of integers and booleans respectively.
- Each well-typed expression in $T_\Omega(X)$ denotes either an integer or boolean value depending upon its type.
- Let $\mathcal{V} = \{v \mid v : X \to (\mathbb{Z} \cup \mathbb{B})\}$ denote the set of all *valuation environments* which associate with each variable a value of the appropriate type (either integer or boolean).
- With the interpretation of the symbols in the language given earlier we associate a meaning function

$$\mathscr{M} : WT_\Omega(X) \to (\mathcal{V} \to (\mathbb{Z} \cup \mathbb{B}))$$

such that for each well-typed expression $t \in WT_\Omega(X)$, $\mathscr{M}[\![t]\!]$ is a function that extends each $v \in \mathcal{V}$, inductively on the structure of expressions to a value of the appropriate type.

Functional Semantics of FL(X):2

$$\mathscr{M}[\![x]\!]\ v \stackrel{df}{=} v(x) \tag{30}$$

$$\mathscr{M}[\![\mathtt{T}]\!]\ v \stackrel{df}{=} true \tag{31}$$

$$\mathscr{M}[\![\mathtt{F}]\!]\ v \stackrel{df}{=} false \tag{32}$$

$$\mathscr{M}[\![\mathtt{Z}]\!]\ v \stackrel{df}{=} 0 \tag{33}$$

$$\mathscr{M}[\![(\mathtt{P}\ \ t)]\!]\ v \stackrel{df}{=} \mathscr{M}[\![t]\!]\ v - 1 \tag{34}$$

$$\mathscr{M}[\![(\mathtt{S}\ \ t)]\!]\ v \stackrel{df}{=} \mathscr{M}[\![t]\!]\ v + 1 \tag{35}$$

$$\mathscr{M}[\![(\mathtt{IZ}\ \ t)]\!]\ v \stackrel{df}{=} \mathscr{M}[\![t]\!]\ v = 0 \tag{36}$$

$$\mathscr{M}[\![(\mathtt{GTZ}\ \ t)]\!]\ v \stackrel{df}{=} \mathscr{M}[\![t]\!]\ v > 0 \tag{37}$$

$$\mathscr{M}[\![(\mathtt{ITE}\ \ \langle t, t_1, t_0\rangle)]\!]\ v \stackrel{df}{=} \begin{cases} \mathscr{M}[\![t_1]\!]\ v \text{ if } \mathscr{M}[\![t]\!]\ v \\ \mathscr{M}[\![t_0]\!]\ v \text{ if not } \mathscr{M}[\![t]\!]\ v \end{cases} \tag{38}$$

16.4. Equational Reasoning in FL(X)

From the semantics of FL(X) the following identitities are easily derived. We leave the proofs of these identities to the reader. It is also important that some of these identities are used (oriented from left to right) in the definition of the δ-rules as rules of reduction (or "simplification") in order to obtain normal forms. In such cases the equality is made asymmetric (left to right).

Identities used for simplification

$$(\mathtt{P}\ \ (\mathtt{S}\ \ x)) = x \tag{39}$$

$$(\mathtt{S}\ \ (\mathtt{P}\ \ x)) = x \tag{40}$$

$$(\mathtt{ITE}\ \ \langle \mathtt{T}, x, y\rangle) = x \tag{41}$$

$$(\mathtt{ITE}\ \ \langle \mathtt{F}, x, y\rangle) = y \tag{42}$$

$$(\mathtt{IZ}\ \ \mathtt{Z}) = \mathtt{T} \tag{43}$$

$$(\mathtt{GTZ}\ \ \mathtt{Z}) = \mathtt{F} \tag{44}$$

Identities involving normal forms

$$(\texttt{IZ}\ \ (\texttt{S}\ \ n)) = \texttt{F}, \text{ where } (\texttt{S}\ \ n) \text{ is a } \delta\text{-nf} \tag{45}$$
$$(\texttt{IZ}\ \ (\texttt{P}\ \ n)) = \texttt{F}, \text{ where } (\texttt{P}\ \ n) \text{ is a } \delta\text{-nf} \tag{46}$$
$$(\texttt{GTZ}\ \ (\texttt{S}\ \ n)) = \texttt{T}, \text{ where } (\texttt{S}\ \ n) \text{ is a } \delta\text{-nf} \tag{47}$$
$$(\texttt{GTZ}\ \ (\texttt{P}\ \ n)) = \texttt{F}, \text{ where } (\texttt{P}\ \ n) \text{ is a } \delta\text{-nf} \tag{48}$$

Besides the above identities which are actually used in an oriented form for the purpose of computation we may also prove other identities from the functional semantics. Many of these look like they could be included in the rules for computation, but we may not be because of

- the limits of computability in general and
- their inclusion might at times lead to non-determinism and
- in more extreme cases lead to non-termination even though there are deterministic ways to obtain δ-normal forms.

However they are useful for reasoning about programs written in the language. For example the following obvious identity

$$(\texttt{ITE}\ \ \langle b, x, x\rangle) = x, \text{ where } b \text{ is a boolean} \tag{49}$$
$$\tag{50}$$

is useful for simplifying a program for human reasoning. However, when included as a δ-rule, it greatly complicates the computation when equality of the two arms of the conditional need to be checked (when they are not merely variables but complicated expressions themselves). There is a further complication of defining under what conditions this equality checking needs to be performed.

Reduction Semantics

Just as the dynamic behaviour of a λ-term may be described by β-reduction, we may describe the dynamic behaviour of a FL(X) expression through a notion of reduction called $\delta - reduction$. It is important that such a notion of reduction produces results (values) that are consistent with the functional semantics.

Example 16.2 *The simplifications used to obtain the answer* 197 *from the expression* $14^2 + 1$ *is an example of the* δ*-rules used in an applied* λ*-calculus on the naturals.*

We first begin with the δ-normal forms for integers and booleans.

The Normal forms for Integers

Zero. $\mathtt{Z}$ is the unique representation of the number 0 and every integer expression that is equal to 0 must be reducible to $\mathtt{Z}$.

Positive integers. Each positive integer k is uniquely represented by the expression $(\mathtt{S}^k\ \ \mathtt{Z})$ where the super-script k denotes a k-fold application of $\mathtt{S}$.

Negative integers. Each negative integer $-k$ is uniquely represented by the expression $(\mathtt{P}^k\ \ \mathtt{Z})$ where the super-script k denotes a k-fold application of $\mathtt{P}$.

δ rules for Integers

See also section 16.4

$$(\texttt{P} \quad (\texttt{S} \quad x)) \longrightarrow_{\delta} x \tag{51}$$

$$(\texttt{S} \quad (\texttt{P} \quad x)) \longrightarrow_{\delta} x \tag{52}$$

δ Normal Forms

Lemma 16.3 *The following well-typed (in any type environment Γ) terms are exactly the δ-normal forms in WT_Ω along with their respective meanings in the functional semantics (in any dynamic environment $v \in \mathcal{V}$).*

1. $\Gamma \vdash \mathtt{Z} : \underline{\mathtt{int}}$ *and* $\mathscr{M}[\![\mathtt{Z}]\!]\ v = 0$
2. $\Gamma \vdash \mathtt{T} : \underline{\mathtt{bool}}$ *and* $\mathscr{M}[\![\mathtt{T}]\!]\ v = true$
3. $\Gamma \vdash \mathtt{F} : \underline{\mathtt{bool}}$ *and* $\mathscr{M}[\![\mathtt{F}]\!]\ v = false$
4. *For each positive integer* k, $\Gamma \vdash (\mathtt{S}^k\ \ \mathtt{Z}) : \underline{\mathtt{int}}$ *and* $\mathscr{M}[\![(\mathtt{S}^k\ \ \mathtt{Z})]\!]\ v = k$
5. *For each positive integer* k, $\Gamma \vdash (\mathtt{P}^k\ \ \mathtt{Z}) : \underline{\mathtt{int}}$ *and* $\mathscr{M}[\![(\mathtt{P}^k\ \ \mathtt{Z})]\!]\ v = -k$

δ Rules for Conditional

See also section 16.4

Pure Boolean Reductions . The constructs `T` and `F` are the normal forms for boolean values.

$$(\texttt{ITE}\ \ \langle \texttt{T}, x, y\rangle) \longrightarrow_\delta x \tag{53}$$

$$(\texttt{ITE}\ \ \langle \texttt{F}, x, y\rangle) \longrightarrow_\delta y \tag{54}$$

δ Rules: Zero Test

See also section 16.4

Testing for zero .

$$(\mathtt{IZ}\ \ \mathtt{Z}) \longrightarrow_{\delta} \mathtt{T} \tag{55}$$

$$(\mathtt{IZ}\ \ (\mathtt{S}\ \ n)) \longrightarrow_{\delta} \mathtt{F}, \text{ where } (\mathtt{S}\ \ n) \text{ is a } \delta\text{-nf} \tag{56}$$

$$(\mathtt{IZ}\ \ (\mathtt{P}\ \ n)) \longrightarrow_{\delta} \mathtt{F}, \text{ where } (\mathtt{P}\ \ n) \text{ is a } \delta\text{-nf} \tag{57}$$

δ Rules: Positivity

See also section 16.4

$$(\texttt{GTZ}\ \ \texttt{Z}) \longrightarrow_\delta \texttt{F} \tag{58}$$

$$(\texttt{GTZ}\ \ (\texttt{S}\ \ n)) \longrightarrow_\delta \texttt{T}, \text{ where } (\texttt{S}\ \ n) \text{ is a } \delta\text{-nf} \tag{59}$$

$$(\texttt{GTZ}\ \ (\texttt{P}\ \ n)) \longrightarrow_\delta \texttt{F}, \text{ where } (\texttt{P}\ \ n) \text{ is a } \delta\text{-nf} \tag{60}$$

Exercise 16.1

1. *Find examples of expressions in FL0 which have more than one computation.*
2. *Prove that* $\longrightarrow_\delta$ *is terminating.*
3. *Prove that* $\longrightarrow_\delta$ *is Church-Rosser.*
4. *The language FL(X) extends FL0 with variables. What are the new* δ*-normal forms in FL(X)?*
5. **Subject reduction**. *Prove that for any well-typed term* $t \in WT_\Omega(X)$*, and* $\alpha \in \{\underline{\texttt{int}}, \underline{\texttt{bool}}\}$ *if* $\Gamma \vdash t : \alpha$ *and* $t \longrightarrow_\delta t'$ *then* $\Gamma \vdash t' : \alpha$.
6. **Referential Transparency**. *Let* $t \in WT_\Omega(X)$*,* $FV(t) = \{x_1, \ldots, x_n\}$ *and let* v *be a valuation environment. If* $\{t_1, \ldots, t_n\}$ *are ground terms such that for each* i*,* $1 \leq i \leq n$*,* $\mathscr{M}[\![x_i]\!]\ v = \mathscr{M}[\![t_i]\!]\ v$ *then prove that*

   (a) $\mathscr{M}[\![t]\!]\ v = \mathscr{M}[\![\{t_1/x_1, \ldots, t_n/x_n\}t]\!]\ v$ *and*

   (b) $\{t_1/x_1, \ldots, t_n/x_n\}t \longrightarrow_\delta^* u$ *where* $\mathscr{M}[\![u]\!]\ v = \mathscr{M}[\![t]\!]\ v$

   where $\{t_1/x_1, \ldots, t_n/x_n\}t$ *denotes the simultaneous syntactic substitution of every occurrence of variable* x_i *by the ground term* t_i *for* $1 \leq i \leq n$.

Λ+FL(X): The Power of Functions

To make the language powerful we require the ability to define functions, both non-recursive and recursive. We define an applied lambda-calculus of lambda terms $\Lambda_\Omega(X)$ over this set of terms as follows:

$$L, M, N ::= t \in T_\Omega(X) \;\Big|\; \lambda x[L] \;\Big|\; (L\ \ M) \tag{61}$$

This is a two-level grammar combining the term grammar (29) with λ-abstraction and λ-application.

Some Non-recursive Operators

We may "program" the other boolean operations as follows:

$$\begin{aligned}
\mathsf{NOT} &\stackrel{df}{=} \lambda x[\mathtt{ITE}\ \langle x, \mathtt{F}, \mathtt{T}\rangle] \\
\mathsf{AND} &\stackrel{df}{=} \lambda\langle x, y\rangle[\mathtt{ITE}\ \langle x, y, \mathtt{F}\rangle] \\
\mathsf{OR} &\stackrel{df}{=} \lambda\langle x, y\rangle[\mathtt{ITE}\ \langle x, \mathtt{T}, y\rangle]
\end{aligned}$$

We may also "program" the other integer comparison operations as follows:

$$\begin{aligned}
\mathsf{GEZ} &\stackrel{df}{=} \lambda x[\mathtt{OR}\ \langle(\mathtt{IZ}\ x), (\mathtt{GTZ}\ x)\rangle] \\
\mathsf{LTZ} &\stackrel{df}{=} \lambda x[\mathtt{NOT}\ (\mathtt{GEZ}\ x)] \\
\mathsf{LEZ} &\stackrel{df}{=} \lambda x[\mathtt{OR}\ \langle(\mathtt{IZ}\ x), (\mathtt{LTZ}\ x)\rangle]
\end{aligned}$$

Λ+FL(X): Lack of Higher-order Power?

Example 16.4 *The grammar (61) does not allow us to define expressions such as the following:*

1. *the successor of the result of an application* $(\mathtt{S}\ \ (L\ \ M))$ *where* $(L\ \ M)$ *yields an integer value.*
2. *higher order conditionals e.g.* $\lambda x[(\mathtt{ITE}\ \ \langle(L\ \ x),(M\ \ x),(N\ \ x)\rangle)]$ *where* $(L\ \ x)$ *yields a boolean value for an argument of the appropriate type.*
3. *In general, it does not allow the constructors to be applied to* λ*-expressions.*

So we extend the language by allowing a free intermixing of λ-terms and terms of the sub-language $T_\Omega(X)$.

$\Lambda_{FL}(X)$: Higher order functions

We need to *flatten* the grammar of (61) to allow λ-terms also to be used as arguments of the constructors of the term-grammar (29). The language of applied λ-terms (viz. $\Lambda_\Omega(X)$) now is defined by the grammar.

$$
\begin{aligned}
L, M, N ::= x \in X \quad & \mid\ \mathtt{Z} \ \mid\ \mathtt{T} \ \mid\ \mathtt{F} \\
& \mid\ (\mathtt{P}\ \ L) \ \mid\ (\mathtt{S}\ \ L) \\
& \mid\ (\mathtt{IZ}\ \ L) \ \mid\ (\mathtt{GTZ}\ \ L) \\
& \mid\ (\mathtt{ITE}\ \ \langle L, M, N\rangle) \\
& \mid\ \lambda x[L] \ \mid\ (L\ \ M)
\end{aligned} \tag{62}
$$

Unfortunately the result of *flatten*ing the grammar leads to an even larger number of meaningless expressions (in particular, we may be able to generate self-referential ones or ones that may not even be interpretable as functions which yield integer or boolean values.

It is therefore imperative that we define a *type-checking* mechanism to rule out meaningless expressions. As mentioned before, type-checking is not context-free and hence cannot be done through mechanisms such as scanning and parsing and will have to be done separately before any code-generation takes place.

We will in fact, go a step further and design a *type-inferencing* mechanism that will prevent meaningless expressions from being allowed.

Further, given a well-typed expression we need to be able to define a meaning for each expression that is somehow compatible with our intuitive understanding of what λ-expressions involving integer and boolean operations mean. This meaning is defined through an *operational semantics* i.e. a system of transitions on how computation actually takes place for each expression. We define this through a reduction mechanism that is consistent with reduction relations that we have earlier studied for the untyped λ-calculus.

In order for it to be compatible with the notions of reduction in the λ-calculus we require to define a notion of reduction first for expressions that do not involve either λ abstraction or λ application. We refer to this notion of reduction as *δ-reduction*. Furthermore we need to be able to define *δ-normal forms* for these expressions. Since the language is completely symbolic, these normal forms would serve as the final answers obtained in the evaluation of these expressions.

Exercise 16.2

1. *Prove that the language of (61) is properly contained in the language of (62).*
2. *Give examples of meaningful terms generated by the grammar (62) which cannot be generated by the grammar (61).*

The full power of a programming language will not be realised without a recursion mechanism. The untyped lambda-calculus has "paradoxical combinators" which behave like recursion operators upto $=_\beta$.

Definition 16.5 *A combinator* Y *is called a* **fixed-point combinator** *if for every lambda term* L, Y *satisfies the fixed-point property*

$$\boxed{(\mathsf{Y}\ \ L) =_\beta (L\ \ (\mathsf{Y}\ \ L))} \tag{63}$$

Curry's Y combinator ($\mathsf{Y_C}$)

$$\mathsf{Y_C} \stackrel{df}{=} \lambda f[(\mathsf{C}\ \ \mathsf{C})] \text{ where } \mathsf{C} \stackrel{df}{=} \lambda x[(f\ \ (x\ \ x))]$$

Turing's Y combinator ($\mathsf{Y_T}$)

$$\mathsf{Y_T} \stackrel{df}{=} (\mathsf{T}\ \ \mathsf{T}) \text{ where } \mathsf{T} \stackrel{df}{=} \lambda y\ x[(x\ \ (y\ y\ x\ \ ))]$$

The Paradoxical Combinators

Lemma 16.6 *Both* Y_C *and* Y_T *satisfy the fixed-point property.*

$\square$

Proof of lemma 16.6

Proof: For each term L we have

$$\begin{array}{ll} & (\mathsf{Y_c}\ \ L) \\ \equiv & (\lambda f[(\mathsf{C}\ \ \mathsf{C})]\ \ L) \\ \to_\beta^1 & (\{L/f\}\mathsf{C}\ \ \{L/f\}\mathsf{C}) \\ \equiv & (\lambda x[(L\ \ (x\ \ x))]\ \ \lambda x[(L\ \ (x\ \ x))]) \\ =_\beta & (L\ \ (\mathsf{Y_c}\ \ \mathsf{L})) \end{array}$$

Similarly for $\mathsf{Y_T}$ it may be verified that it satisfies the fixed-point property.

$$\begin{array}{ll} & (Y_T\ \ L) \\ \equiv & ((\mathsf{T}\ \ \mathsf{T})\ \ L) \\ \equiv & ((\lambda y[\lambda x[(x\ \ ((y\ \ y)\ \ x))]]\ \ \mathsf{T})\ \ L) \\ \to_\beta^2 & (L\ \ ((\mathsf{T}\ \ \mathsf{T})\ \ L)) \\ =_\beta & (L\ \ (\mathsf{Y_T}\ \ L)) \end{array}$$

QED

$\Lambda_{RecFL}(X)$: Recursion

- But the various Y combinators unfortunately will not satisfy any typing rules that we may define for the language, because they are all "self-applicative" in nature.
- Instead it is more convenient to use the fixed-point property and define a new constructor with a δ-rule which satisfies the fixed-point property (definition (63)).
- `REC` is assigned the type $((\tau \rightarrow \tau) \rightarrow \tau)$ for each type τ.

$\Lambda_{RecFL}(X)$: Adding Recursion

We extend the language $\Lambda_{FL}(X)$ with a new constructor

$$L ::= \ldots \quad | \quad (\mathtt{REC} \;\; L)$$

and add the fixed point property as a δ-rule

$$\boxed{(\mathtt{REC} \;\; L) \longrightarrow_{\delta} (L \;\; (\mathtt{REC} \;\; L)) \qquad (64)}$$

Typing REC

With REC : $((\tau \to \tau) \to \tau)$ and $L : \tau \to \tau$ we have that

$$\begin{aligned}(\mathtt{REC} \quad L) \qquad\quad &: \tau \\ (L \quad (\mathtt{REC} \quad L)) &: \tau\end{aligned}$$

which

- type-checks (without recourse to self-reference) as a constructor and
- is consistent with our intuition about recursion as a syntactic unfolding operator.

Recursion Example: Addition

Consider addition on integers as a binary operation to be defined in this language. We use the following properties of addition on the integers to define it by induction on the first argument.

Example 16.7

$$x + y = \begin{cases} y & \textit{if } x = 0 \\ (x-1) + (y+1) & \textit{if } x > 0 \\ (x+1) + (y-1) & \textit{if } x < 0 \end{cases} \tag{65}$$

Using the constructors of $\Lambda_{RecFL}(X)$ we require that any (curried) definition of addition on numbers should be a solution to the following equation in $\Lambda_{RecFL}(X)$ for all (integer) expression values of x and y.

$$(plusc\ \ x\ y) =_{\beta\delta} \texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (plusc\ \ (\texttt{P}\ x)\ (\texttt{S}\ y)), (plusc\ \ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle \tag{66}$$

Equation (66) may be rewritten using abstraction as follows:

$$plusc =_{\beta\delta} \lambda x[\lambda y[\texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (plusc\ \ (\texttt{P}\ x)\ (\texttt{S}\ y)), (plusc\ \ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle]] \tag{67}$$

We may think of equation (67) as an equation to be solved in the unknown variable $plusc$.

Consider the (applied) λ-term obtained from the right-hand-side of equation (67) by simply abstracting the unknown $plusc$.

$$\mathsf{addc} \stackrel{df}{=} \lambda f[\lambda x\ y[\texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (f\ \ (\texttt{P}\ x)\ (\texttt{S}\ y)), (f\ \ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle]] \tag{68}$$

Claim 16.8

$$(\texttt{REC}\ \ \mathsf{addc}) \longrightarrow_{\delta} (\mathsf{addc}\ \ (\texttt{REC}\ \ \mathsf{addc})) \tag{69}$$

and hence

$$(\texttt{REC}\ \ \mathsf{addc}) =_{\beta\delta} (\mathsf{addc}\ \ (\texttt{REC}\ \ \mathsf{addc})) \tag{70}$$

Claim 16.9 $(\texttt{REC}\ \ \mathsf{addc})$ *satisfies exactly the equation (67). That is*

$$((\texttt{REC}\ \ \mathsf{addc})\ \ x\ y) =_{\beta\delta} \texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), ((\texttt{REC}\ \ \mathsf{addc})\ \ (\texttt{P}\ x)\ (\texttt{S}\ y)), ((\texttt{REC}\ \ \mathsf{addc})\ \ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle \tag{71}$$

Hence we may regard $(\texttt{REC}\ \ \mathsf{addc})$ where addc is defined by the right-hand-side of definition (68) as the required solution to the equation (66) in which *plusc* is an unknown.

The abstraction shown in (68) and the claims (16.8) and (16.9) simply go to show that $M \equiv_{\alpha} \lambda f[\{f/z\}L]$ is a solution to the equation $z =_{\beta\delta} L$, whenever such a solution does exist. Further, the claims also show that we may "unfold" the recursion (on demand) by simply performing the substitution $\{L/z\}L$ for each free occurrence of z within L.

Exercise 16.3

1. *Prove that the relation* $\longrightarrow_\delta$ *is confluent.*
2. *The language FL does not have any operators that take boolean arguments and yields integer values. Define a standard conversion function* B2I *which maps the value* `F` *to* `Z` *and* `T` *to* `(S Z)`.
3. *Using the combinator* add *and the other constructs of* $\Lambda_\Sigma(X)$ *to*
   (a) *define the equation for products of numbers in the language.*
   (b) *define the multiplication operation* mult *on integers and prove that it satisfies the equation(s) for products.*
4. *The equation (65) is defined conditionally. However the following is equally valid for all integer values* x *and* y.
$$x + y = (x - 1) + (y + 1) \tag{72}$$
   (a) *Follow the steps used in the construction of* addc *to define a new applied* addc′ *that instead uses equation (72).*
   (b) *Is* $(\mathtt{REC}\ \ \mathsf{addc}') =_{\beta\delta} (\mathsf{addc}'\ \ (\mathtt{REC}\ \ \mathsf{addc}'))$*?*
   (c) *Is* $\mathsf{addc} =_{\beta\delta} \mathsf{addc}'$*?*
   (d) *Is* $(\mathtt{REC}\ \ \mathsf{addc}) =_{\beta\delta} (\mathtt{REC}\ \ \mathsf{addc}')$*?*
   (e) *Computationally speaking (in terms of* β *and* δ *reductions), what is the difference between* addc *and* addc′*?*
5. *The function* addc *was defined in curried form. Use the pairing function in the untyped* λ*-calculus, to define*
   (a) *addition and multiplication as binary functions independently of the existing functions.*

(b) *the binary 'curry' function which takes a binary function and its arguments and creates a curried version of the binary function.*

Typing $\Lambda_{RecFL(X)}$ expressions

We have already seen that the simple language FL has

- two kinds of expressions: integer expressions and boolean expressions,
- there are also constructors which take integer expressions as arguments and yield boolean values
- there are also function types which allow various kinds of functions to be defined on boolean expressions and integer expressions.

The Need for typing in $\Lambda_{RecFL}(X)$

- A type is an important *attribute* of any variable, constant or expression, since every such object can only be used in certain kinds of expressions.
- Besides the need for type-checking rules on $T_\Omega(X)$ to prevent illegal constructor operations,
  - rules are necessary to ensure that λ-applications occur only between terms of appropriate types in order to remain meaningful.
  - rules are necessary to ensure that all terms have clearly defined types at compile-time so that there are no run-time type violations.

TL: A Language of Simple Types

Consider the following language of types (in fully parenthesized form) defined over an infinite collection $'\mathtt{a} \in TV$ of type variables, disjoint from the set of variables. We also have two type constants int and bool.

$$\sigma, \tau ::= \mathtt{int} \mid \mathtt{bool} \mid {'\mathtt{a}} \in TV \mid (\sigma * \tau) \mid (\sigma \rightarrow \tau)$$

Notes.

- int and bool are *type constants*.
- In any type expression τ, $TVar(\tau)$ is the set of type variables
- $*$ is the product operation on types and
- $\rightarrow$ is the function operator on types.
- We require $*$ because of the possibility of defining functions of various kinds of arities in $\Lambda_\Omega(X)$.

TL: Precedence and Associativity

- **Precedence.** We assume $*$ has a higher precedence than $\rightarrow$.
- **Associativity.**
  - $*$ is *left associative* whereas
  - $\rightarrow$ is *right associative*

Type-inference Rules: Infrastructure

The question of assigning types to complicated expressions which may have variables in them still remains to be addressed.

Type inferencing. Can be done using type assignment rules, by a recursive travel of the abstract syntax tree.

Free variables (names) are already present in the *environment* (symbol table).

Constants and Constructors. May have their types either pre-defined or there may be axioms assigning them types.

Bound variables. May be necessary to introduce "fresh" type variables in the environment.

Type Inferencing: Infrastructure

The elementary typing defined previously (§16.3.1) for the elementary expressions of FL does not suffice

1. in the presence of λ abstraction and application, which allow for higher-order functions to be defined
2. in the presence of polymorphism, especially when we do not want to unnecessarily decorate expressions with their types.

Type Assignment: Infrastructure

- Assume Γ is the environment[a] (an association list) which may be looked up to determine the types of individual names. For each variable $x \in X$, $\Gamma(x)$ yields the type of x i.e. $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$.
- For each (sub-)expression in $\Lambda_{RecFL}(X)$ we define a set C of *type constraints* of the form $\sigma = \tau$, where T is the set of type variables used in C.
- The type constraints are defined by induction on the structure of the expressions in the language $\Lambda_{RecFL}(X)$.
- The expressions of $\Lambda_{RecFL}(X)$ could have free variables. The type of the expression would then depend on the types assigned to the free variables. This is a simple kind of polymorphism.
- It may be necessary to generate new type variables as and when required during the process of inferencing and assignment.

[a] usually a part of the symbol table

Constraint Typing Relation

Definition 16.10 *For each term* $L \in \Lambda_{RecFL}(X)$ *the* **constraint typing relation** *is of the form*

$$\Gamma \vdash L : \tau \;\; \triangleright_T \;\; C$$

where

- Γ *is called the context*[a] *and defines the stack of assumptions*[b] *that may be needed to assign a type (expression) to the (sub-)expression* L.
- τ *is the type(-expression) assigned to* L
- C *is the set of constraints*
- T *is the set of "fresh" type variables used in the (sub-)derivations*

[a] usually in the symbol table
[b] including new type variables

Typing axioms: Basic 1

The following axioms (c.f Typing FL0 Expressions) may be either predefined or applied during the scanning and parsing phases of the compiler to assign types to the individual tokens and thus create an *initial* type environment Γ_0.

Z $\dfrac{}{\Gamma \vdash \texttt{Z} : \underline{\texttt{int}} \quad \rhd_{\emptyset} \quad \emptyset}$

T $\dfrac{}{\Gamma \vdash \texttt{T} : \underline{\texttt{bool}} \quad \rhd_{\emptyset} \quad \emptyset}$

F $\dfrac{}{\Gamma \vdash \texttt{F} : \underline{\texttt{bool}} \quad \rhd_{\emptyset} \quad \emptyset}$

S $\dfrac{}{\Gamma \vdash \texttt{S} : \underline{\texttt{int}} \rightarrow \underline{\texttt{int}} \quad \rhd_{\emptyset} \quad \emptyset}$

P $\dfrac{}{\Gamma \vdash \texttt{P} : \underline{\texttt{int}} \rightarrow \underline{\texttt{int}} \quad \rhd_{\emptyset} \quad \emptyset}$

IZ $\dfrac{}{\Gamma \vdash \texttt{IZ} : \underline{\texttt{int}} \rightarrow \underline{\texttt{bool}} \quad \rhd_{\emptyset} \quad \emptyset}$

GTZ $\dfrac{}{\Gamma \vdash \texttt{GTZ} : \underline{\texttt{int}} \rightarrow \underline{\texttt{bool}} \quad \rhd_{\emptyset} \quad \emptyset}$

Typing axioms: Basic 2

$$\textbf{ITEI}\quad \frac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}}*\underline{\mathtt{int}}*\underline{\mathtt{int}}\rightarrow\underline{\mathtt{int}} \quad \rhd_{\emptyset} \ \emptyset}$$

$$\textbf{ITEB}\quad \frac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}}*\underline{\mathtt{bool}}*\underline{\mathtt{bool}}\rightarrow\underline{\mathtt{bool}} \quad \rhd_{\emptyset} \ \emptyset}$$

Notice that the constructor `ITE` is *overloaded* and actually is two constructors `ITEI` and `ITEB`. Which constructor is actually used will depend on the context and the type-inferencing mechanism.

Type Rules: Variables and Abstraction

$$\textbf{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x) \quad \rhd_{\emptyset} \quad \emptyset}$$

$$\textbf{Abs}\quad \frac{\Gamma, x : \sigma \vdash L : \tau \quad \rhd_{T} \quad C}{\Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \quad \rhd_{T} \quad C}$$

Type Rules: Application

$$\textbf{App}\quad \frac{\Gamma \vdash L : \sigma \;\; \rhd_{T_1} \;\; C_1 \qquad \Gamma \vdash M : \tau \;\; \rhd_{T_2} \;\; C_2}{\Gamma \vdash (L \;\; M) : {}'\mathtt{a} \;\; \rhd_{T'} \;\; C'} \quad \textbf{(Conditions 1. and 2.)}$$

where

- **Condition 1.** $T_1 \cap T_2 = T_1 \cap TVar(\tau) = T_2 \cap TVar(\sigma) = \emptyset$
  Condition 2. ${}'\mathtt{a} \notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2)$.
- $T' = T_1 \cup T_2 \cup \{{}'\mathtt{a}\}$
- $C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow {}'\mathtt{a}\}$

Example 16.11 *Consider the following simple combinator* $\lambda x[\lambda y[\lambda z[(x\ \ (y\ \ z))]]]$ *which defines the function composition operator. Since there are three bound variables* x, y *and* z *we begin with an initial assumption* $\Gamma = x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c}$ *which assign arbitrary types to the bound variables, represented by the type variables* $\mathtt{'a}$, $\mathtt{'b}$ *and* $\mathtt{'c}$ *respectively. Note however, that since it has no free variables, its type does not depend on the types of any variables. We expect that at the end of the proof there would be no assumptions.Our inference for the type of the combinator then proceeds as follows.*

1. $x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c} \vdash x : \mathtt{'a} \ \ \triangleright_{\emptyset} \ \ \emptyset$ *(***Var***)*
2. $x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c} \vdash y : \mathtt{'b} \ \ \triangleright_{\emptyset} \ \ \emptyset$ *(***Var***)*
3. $x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c} \vdash z : \mathtt{'c} \ \ \triangleright_{\emptyset} \ \ \emptyset$ *(***Var***)*
4. $x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c} \vdash (y\ \ z) : \mathtt{'d} \ \ \triangleright_{\{\mathtt{'d}\}} \ \ \{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}\}$ *(***App***)*
5. $x : \mathtt{'a}, y : \mathtt{'b}, z : \mathtt{'c} \vdash (x\ \ (y\ \ z)) : \mathtt{'e} \ \ \triangleright_{\{\mathtt{'d},\mathtt{'e}\}} \ \ \{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}, \mathtt{'a} = \mathtt{'d}{\to}\mathtt{'e}\}$ *(***App***)*
6. $x : \mathtt{'a}, y : \mathtt{'b} \vdash \lambda z[(x\ \ (y\ \ z))] : \mathtt{'c}{\to}\mathtt{'e} \ \ \triangleright_{\{\mathtt{'d},\mathtt{'e}\}} \ \ \{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}, \mathtt{'a} = \mathtt{'d}{\to}\mathtt{'e}\}$ *(***Abs***)*
7. $x : \mathtt{'a} \vdash \lambda y[\lambda z[(x\ \ (y\ \ z))]] : \mathtt{'b}{\to}\mathtt{'c}{\to}\mathtt{'e} \ \ \triangleright_{\{\mathtt{'d},\mathtt{'e}\}} \ \ \{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}, \mathtt{'a} = \mathtt{'d}{\to}\mathtt{'e}\}$ *(***Abs***)*
8. $\vdash \lambda x[\lambda y[\lambda z[(x\ \ (y\ \ z))]]] : \mathtt{'a}{\to}\mathtt{'b}{\to}\mathtt{'c}{\to}\mathtt{'e} \ \ \triangleright_{\{\mathtt{'d},\mathtt{'e}\}} \ \ \{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}, \mathtt{'a} = \mathtt{'d}{\to}\mathtt{'e}\}$ *(***Abs***)*

Hence $\lambda x[\lambda y[\lambda z[(x\ \ (y\ \ z))]]] : \mathtt{'a}{\to}\mathtt{'b}{\to}\mathtt{'c}{\to}\mathtt{'e}$ *subject to the constraints given by* $\{\mathtt{'b} = \mathtt{'c}{\to}\mathtt{'d}, \mathtt{'a} = \mathtt{'d}{\to}\mathtt{'e}\}$ *which yields* $\lambda x[\lambda y[\lambda z[(x\ \ (y\ \ z))]]] : (\mathtt{'d}{\to}\mathtt{'e}){\to}(\mathtt{'c}{\to}\mathtt{'d}){\to}\mathtt{'c}{\to}\mathtt{'e}$

Principal Type Schemes

Definition 16.12 *A solution for* $\Gamma \vdash L : \tau \;\; \triangleright_T \;\; C$ *is a pair* $\langle S, \sigma \rangle$ *where* S *is a substitution of type variables in* τ *such that* $S(\tau) = \sigma$*.*

- The rules yield a *principal type scheme* for each well-typed applied λ-term.
- The term is *ill-typed* if there is no solution that satisfies the constraints.
- Any substitution of the type variables which satisfies the constraints C is an instance of the most general polymorphic type that may be assigned to the term.

Exercise 16.4

1. *The language has several constructors which behave like functions. Derive the following rules for terms in* $T_\Omega(X)$ *from the* *basic typing axioms* *and the rule* **App**.

$$\textbf{Sx}\quad \frac{\Gamma \vdash t : \tau \quad \rhd_T \quad C}{\Gamma \vdash (\texttt{S} \quad t) : \underline{\texttt{int}} \quad \rhd_T \quad C \cup \{\tau = \underline{\texttt{int}}\}}$$

$$\textbf{Px}\quad \frac{\Gamma \vdash t : \tau \quad \rhd_T \quad C}{\Gamma \vdash (\texttt{P} \quad t) : \underline{\texttt{int}} \quad \rhd_T \quad C \cup \{\tau = \underline{\texttt{int}}\}}$$

$$\textbf{IZx}\quad \frac{\Gamma \vdash t : \tau \quad \rhd_T \quad C}{\Gamma \vdash (\texttt{IZ} \quad t) : \underline{\texttt{bool}} \quad \rhd_T \quad C \cup \{\tau = \underline{\texttt{int}}\}}$$

$$\textbf{GTZx}\quad \frac{\Gamma \vdash t : \tau \quad \rhd_T \quad C}{\Gamma \vdash (\texttt{GTZ} \quad t) : \underline{\texttt{bool}} \quad \rhd_T \quad C \cup \{\tau = \underline{\texttt{int}}\}}$$

$$\textbf{ITEx}\quad \frac{\begin{array}{c}\Gamma \vdash t : \sigma \quad \rhd_T \quad C \\ \Gamma \vdash t_1 : \tau \quad \rhd_{T_1} \quad C_1 \\ \Gamma \vdash t_0 : \upsilon \quad \rhd_{T_0} \quad C_0\end{array}}{\Gamma \vdash (\texttt{ITE} \quad \langle t, t_1, t_0\rangle) : \tau \quad \rhd_{T'} \quad C'} \quad (T \cap T_1 = T_1 \cap T_0 = T_0 \cap T = \emptyset)$$

where $T' = T \cup T_1 \cup T_0$ *and* $C' = C \cup C_1 \cup C_0 \cup \{\sigma = \underline{\texttt{bool}}, \tau = \upsilon\}$

2. *Use the rules to define the type of the combinators* K *and* S*?*

3. *How would you define a type assignment for the recursive function* addc *defined by equation (68).*

4. *Prove that the terms,* $\omega = \lambda x[(x \;\; x)]$ *and* $\Omega = (\omega \;\; \omega)$ *are ill-typed.*

5. *Are the following well-typed or ill-typed? Prove your answer.*

   (a) $(\mathsf{K} \;\; \mathsf{S})$

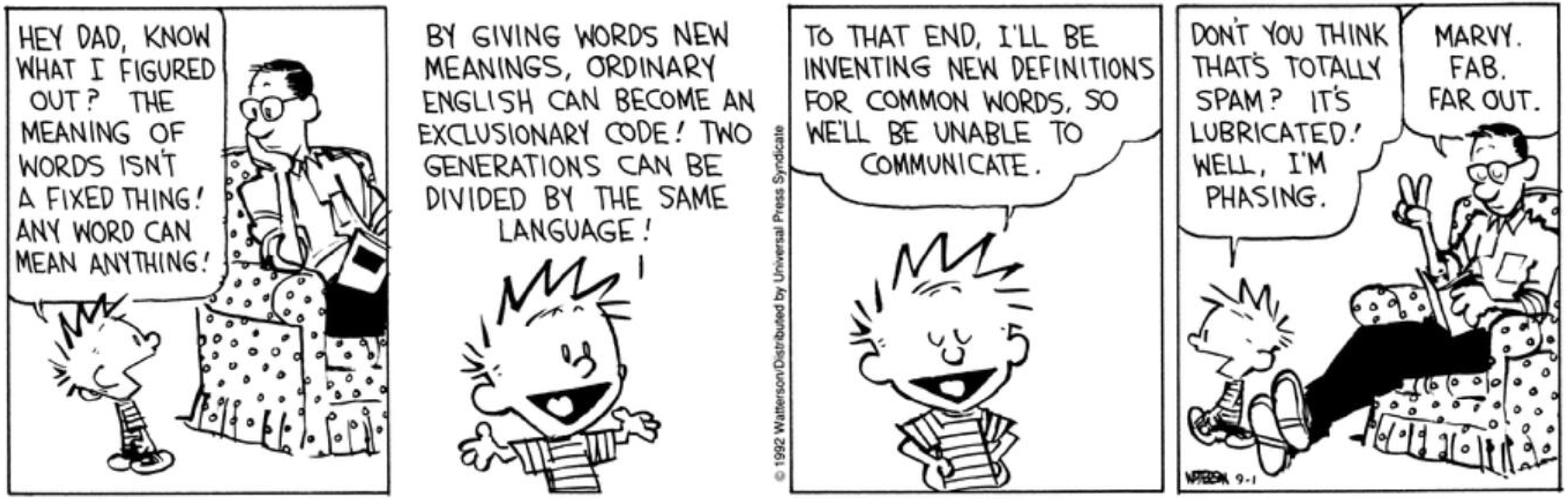
   (b) $((\mathsf{K} \;\; \mathsf{S}) \;\; \omega)$

   (c) $(((\mathsf{S} \;\; \mathsf{K}) \;\; K) \;\; \omega)$

   (d) $(\mathtt{ITE} \;\; \langle(\mathtt{IZ} \;\; x), \mathtt{T}, (\mathsf{K} \;\; \mathsf{x})\rangle)$

Formal Semantics of Languages

Calvin and Hobbes by Bill Watterson for September 01, 1992



The Concept of Environment

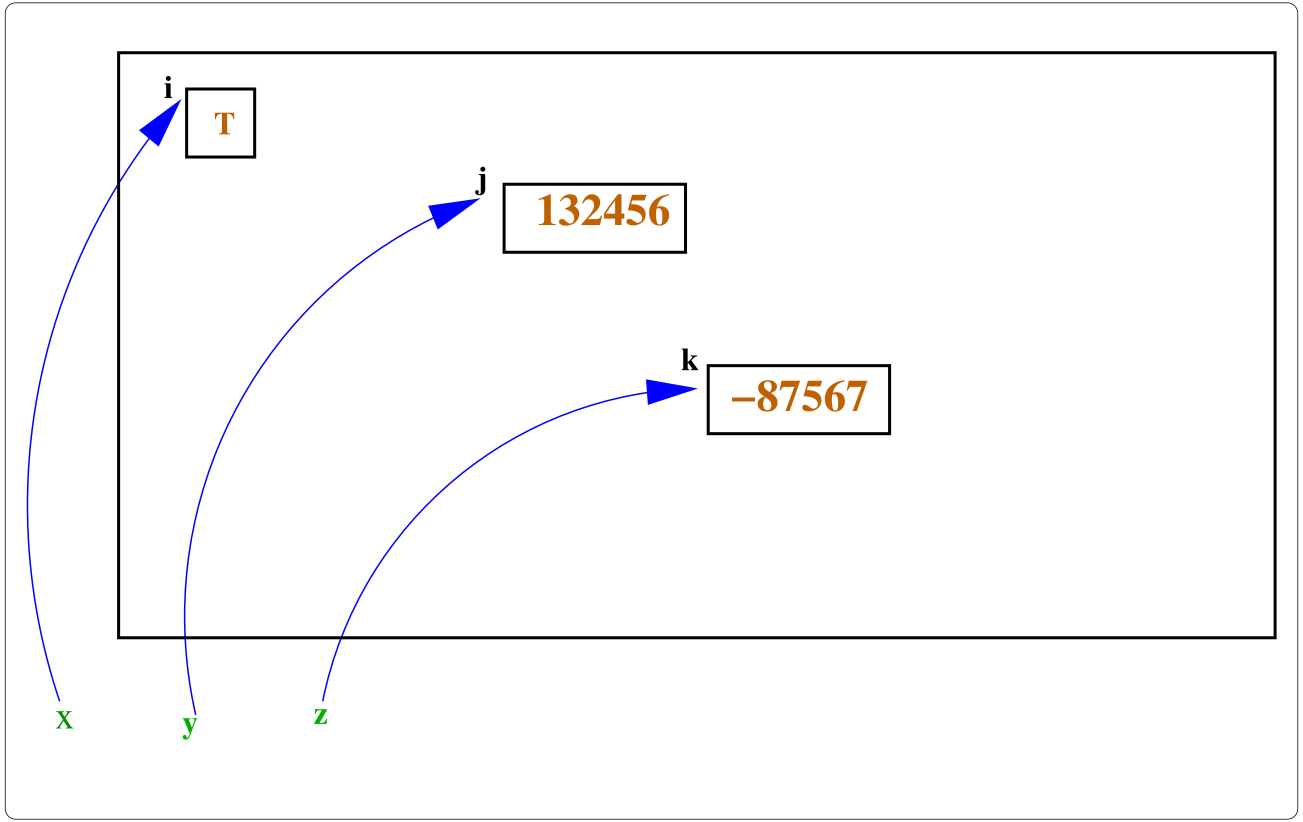
- Any imperative language indirectly exposes the *memory* (or *store*) to the user for manipulation.
- Memory (or store) is a set Loc of locations used to store the values of variables.
- Unlike in an actual computer, we do not consider all memory locations to be of the same size and shape. It is necessary to be able to associate a location to be a single cell that can store even a complex value of the appropriate type.
- Each variable in an imperative program is assigned a location.
- The *environment* $\gamma : X \rightarrow Loc$ is an association of (imperative) variables to locations and $Env = \{\gamma \mid \gamma : X \rightarrow Loc\}$ denotes the set of all environments.

The Concept of State

- $Values$ is the set of values that may be stored in memory. In general values is a disjoint union of various sets of values. In the case of the simple language FL0, $Values = (\underline{\texttt{int}} \uplus \underline{\texttt{bool}})$[a].
- We define the store to be a (partial) function from Loc to the set $Values$ of possible values that may be stored in memory. $\sigma : Loc \rightharpoonup Values$. $Stores = \{\sigma \mid \sigma : Loc \rightharpoonup Values\}$ is the set of possible stores.
- The *(dynamic) state* of a program is defined by the pair $(\gamma, \sigma) \in States = Env \times Stores$.

[a]The use of $\uplus$ rather than $\cup$ ensures that for each element in the set $Values$ it is possible to identify which component set it comes from.

State: Illustration



- **l-values.** $\gamma(x) = \mathbf{i} : \underline{\texttt{bool}}$, $\gamma(y) = \mathbf{j} : \underline{\texttt{int}}$, $\gamma(z) = \mathbf{k} : \underline{\texttt{int}}$
- **r-values.** $\sigma(\mathbf{i}) = T : \underline{\texttt{bool}}$, $\sigma(\mathbf{j}) = 132456 : \underline{\texttt{int}}$, $\sigma(\mathbf{k}) = -87567 : \underline{\texttt{int}}$

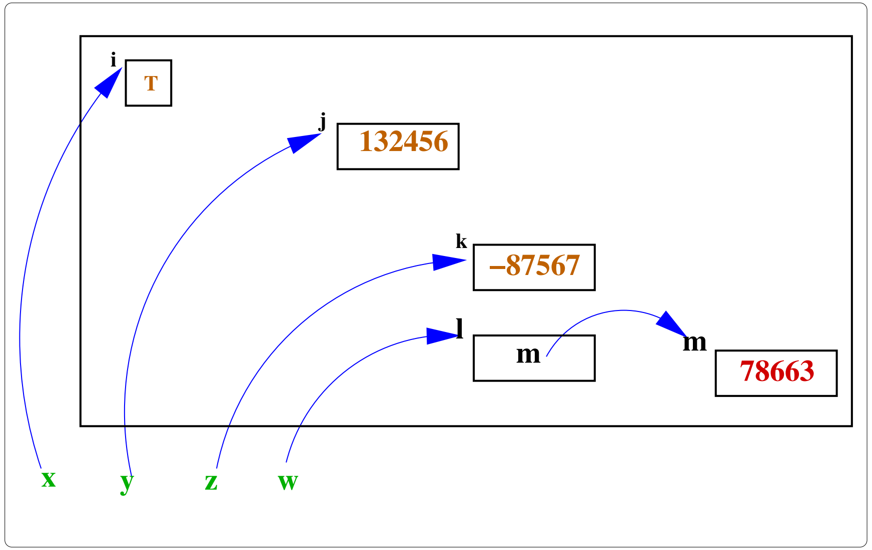
References in Languages

ML-like impure functional languages

- have an explicit polymorphic $'a$ `ref` type constructor. Hence x : `bool` `ref`, y, z : `int` `ref` and x is a named reference to the location $\mathbf{i}$
- have an explicit unary dereferencing operator ! to read the value contained in the location referenced by x, i.e. $!x = \sigma(\mathbf{i})$.
- The actual locations however are not directly visible.

C-like imperative languages are not as fussy as the ML-like languages. C (and C++) even treats locations only as integers and allows integer operations to be preformed on them!

l-values and r-values



- $\mathbf{l}$ is the l-value of w i.e $\gamma(w) = \mathbf{l} \in Loc$
- $\mathbf{m}$ is the r-value of w i.e. $\sigma(\gamma(w)) = !w = \mathbf{m} \in Loc$
- $\mathbf{m}$ is also an l-value since $!w : \underline{\texttt{int}}\ \underline{\texttt{ref}}$
- $!(!w) = \mathbf{78663} : \underline{\texttt{int}}$ is the r-value of $!w$

17.0.1. l-values, r-values, aliasing and indirect addressing

The terms "l-value" (for "left-value") and "r-value" (for "right-value") come from the practice in most imperative languages of writing assignment commands by overloading the variable name to denote both its address ($\gamma(x)$) in *Loc* as well as the value $\sigma(\gamma(x))$ stored in memory. Consider the example,

- $x := x + y$ (Pascal)
- $x = x + y$ (C, C++, Java, Python, Perl)

The occurrence of "x" on the left-hand side of the assignment command denotes a location $\gamma(x)$ whereas the occurrences of "x" and "y" on the right-hand-side of the assignment denote the values $\sigma(\gamma(x))$ and $\sigma(\gamma(y))$ respectively. The term "dereferencing" is used to denote the action of "reading" the value stored in a location.

- This notation for assignment becomes a source of tremendous confusion when locations are also valid values, as in the case of indirect addressing (look at w) and may be manipulated.
- The confusion is further exacerbated when locations are also integers indistinguishable from the integers stored in the locations. The result of dereferencing an integer variable may be one of the following.
  - An invalid location leading to a segmentation fault. For instance, the integer could be negative or larger than any valid memory address.

– Another valid location with an undefined value or with a value defined previously when the location was assigned to some other variable in a different job. This could lead to puzzling results in the current program.
– Another valid location which is already the address of a variable in the program (leading to an aliasing totally unintended by the programmer). This could also lead to puzzling results in the current program.

Modern impure functional languages (which have strong-typing facilties) usually clearly distinguish between locations and values as different types. Hence every imperative variable represents only an l-value. Its r-value is obained by applying a dereferencing operation (the prefix operation !). Hence the same assignment command in ML-like languages would be written

- $x := !x + !y$ (ML and OCaml)

The following interactive ML session illustrates aliasing and the effect on the aliased variables.

```
Standard ML of New Jersey v110.76 [built: Tue Oct 22 14:04:11 2013]
- val u = ref 1;
val u = ref 1 : int ref
- val v = u; (* u and v are aliases for the same location *)
val v = ref 1 : int ref
- v := !v+1;
val it = () : unit
- !u;
```

```
val it = 2 : int
- !v;
val it = 2 : int
- v := !v+1;
val it = () : unit
- !u;.val it = 3 : int
- !v;
val it = 3 : int
-
```

The following ML-session illustrates indirect addressing (and if you get confused, don't come to me, I am confused too; confusion is the price we pay for indiscriminate modification of state).

```
Standard ML of New Jersey v110.76 [built: Tue Oct 22 14:04:11 2013]
- val x = ref (ref 0);
val x = ref (ref 0) : int ref ref
- val y = !x;
val y = ref 0 : int ref
- val z = ref y;
val z = ref (ref 0) : int ref ref
- y := !y+1;
val it = () : unit
```

```
- !y;
val it = 1 : int
- !z;
val it = ref 1 : int ref
- !(!z);
val it = 1 : int
- !(!x);
val it = 1 : int
-
```

With the introduction of references, the store may have locations whose r-value is another location. In such a situation. Further if locations themselves are addressed by (non-negative) integer values, there woudl be an obvious clash with normal integer values of simple integer variables. Hence we woudl have to address this problem as follows.

- Allow locations to be part of the set of values and
- In the interests of maintaining a strong type discipline, distinguish between sets of values that may not be disjoint.

We would then address the above problems by (re-)defining $Values$ to include locations too. Thus with the inclusion of references we have $Values = \underline{\texttt{int}} \uplus \underline{\texttt{bool}} \uplus Loc$.

Semantics of Expressions of FL(X)

- Consider the language FL(X). Instead of the δ-rules defined earlier, we assume that these terms are evaluated on a hardware which can represent `int` and `bool`.
- Assume `int` is the hardware representation of the integers and $\texttt{bool} = \{\texttt{T}, \texttt{F}\}$.
- We assume that every (sub-)expression in the language has been typed with a unique type attribute.

Functional Semantics of FL(X)

The language FL(X) is a pure expression language that is

- free of side-effects and
- free of references

Each expression $e \in WT_\Omega(X)$ denotes a value from the set $Values$ depending on the state it is evaluated in. The meaning of $e \in WT_\Omega(X)$ is given by a function

$$\mathscr{E}[\![e]\!] : States \to Values \tag{73}$$

and is therefore a function of the state. Hence

$$\mathscr{E} : WT_\Omega(X) \to States \to Values \tag{74}$$

The semantics given earlier now reads as follows (with the valuation v replaced by a state $s = (\gamma, \sigma)$).

A New Functional Semantics of FL(X)

$$\mathscr{E}[x]\ s \stackrel{df}{=} \sigma(\gamma(x)),\ \text{ for }\ s = (\gamma, \sigma) \tag{75}$$

$$\mathscr{E}[\mathtt{T}]\ s \stackrel{df}{=} true \tag{76}$$

$$\mathscr{E}[\mathtt{F}]\ s \stackrel{df}{=} false \tag{77}$$

$$\mathscr{E}[\mathtt{Z}]\ s \stackrel{df}{=} 0 \tag{78}$$

$$\mathscr{E}[(\mathtt{P}\ \ t)]\ s \stackrel{df}{=} (\mathscr{E}[t]\ \ s) - 1 \tag{79}$$

$$\mathscr{E}[(\mathtt{S}\ \ t)]\ s \stackrel{df}{=} (\mathscr{E}[t]\ \ s) + 1 \tag{80}$$

$$\mathscr{E}[(\mathtt{IZ}\ \ t)]\ s \stackrel{df}{=} (\mathscr{E}[t]\ \ s) = 0 \tag{81}$$

$$\mathscr{E}[(\mathtt{GTZ}\ \ t)]\ s \stackrel{df}{=} (\mathscr{E}[t]\ \ s) > 0 \tag{82}$$

$$\mathscr{E}[(\mathtt{ITE}\ \ \langle t, t_1, t_0\rangle)]\ s \stackrel{df}{=} \begin{cases} \mathscr{E}[t_1]\ \ s \text{ if } \mathscr{E}[t]\ \ s \\ \mathscr{E}[t_0]\ \ s \text{ if not } \mathscr{E}[t]\ \ s \end{cases} \tag{83}$$

Evaluating FL(X) on a machine

- We previously treated FL(X) as simply a data-type and gave δ-rules. See
  - lemma 16.3,
  - δ-nf for the conditonal,
  - the zero test
  - test for positivity
- Here we define a deterministic evaluation mechanism $\longrightarrow_e$ on a more realistic hardware which supports integers and booleans
- The normal forms on this machine would have to be appropriate integer and boolean constants as represented in the machine.
- We define an expression evaluation relation $\longrightarrow_e$ such that

$$\longrightarrow_e \subseteq (States \times WT_\Omega(X)) \times (States \times (\mathtt{int} \cup \mathtt{bool}))$$

Operational Semantics: Constants and Variables

> Unless there is a prior generally-accepted mathematical definition of a language at hand, who is to say whether a proposed implementation is correct?
>
> *Dana S. Scott (1969)*

Let $\sigma \in$ States be any state.

$$\mathbf{T}\ \frac{}{\gamma \vdash \langle \sigma, \mathtt{T} \rangle \longrightarrow_e \langle \sigma, \mathtt{T} \rangle}$$

$$\mathbf{F}\ \frac{}{\gamma \vdash \langle \sigma, \mathtt{F} \rangle \longrightarrow_e \langle \sigma, \mathtt{F} \rangle}$$

$$\mathbf{Z}\ \frac{}{\gamma \vdash \langle \sigma, \mathtt{Z} \rangle \longrightarrow_e \langle \sigma, \mathtt{0} \rangle}$$

$$\mathbf{x}\ \frac{}{\gamma \vdash \langle \sigma, x \rangle \longrightarrow_e \langle \sigma, \sigma(\gamma(x)) \rangle}$$

Operational Semantics: Integer-valued Expressions

$$\mathbf{P}\quad \frac{\gamma \vdash \langle \sigma, e\rangle \;\longrightarrow_e\; \langle \sigma, \mathtt{m}\rangle}{\gamma \vdash \langle \sigma, (\mathtt{P}\ \ e)\rangle \;\longrightarrow_e\; \langle \sigma, \mathtt{m} - 1\rangle} \quad (e, \mathtt{m} : \underline{\mathtt{int}})$$

$$\mathbf{S}\quad \frac{\gamma \vdash \langle \sigma, e\rangle \;\longrightarrow_e\; \langle \sigma, \mathtt{m}\rangle}{\gamma \vdash \langle \sigma, (\mathtt{S}\ \ e)\rangle \;\longrightarrow_e\; \langle \sigma, \mathtt{m} + 1\rangle} \quad (e, \mathtt{m} : \underline{\mathtt{int}})$$

Operational Semantics: Boolean-valued Expressions

$$\textbf{IZ0} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{m} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{IZ} \quad e) \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{F} \rangle} \quad (e, \mathtt{m} : \underline{\mathtt{int}}, \mathtt{m} <> \mathtt{0})$$

$$\textbf{IZ1} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{0} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{IZ} \quad e) \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{T} \rangle} \quad (e : \underline{\mathtt{int}})$$

$$\textbf{GTZ0} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{m} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{GTZ} \quad e) \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{F} \rangle} \quad (e, \mathtt{m} : \underline{\mathtt{int}}, \mathtt{m} <= \mathtt{0})$$

$$\textbf{GTZ1} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{m} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{GTZ} \quad e) \rangle \quad \longrightarrow_e \quad \langle \sigma, \mathtt{T} \rangle} \quad (e, \mathtt{m} : \underline{\mathtt{int}}, \mathtt{m} > \mathtt{0})$$

Operational Semantics: Conditional Expressions

$$\textbf{ITEI0}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathtt{F} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{ITE}\ \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_0 \rangle} \quad (e_1, e_0 : \underline{\mathtt{int}})$$

$$\textbf{ITEI1}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathtt{T} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{ITE}\ \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_1 \rangle} \quad (e_1, e_0 : \underline{\mathtt{int}})$$

$$\textbf{ITEB0}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathtt{F} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{ITE}\ \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_0 \rangle} \quad (e_1, e_0 : \underline{\mathtt{bool}})$$

$$\textbf{ITEB1}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathtt{T} \rangle}{\gamma \vdash \langle \sigma, (\mathtt{ITE}\ \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_1 \rangle} \quad (e_1, e_0 : \underline{\mathtt{bool}})$$

Reduction vs. Functional vs. Operational Semantics

Theorem 17.1 *For each term* $e \in WT_{\Omega}(X)$ *and state* $s = (\gamma, \sigma)$*, and valuation* $v = \sigma \circ \gamma$

$$\mathscr{M}[\![e]\!]\ v = \mathscr{E}[\![e]\!]\ s \tag{84}$$

$$\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_{e}\ \langle \sigma, \mathscr{E}[\![e]\!]\ s \rangle \tag{85}$$

■

Taken in conjunction with lemma 16.3, δ-nf for integers, δ-nf for the conditonal, the zero test and test for positivity essentially all the various semantics and computation rules may be proved mutually equivalent and conforming to our informal understanding of the language.

The expression language is purely functional since the evaluation of an expression in any state does not effect any change in the state.

WHILE: Big-Step Semantics

We are faced with the problem of variables which actually vary

Christopher Strachey (1967)

The WHILE language

- We initially define a simple language of *commands*.
- The expressions of the language are those of any term algebra $T_\Omega(X)$.
- We simply assume there is a well-defined relation $\longrightarrow_e$ for evaluating expressions that is proven correct.

State Changes or Side-Effects

- State changes are usually programmed by assignment commands which occur one location at a time.
- In the simple WHILE language side-effects do not occur except by *explicit* assignment commands.

Modelling a Side-Effect

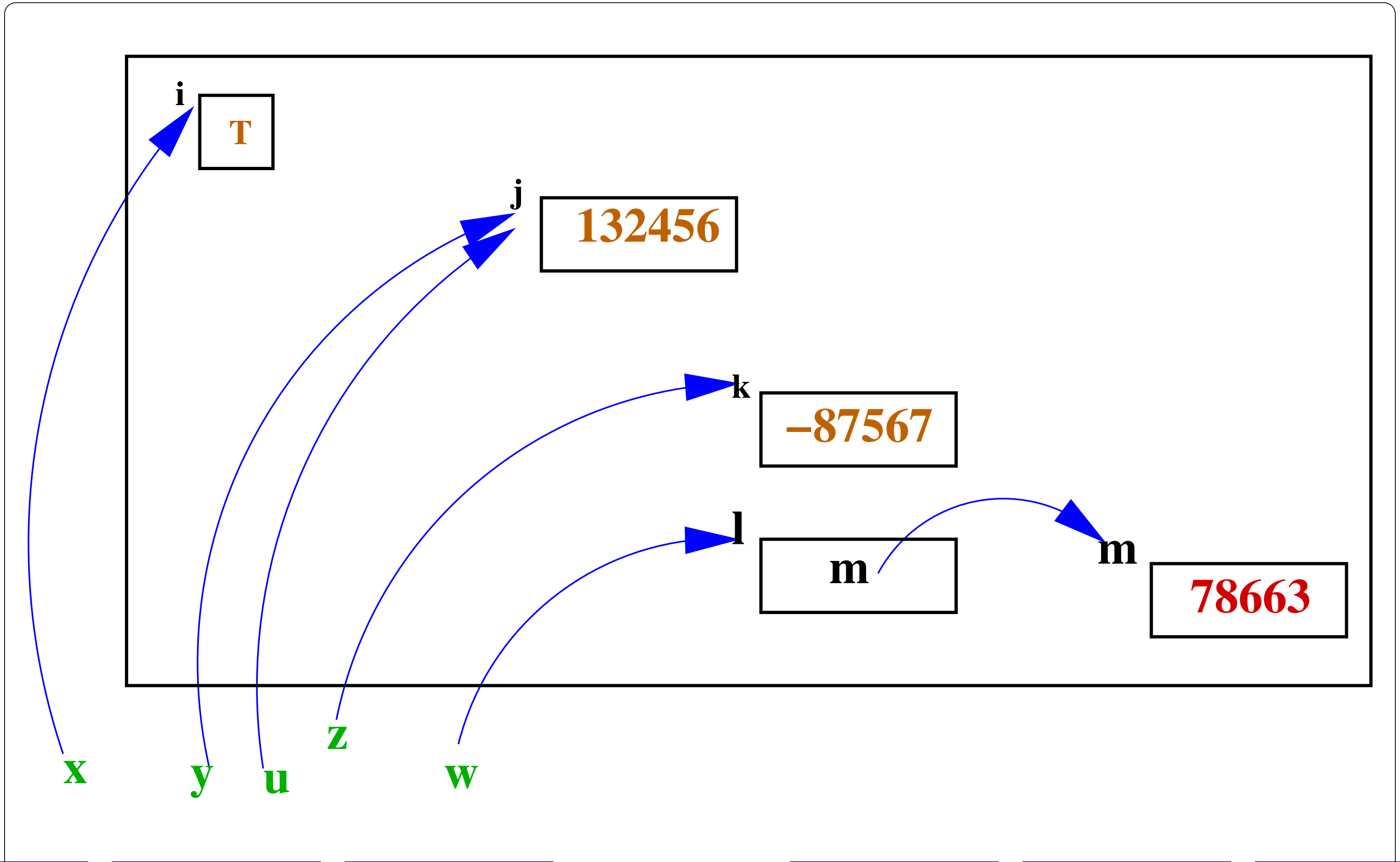
Given a store σ, a variable x such that $\gamma(x) = \ell$ and $\sigma(\ell) = a$, the state change effected by the assignment $x := b$ is a new store that is identical to σ except at the location $\gamma(x)$ which now contains the value b

$$\sigma' = [\gamma(x) \mapsto b]\sigma$$

i.e.

$$\sigma'(\ell) = \begin{cases} \sigma(\ell) & \text{if } \ell \neq \gamma(x) \\ b & \text{otherwise} \end{cases}$$

Definition 17.2 *Two (or more) variables are called* **aliases** *if they denote the same location (y and u in the figure below).*



The Commands of the WHILE Language

$$
\begin{array}{rcll}
c_0, c_1, c & ::= & \mathbf{skip} & \textbf{Skip} \\
& \mid & x := e & \textbf{Assgn} \\
& \mid & \{c_0\} & \textbf{Block} \\
& \mid & c_0; c_1 & \textbf{Seq} \\
& \mid & \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0\ \mathbf{fi} & \textbf{Cond} \\
& \mid & \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od} & \textbf{While}
\end{array}
$$

where e is either an integer or boolean expression in the language FL(X) with operational semantics as given before.

For any signature Ω and a set of variables X we denote the set of all commands over the well-typed expressions $WT_\Omega(X)$ by WHILE$_\Omega(X)$

Functional Semantics of WHILE

A New Functional Semantics of FL(X)

$$\mathscr{C} : \mathsf{WHILE}_\Omega(X) \to States \to States$$

For any $s = (\gamma, \sigma) \in States$,

$$\begin{aligned}
\mathscr{C}[\![\mathbf{skip}]\!]\ s &\stackrel{df}{=} s \\
\mathscr{C}[\![x := e]\!]\ s &\stackrel{df}{=} (\gamma, \sigma') \text{ where } \sigma' = [\gamma(x) \mapsto \mathscr{E}[\![e]\!]\ s]\sigma \\
\mathscr{C}[\![\{c\}]\!]\ s &\stackrel{df}{=} \mathscr{C}[\![c]\!]\ s \\
\mathscr{C}[\![c_0; c_1]\!]\ s &\stackrel{df}{=} \mathscr{C}[\![c_1]\!]\ (\mathscr{C}[\![c_0]\!]\ s) \\
\mathscr{C}[\![\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0\ \mathbf{fi}]\!]\ s &\stackrel{df}{=} \begin{cases} \mathscr{C}[\![c_1]\!]\ s & \text{if } \mathscr{E}[\![e]\!]\ s = true \\ \mathscr{C}[\![c_0]\!]\ s & \text{if } \mathscr{E}[\![e]\!]\ s = false \end{cases}
\end{aligned}$$

Rewriting Functional Semantics of WHILE

We may express the same semantics as follows using the λ notation as follows.

$$\mathscr{C}[\![\mathbf{skip}]\!] \stackrel{df}{=} \lambda s[s] \tag{86}$$

$$\mathscr{C}[\![x := e]\!] \stackrel{df}{=} \lambda(\gamma, \sigma)[(\gamma, \sigma')] \tag{87}$$

$$\text{where } \sigma' = [\gamma(x) \mapsto \mathscr{E}[\![e]\!]\ (\gamma, \sigma)]\sigma$$

$$\mathscr{C}[\![\{c\}]\!] \stackrel{df}{=} \lambda s[\mathscr{C}[\![c]\!]\ s] \tag{88}$$

$$\mathscr{C}[\![c_0; c_1]\!] \stackrel{df}{=} \lambda s[\mathscr{C}[\![c_1]\!]\ (\mathscr{C}[\![c_0]\!]\ s)] \tag{89}$$

$$\mathscr{C}[\![\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0\ \mathbf{fi}]\!] \stackrel{df}{=} \lambda s\left[\begin{cases} \mathscr{C}[\![c_1]\!]\ s & \text{if } \mathscr{E}[\![e]\!]\ s = true \\ \mathscr{C}[\![c_0]\!]\ s & \text{if } \mathscr{E}[\![e]\!]\ s = false \end{cases}\right] \tag{90}$$

Semantics-induced Equality on Commands

Before defining the meaning of the command **while** e **do** c **od** it is only appropriate that we mention some trivial properties of the language regarded as an algebra (see theorem 17.4).

As in the case of the idenitites of the expression language, the semantics of commands induces an equality relation on commands.

Definition 17.3 *For any two commands* c_1 *and* c_2*,*

1. $\mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$ *if and only if for all* $s \in States$, $\mathscr{C}[\![c_1]\!]\ s = \mathscr{C}[\![c_2]\!]\ s$
2. $c_1 =_c c_2$ *if and only if* $\mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$

$\mathsf{WHILE}_\Omega(X)$ is a Monoid

Theorem 17.4 $\mathsf{WHILE}_\Omega(X)$ *is a monoid under sequencing* $(;)$ *with* $\mathbf{skip}$ *as the identity element. That is, for all commands* $c_0, c_1, c_2 \in \mathsf{WHILE}_\Omega(X)$,

Closure of sequencing. $c_1; c_2 \in \mathsf{WHILE}_\Omega(X)$.

Associativity of sequencing. $c_0; \{c_1; c_2\} =_c \{c_0; c_1\}; c_2$

Identity of sequencing . $c_0; \mathbf{skip} =_c c_0 =_c \mathbf{skip}; c_0$

■

We could easily add (for convenience) an "**if** e **then** c **fi**" command to the language (see section 4.4) with the meaning

$$\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{fi} =_c \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ \mathbf{skip}\ \mathbf{fi} \tag{91}$$

The While Loop: Identities

Intuitively the while-loop satisfies the following identities

$$
\begin{aligned}
& \mathscr{C}[\![\textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}]\!]\ s \\
& = \begin{cases} \mathscr{C}[\![c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}]\!]\ s & \text{if } \mathscr{E}[\![e]\!]\ s = true \\ s & \text{if } \mathscr{E}[\![e]\!]\ s = false \end{cases} \\
\{\text{By equation (89)}\} & = \begin{cases} \mathscr{C}[\![\textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}]\!]\ (\mathscr{C}[\![c]\!]\ s) & \text{if } \mathscr{E}[\![e]\!]\ s = true \\ s & \text{if } \mathscr{E}[\![e]\!]\ s = false \end{cases} \\
\{\text{By equation (90)}\} & = \mathscr{C}[\![\textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{else skip fi}]\!]\ s \\
\{\text{By equation (91)}\} & = \mathscr{C}[\![\textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{fi}]\!]\ s
\end{aligned}
$$

Loop Unrolling

Hence by the last identity we get

$$\textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} =_c \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{fi} \qquad (92)$$

The last identity is actually a form of "recursion unfolding" (8) and is called "loop unrolling" since it may be unrolled as many times as we require.

$$\begin{array}{ll} & \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \\ =_c & \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{fi} \\ =_c & \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{fi}\}\ \textbf{fi} \\ =_c & \textbf{if}\ e\ \textbf{then}\ \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{if}\ e\ \textbf{then}\ \{c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od}\}\ \textbf{fi}\}\ \textbf{fi}\ \textbf{fi} \\ =_c & \cdots \end{array}$$

The Semantics of the while command

From the above identities and taking inspiration from the claims 16.8, 16.9 and the fixed-point theorem and corollary 13.6 it follows that

$$\mathscr{C}[\![\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}]\!] \stackrel{df}{=} \mathsf{Y}_{\mathsf{C}}\ (\lambda f\ \lambda s \left[\begin{cases} f\ (\mathscr{C}[\![c]\!]\ s) & \text{if } \mathscr{E}[\![e]\!]\ s = true \\ s & \text{if } \mathscr{E}[\![e]\!]\ s = false \end{cases}\right]) \tag{93}$$

17.3. Loop unrolling

We know that each sentence of a language has to be finite. However the **while do od**-loop may be considered a finite representation of an "infinite" program.

Example 17.5 *Consider the program*

$$x := \mathtt{Z};\ \mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}$$

By the semantics it is clear that this program does not ever terminate. The value of x *is given by* $\sigma(\gamma(x))$*. Starting from* $x := \mathtt{Z}$*, and abbreviating the state to the value* $\sigma(\gamma(x))$ *we have*

$$\begin{array}{cl} & \mathscr{C}[\![\mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}]\!]\ 0 \\ = & \mathscr{C}[\![\mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}]\!]\ 1 \\ = & \mathscr{C}[\![\mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}]\!]\ 2 \\ = & \mathscr{C}[\![\mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}]\!]\ 3 \\ = & \mathscr{C}[\![\mathbf{while}\ \mathtt{T}\ \mathbf{do}\ x := (\mathtt{S}\ \ x)\ \mathbf{od}]\!]\ 4 \\ \vdots & \vdots \\ = & \cdots \end{array}$$

ad infinitum.

Example 17.6 *Consider the program*

$$\textbf{while } \mathtt{T} \textbf{ do skip od}$$

By equation (17.2) the loop unrolls infinitely and even though **skip** *does not alter the state in any way, it does not terminate either. Hence in terms of the operational semantics of the rule this command never actually yields a state. Taking a cue from the previous example, we may think of this program as representing a function that is undefined in all states. Letting* $\perp$ *denote the undefined state we have*

$$\mathscr{C}[\![\textbf{while } \mathtt{T} \textbf{ do skip od}]\!]\ s = \perp$$

Operational Semantics: Basic Commands

$$\textbf{Skip} \quad \frac{}{\gamma \vdash \langle \sigma, \textbf{skip} \rangle \ \longrightarrow_c^1 \ \sigma}$$

$$\textbf{Assgn} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ m}{\gamma \vdash \langle \sigma, x := e \rangle \ \longrightarrow_c^1 \ [\gamma(x) \mapsto m]\sigma}$$

Notes:

1. The **Skip** rule corresponds to any of the following:
   - a `noop`
   - the identity function or identity relation on states
   - a command which has no effect on states
2. The assignment is the only command in our language which creates a side-effect (actually changes state)

Operational Semantics: Blocks

We have defined a block as simply a command enclosed in braces. It is meant to delimit a (new) scope. Later we will see that there could be local declarations as well, in which case the semantics changes slightly to include a new scope

$$\boxed{\textbf{Block}\ \frac{\gamma \vdash \langle \sigma, c \rangle \ \longrightarrow_c^1 \ \sigma'}{\gamma \vdash \langle \sigma, \{c\} \rangle \ \longrightarrow_c^1 \ \sigma'}}$$

Operational Semantics: Sequencing

$$\textbf{Seq}\quad \frac{\begin{array}{c}\gamma \vdash \langle \sigma, c_0 \rangle \longrightarrow^1_c \sigma', \\ \gamma \vdash \langle \sigma', c_1 \rangle \longrightarrow^1_c \sigma''\end{array}}{\gamma \vdash \langle \sigma, c_0; c_1 \rangle \longrightarrow^1_c \sigma''}$$

Notice that sequencing is precisely the composition of relations. If the relation $\longrightarrow^1_c$ is a function (in the case of our language it actually is a function), sequencing would then be a composition of functions

Operational Semantics: Conditionals

$$\textbf{Cond0}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \mathtt{F}, \qquad \gamma \vdash \langle \sigma, c_0 \rangle \longrightarrow_c^1 \sigma_0}{\gamma \vdash \langle \sigma, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0\ \mathbf{fi} \rangle \longrightarrow_c^1 \sigma_0}$$

$$\textbf{Cond1}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \mathtt{T}, \qquad \gamma \vdash \langle \sigma, c_1 \rangle \longrightarrow_c^1 \sigma_1}{\gamma \vdash \langle \sigma, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0\ \mathbf{fi} \rangle \longrightarrow_c^1 \sigma_1}$$

Selective Evaluation.
Notice again the effect of selective evaluation in the operational semantics of the conditional and again in the operational semantics of the while loop.

The While loop: Operational Rules

- We use the fact that the $\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}$ is really a form of recursion – actually it is a form of "*tail recursion*". Hence the execution behaviour of $\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}$ is exactly that of

$$\mathbf{if}\ e\ \mathbf{then}\ \{c; \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}\}\ \mathbf{else}\ \mathbf{skip}\ \mathbf{fi} \tag{94}$$

- The following rules may be derived from (94) using the rules for conditional, sequencing and skip (though the number of steps may not exactly correspond).

Operational Semantics: The While Loop

The While Loop: Identities

$$\textbf{While0}\quad \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \mathtt{F}}{\gamma \vdash \langle \sigma, \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma}$$

$$\textbf{While1}\quad \frac{\begin{array}{c}\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \mathtt{T}, \\ \gamma \vdash \langle \sigma, c \rangle \ \longrightarrow_c^1 \ \sigma', \\ \gamma \vdash \langle \sigma', \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma''\end{array}}{\gamma \vdash \langle \sigma, \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma''}$$

The While Loop: caveats

Notice that the above rules are applicable only if all commands are terminating! In particular,

1. the execution of the whole while loop needs to terminate. For this to happen it is necessary (though not sufficient) that
2. the execution of the body c also needs to terminate.

The effect of side-effects. In the above operational rules we have assumed that expression-evaluation has no side-effects i.e. there are no changes to the state of the program during or as result of expression evaluation. However many programming languages allow side-effects to global or non-local variables during expression evaluation. This does not significantly change the semantics, though it does very significantly change our ability to reason about such programs. In the presence of side-effects during expression-evalaution the semantics of commands would also changeas we show in the following modified semantics of commands. In particular we need to carry state information during expression evaluation.

$$\textbf{Assgn0}\ \frac{}{\gamma \vdash \langle \sigma, x := m \rangle \ \longrightarrow_c^1 \ [\gamma(x) \mapsto m]\sigma}$$

$$\textbf{Assgn1}\ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma', e' \rangle}{\gamma \vdash \langle \sigma, x := e \rangle \ \longrightarrow_c^1 \ \langle \sigma', x := e' \rangle}$$

$$\textbf{Cond0}\ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma', \mathtt{F} \rangle, \quad \gamma \vdash \langle \sigma', c_0 \rangle \ \longrightarrow_c^1 \ \sigma_0}{\gamma \vdash \langle \sigma, \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_0\ \textbf{fi} \rangle \ \longrightarrow_c^1 \ \sigma_0}$$

$$\textbf{Cond1}\ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma', \mathtt{T} \rangle, \quad \gamma \vdash \langle \sigma', c_1 \rangle \ \longrightarrow_c^1 \ \sigma_1}{\gamma \vdash \langle \sigma, \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_0\ \textbf{fi} \rangle \ \longrightarrow_c^1 \ \sigma_1}$$

$$\textbf{While0}\ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma', \mathtt{F} \rangle}{\gamma \vdash \langle \sigma, \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma'}$$

$$\textbf{While1}\ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma', \mathtt{T} \rangle, \quad \gamma \vdash \langle \sigma', c \rangle \ \longrightarrow_c^1 \ \sigma'', \quad \gamma \vdash \langle \sigma'', \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma'''}{\gamma \vdash \langle \sigma, \textbf{while}\ e\ \textbf{do}\ c\ \textbf{od} \rangle \ \longrightarrow_c^1 \ \sigma'''}$$

The rules of the other constructs viz. skip, blocks and sequencing, remain unchanged.

Local Declarations

We introduce declarations through a new syntactic category $Decls$ defined as follows:

$$
\begin{array}{rcl}
d_1, d_2, d \ ::= & \underline{\mathtt{int}}\ x \mid \underline{\mathtt{bool}}\ y \mid d_1; d_2 \\
c \ ::= & \cdots \mid \{d; c\}
\end{array}
$$

- Most languages insist on a "declaration before use" discipline,
- Declarations create "little new environments".
- Need to be careful about whether a variable is at all defined.
- Even if the l-value of a variable is defined, its r-value may not be defined. The rules for variables and assignments then need to be changed to the following.

Some changed rules

- We use the symbol $\perp$ to denote the undefined.
- We use $z \neq \perp$ to denote that z is well-defined.

$$\mathbf{x'} \quad \frac{}{\gamma \vdash \langle \sigma, x \rangle \longrightarrow_e \langle \sigma, \sigma(\gamma(x)) \rangle} \quad (\sigma(\gamma(x)) \neq \perp)$$

$$\mathbf{Assgn0'} \quad \frac{}{\gamma \vdash \langle \sigma, x := m \rangle \longrightarrow_c^1 [\gamma(x) \mapsto m]\sigma} \quad (\gamma(x) \neq \perp)$$

$$\mathbf{Assgn1'} \quad \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, e' \rangle}{\gamma \vdash \langle \sigma, x := e \rangle \longrightarrow_c^1 \langle \sigma, x := e' \rangle} \quad (\gamma(x) \neq \perp)$$

Declarations: Little Environments

The effect of a declaration is to create a little environment which is pushed onto the existing environment. The transition relation

$$\longrightarrow_d \subseteq ((Env \times Stores \times Decls) \times (Env \times Stores))$$

$$\textbf{int} - \textbf{x} \quad \frac{}{\gamma \vdash \langle \sigma, \texttt{int}\ x \rangle \longrightarrow_d \langle [x \mapsto l], [l \mapsto \perp]\sigma \rangle} \quad (l \notin Range(\gamma))$$

$$\textbf{bool} - \textbf{x} \quad \frac{}{\gamma \vdash \langle \sigma, \texttt{bool}\ x \rangle \longrightarrow_d \langle [x \mapsto l], [l \mapsto \perp]\sigma \rangle} \quad (l \notin Range(\gamma))$$

Scope

- The scope of a name begins from its definition and ends where the corresponding scope ends
- Scopes end with definitions of functions
- Scopes end with the keyword `end` in any `let ... in ...end` or `local ... in ...end`
- Scopes are delimited by brackets "$[\ldots]$" in (fully-bracketed) λ-abstractions.
- We simply use $\{\}$ to delimit scope

Scope Rules

- Scopes may be disjoint
- Scopes may be nested one completely within another
- A scope cannot span two disjoint scopes
- Two scopes cannot (partly) overlap

forward

Example 17.7 `local`. *Consider the following example ML program which uses local declarations in the development of the algorithm to determine whether a positive integer is perfect.*

```
local
  exception invalidArg;

  fun ifdivisor3 (n, k) =
    if n <= 0 orelse
       k <= 0 orelse
       n < k
    then raise invalidArg
    else if n mod k = 0
    then k
    else 0;
  fun sum_div2 (n, l, u) =
    if n <= 0 orelse
       l <= 0 orelse
```

```
      l > n orelse
      u <= 0 orelse
      u > n
    then raise invalidArg
    else if l > u
    then 0
    else ifdivisor3 (n, l)
      + sum_div2 (n, l+1, u)

in

  fun perfect n =
    if n <= 0
    then raise invalidArg
    else
      let
        val nby2 = n div 2
      in
```

```
        n = sum_div2 (n, 1, nby2)
        end
end
```

Scope & `local`

```
local
        fun fun1   y =
                        . . .

        fun fun2   z =  . . .
                            fun1

  in
        fun fun3   x  =   . . .
                         fun2 . . .
                           fun1 . . .
  end
```

Execution in the Modified Environment

Once a declaration has been processed a new scope γ' is created in which the new variables are available for use in addition to everything else that was previously present in the environment γ (unless it has been "hidden" by the use of the same name in the new scope). γ' is pushed onto γ to create a new environment $\gamma[\gamma']$. For any variable x,

$$\gamma[\gamma'](x) = \begin{cases} \gamma'(x) & \text{if } x \in Dom(\gamma') \\ \gamma(x) & \text{if } x \in Dom(\gamma) - Dom(\gamma') \\ \perp & otherwise \end{cases}$$

$$\boxed{\mathbf{D-Seq} \quad \frac{\gamma \vdash \langle \sigma, d_1 \rangle \longrightarrow_d \langle \gamma_1, \sigma_1 \rangle \qquad \gamma[\gamma_1] \vdash \langle \sigma_1, d_2 \rangle \longrightarrow_d \langle \gamma_2, \sigma_2 \rangle}{\gamma \vdash \langle \sigma, d_1; d_2 \rangle \longrightarrow_d \langle \gamma_1[\gamma_2], \sigma_2 \rangle}}$$

Semantics of Anonymous Blocks

$$\textbf{Block} \quad \frac{\gamma \vdash \langle \sigma, d \rangle \longrightarrow_d^* \langle \gamma', \sigma' \rangle \qquad \gamma[\gamma'] \vdash \langle \sigma', c \rangle \longrightarrow_c^* \sigma''}{\gamma \vdash \langle \sigma, \{d; c\} \rangle \longrightarrow_c \sigma'' \upharpoonright Dom(\sigma)}$$

Note.

- Note the use of the multi-step transitions on both declarations and commands
- We have given up on single-step movements, since taking these "big"-steps in the semantics is more convenient and less cumbersome
- Note that the "little" environment γ' which was produced by the declaration d is no longer present on exiting the block.
- On exiting the block the domain of the state returns to $Dom(\sigma)$, shedding the new locations that were created for the "little" environment.

Parameterless Subroutines: Named Blocks

The introduction of named blocks allows transfer of control from more control points than in the case of unnamed blocks.

$$
\begin{array}{rcl|l}
d_1, d_2, d & ::= & \cdots & \underline{\mathtt{sub}}\ P = c \\
c & ::= & \cdots & P
\end{array}
$$

- The scope rules remain the same. All names in c refer to the most recent definition in the innermost enclosing scope of the current scope.
- c may refer to variables that are visible in the static scope of P.

What does a procedure name represent?

- An anonymous block transforms a store σ to another store σ'.
- Each procedure name stands for a piece of code which effectively transforms the store.
- Unlike an anonymous block which has a fixed position in the code, a named procedure may be called from several points (representing many different states).
- Each procedure represents a "state transformer".
- However under static scope rules, the environment in which a procedure executes remains fixed though the store may vary.
- Our environment, in addition to having locations should also be able to associate names with state transformers.

$$
\begin{aligned}
Proc_0 &= Stores \rightarrow Stores \\
Env &= \{\gamma \mid \gamma : X \rightarrow (Loc + Proc_0)\}
\end{aligned}
$$

Each procedure declaration $\underline{\mathtt{sub}}\ P = c$ modifies the environment γ by associating the procedure name P with an entity called a procedure closure $proc0(c, \gamma)$, which represents the body of the procedure and the environment in which it is to be executed.

$$\mathbf{DSub0}\ \frac{}{\gamma \vdash \langle \sigma, \underline{\mathtt{sub}}\ P = c \rangle \longrightarrow_d \langle [P \mapsto proc0(c, \gamma)]\gamma, \sigma \rangle}$$

$$\mathbf{CSub0}\ \frac{\gamma_1 \vdash \langle \sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P \rangle \longrightarrow_c \langle [P \mapsto proc0(c, \gamma)]\gamma, \sigma' \rangle}\ (\gamma(P) = proc0(c, \gamma_1))$$

If P is recursive then we modify the last rule to

$$\mathbf{CrecSub0}\ \frac{\gamma_2 \vdash \langle \sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P \rangle \longrightarrow_c \langle [P \mapsto proc0(c, \gamma)]\gamma, \sigma' \rangle}\ (\gamma(P) = proc0(c, \gamma_1))$$

where $\gamma_2 = [P \mapsto \gamma(P)]\gamma_1$.

A generalization to mutual recursion with many such procedures is in principle

Subroutines with Value Parameters

We consider the case of only a single parameter for simplicity.

$$\begin{array}{rcl|l|l} d_1, d_2, d & ::= & \cdots & \underline{\texttt{sub}}\ P(\underline{\texttt{t}}\ x) = c & \underline{\texttt{sub}}\ P(\underline{\texttt{bool}}\ x) = c \\ c & ::= & \cdots & P(e) & \end{array}$$

$$\begin{array}{lcl} Proc_0 & = & Stores \to Stores \\ Proc_v & = & (Stores \times (\underline{\texttt{int}} \cup \underline{\texttt{bool}})) \to Stores \\ Proc & = & Proc_0 + Proc_v \\ Env & = & \{\gamma \mid \gamma : X \to (Loc + Proc)\} \end{array}$$

Semantics of Call-by-value

$$\textbf{DSubv}\quad \frac{}{\gamma \vdash \langle \sigma, \underline{\texttt{sub}}\ P(\underline{\texttt{t}}\ x) = c\rangle \longrightarrow_d \langle [P \mapsto proc_v(\underline{\texttt{t}}\ x, c, \gamma)]\gamma, \sigma\rangle}$$

where $\underline{\texttt{t}} \in \{\underline{\texttt{int}}, \underline{\texttt{bool}}\}$

$$\textbf{CrecSubv}\quad \frac{\gamma \vdash \langle \sigma, e\rangle \longrightarrow_e^* v \qquad \gamma_2 \vdash \langle [l \mapsto v]\sigma, c\rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P(e)\rangle \longrightarrow_c \sigma' \upharpoonright Dom(\sigma)} \quad (\gamma(P) = proc_v(\underline{\texttt{t}}\ x, c, \gamma_1))$$

where

- $\gamma_2 = [x \mapsto l][P \mapsto \gamma(P)]\gamma_1$,
- $l \notin Range(\gamma) \cup Dom(\sigma)$ and
- $\gamma(P) = proc_v(\underline{\texttt{t}}\ x, c, \gamma_1)$.

The Call-by-value parameter passing mechanism requires the evaluation of an expression for the value parameter to be passed to the procedure. It requires in addition the allocation of a location to store the value of the actual expression. This strategy while quite efficient for scalar variables is too expensive when the parameters are large structures such as arrays and records. In these case it is more usual to pass merely only a *reference* to the parameter and ensure that all modifications to any component of the formal parameter are instantaneously reflected also in the actual parameter.

We consider the case of a single reference parameter for simplicity. We consider the case of only a single parameter for simplicity.

$$\begin{array}{rcl|l|l} d_1, d_2, d & ::= & \cdots & \underline{\texttt{sub}}\ P(\underline{\texttt{ref t}}\ x) = c & \underline{\texttt{sub}}\ P(\underline{\texttt{ref bool}}\ x) = c \\ c & ::= & \cdots & P(x) & \end{array}$$

Notice that unlike the case of value parameters, the actual parameter in the calling code can only pass a variable that is already present in its environment.

We augment the definition of $Proc$ to include a new entity viz. $Proc_r$. We then have

$$
\begin{array}{lcl}
Proc_0 & = & Stores \rightarrow Stores \\
Proc_v & = & (Stores \times (\underline{\texttt{int}} \cup \underline{\texttt{bool}})) \rightarrow Stores \\
Proc_r & = & (Stores \times Loc) \rightarrow Stores \\
Proc & = & Proc_0 + Proc_v + Proc_r \\
Env & = & \{\gamma \mid \gamma : X \rightarrow (Loc + Proc)\}
\end{array}
$$

$$
\textbf{DSubr} \quad \frac{}{\gamma \vdash \langle \sigma, \underline{\texttt{sub}}\ P(\underline{\texttt{t}}\ x) = c \rangle \longrightarrow_d \langle [P \mapsto proc_r(\underline{\texttt{t}}\ x, c, \gamma)]\gamma, \sigma \rangle}
$$

where $\underline{\texttt{t}} \in \{\underline{\texttt{int}}, \underline{\texttt{bool}}\}$

$$
\textbf{CrecSubr} \quad \frac{\gamma_2 \vdash \langle [\sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P(y) \rangle \longrightarrow_c \sigma'} \quad (\gamma(P) = proc_r(\underline{\texttt{t}}\ x, c, \gamma_1))
$$

where

- $\gamma_2 = [x \mapsto \gamma(y)][P \mapsto \gamma(P)]\gamma_1$,

Logic Programming

> A program is a theory (in some logic) and computation is deduction from the theory.
>
> J. A. Robinson

FOL: Reversing the Arrow

Let

$$\phi \leftarrow \psi \stackrel{df}{=} \psi \rightarrow \phi$$

Consider any clause $C = \{\pi_1, \ldots, \pi_p\} \cup \{\neg\nu_1, \ldots, \neg\nu_n\}$ where π_i, $1 \leq i \leq p$ are *positive* literals and $\neg\nu_j$, $1 \leq j \leq n$ are the *negative* literals. Since a clause in FOL with free variables represents the universal closure of the disjunction of its literals, we have

FOL Arrow Reversal

$$\begin{aligned}
C &\Leftrightarrow \vec{\forall}[(\bigvee_{1\leq i\leq p} \pi_i) \vee (\bigvee_{1\leq j\leq n} \neg\nu_j)] \\
&\Leftrightarrow \vec{\forall}[(\bigvee_{1\leq i\leq p} \pi_i) \vee \neg(\bigwedge_{1\leq j\leq n} \nu_j)] \\
&\Leftrightarrow \vec{\forall}[(\bigwedge_{1\leq j\leq n} \nu_j) \rightarrow (\bigvee_{1\leq i\leq p} \pi_i)] \\
&\equiv \vec{\forall}[(\bigvee_{1\leq i\leq p} \pi_i) \leftarrow (\bigwedge_{1\leq j\leq n} \nu_j)] \\
&\stackrel{df}{=} \pi_1, \ldots, \pi_p \leftarrow \nu_1, \ldots, \nu_n
\end{aligned}$$

Definition 18.1: Horn clauses

Given a clause

$$C \stackrel{df}{=} \pi_1, \ldots, \pi_p \leftarrow \nu_1, \ldots, \nu_n$$

- Then C is a **Horn clause** if $0 \leq p \leq 1$.
- C is called a
  - **program clause** or **rule clause** if $p = 1$,
  - **fact** or **unit clause** if $p = 1$ and $n = 0$,
  - **goal clause** or **query** if $p = 0$,
- Each ν_j is called a **sub-goal** of the goal clause.

FOL: Program or Rule Clause

$$
\begin{aligned}
P &\stackrel{df}{=} \pi \leftarrow \nu_1, \ldots, \nu_n \\
&\equiv \vec{\forall}[\pi \vee (\bigvee_{1 \leq j \leq n} \neg \nu_j)] \\
&\equiv \vec{\forall}[\pi \vee \neg(\bigwedge_{1 \leq j \leq n} \nu_j)]
\end{aligned}
$$

and is read as "π if ν_1 and ν_2 and ... and ν_n".

FOL Facts: Unit Clauses

$$
\begin{aligned}
F &\stackrel{df}{=} \pi \\
&\equiv \vec{\forall}[\pi]
\end{aligned}
$$

FOL: Goal clauses

Given a goal clause

$$G \stackrel{df}{=} \leftarrow \nu_1, \ldots, \nu_n$$
$$\Leftrightarrow \vec{\forall}[\neg\nu_1 \vee \ldots \vee \neg\nu_n]$$
$$\Leftrightarrow \neg\vec{\exists}[\nu_1 \wedge \ldots \wedge \nu_n]$$

If $\vec{y} = FV(\nu_1 \wedge \ldots \wedge \nu_n)$ then the goal is to prove that there exists an assignment to $\vec{y}$ which makes $\nu_1 \wedge \ldots \wedge \nu_n$ true.

Definition 18.2: First-order Logic programs

A **First-order logic program** is a finite set of Horn clauses, i.e. it is a set of **rules** $P = \{h^1, \ldots, h^k\}$, $k \geq 0$ with $h^l \equiv \pi^l \leftarrow \nu_1^l, \ldots, \nu_{n_l}^l$, for $0 \leq l \leq k$. π^l is called the **head** of the rule and $\nu_1^l, \ldots, \nu_{n_l}^l$ is the **body** of the rule.

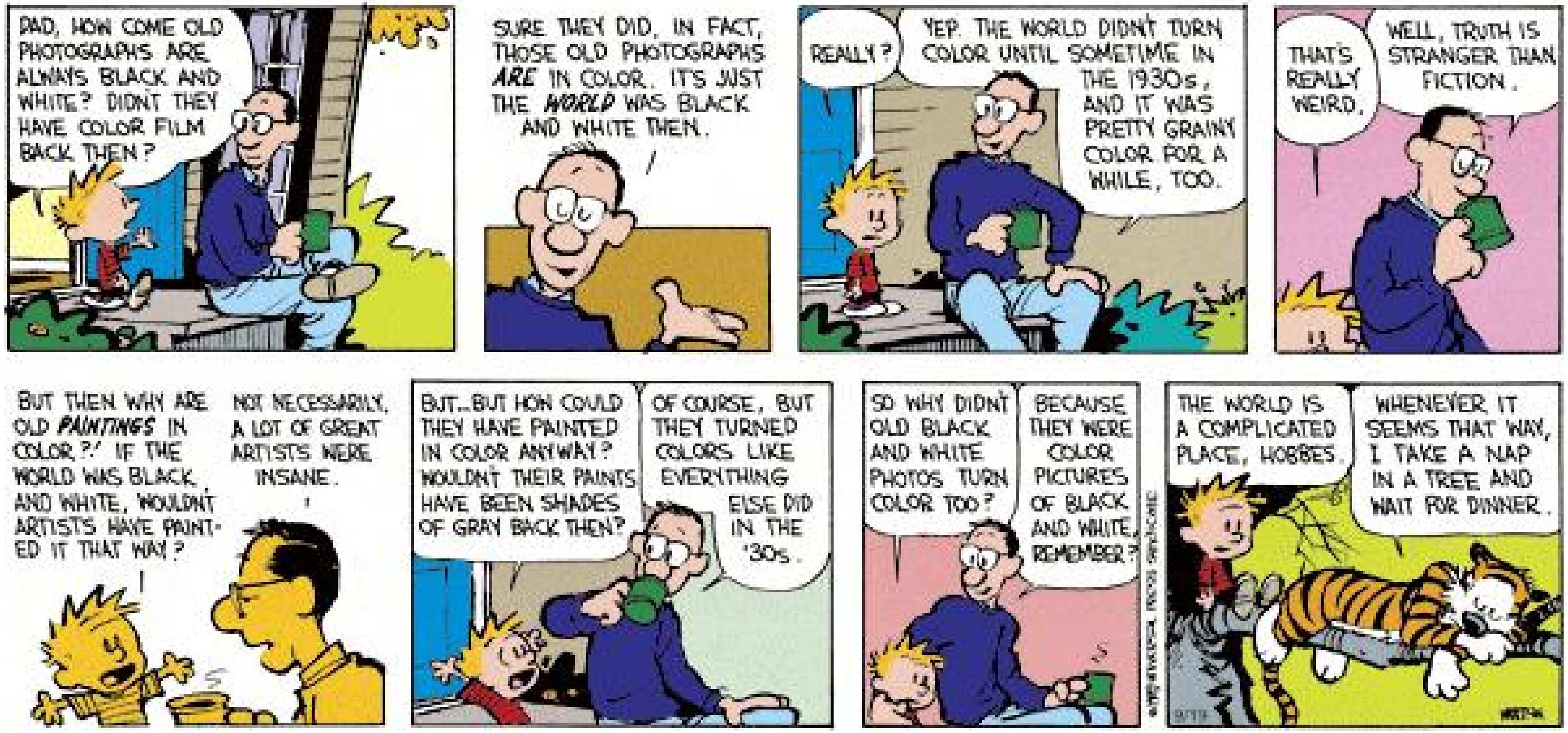
Given a logic program P and a goal clause $G = \{\nu_1, \ldots, \nu_n\}$ the basic idea is to show that

$$\begin{aligned} & P \cup \{G\} \text{ is unsatisfiable} \\ \Leftrightarrow\ & \vec{\exists}[\nu_1 \wedge \cdots \wedge \nu_n] \text{ is a logical consequence of } P \end{aligned}$$

Effectively showing $P \models \vec{\exists}[\nu_1 \wedge \cdots \wedge \nu_n]$ implies that we need to find values for the variables $X = \bigcup_{j=1}^{n} FV(\nu_j)$ which ensure that $P \cup \{G\}$ is unsatisfiable. By Herbrand's theorem this reduces to the problem of finding substitutions of ground terms in the Herbrand base for variables in such a manner as to ensure unsatisfiability of $P \cup \{G\}$. This substitution is also called a *correct answer substitution*.

We may regard a logic program therefore as a set of postulates of a family of models (represented by a Herbrand model) and any correct answer substitution that may be derived (through resolution refutation) as a proof of the Goal as a logical consequence of the postulates. Since resolution refutation is sound and complete we effectively show $P \vdash_{\mathscr{R}} \vec{\exists}[\nu_1 \wedge \cdots \wedge \nu_n]$ where $\vdash_{\mathscr{R}}$ denotes a proof by resolution refutation.

A propositional logic program is one in which there are no variables either in P or in the goal clause G and the execution is a pure application of the rule Res0 to obtain a contradiction.


DAD, HOW COME OLD PHOTOGRAPHS ARE ALWAYS BLACK AND WHITE? DIDN'T THEY HAVE COLOR FILM BACK THEN?
SURE THEY DID. IN FACT, THOSE OLD PHOTOGRAPHS *ARE* IN COLOR. IT'S JUST THE *WORLD* WAS BLACK AND WHITE THEN.
REALLY?
YEP. THE WORLD DIDN'T TURN COLOR UNTIL SOMETIME IN THE 1930s, AND IT WAS PRETTY GRAINY COLOR FOR A WHILE, TOO.
THAT'S REALLY WEIRD.
WELL, TRUTH IS STRANGER THAN FICTION.
BUT THEN WHY ARE OLD *PAINTINGS* IN COLOR?! IF THE WORLD WAS BLACK AND WHITE, WOULDN'T ARTISTS HAVE PAINTED IT THAT WAY?
NOT NECESSARILY. A LOT OF GREAT ARTISTS WERE INSANE.
BUT... BUT HOW COULD THEY HAVE PAINTED IN COLOR ANYWAY? WOULDN'T THEIR PAINTS HAVE BEEN SHADES OF GRAY BACK THEN?
OF COURSE, BUT THEY TURNED COLORS LIKE EVERYTHING ELSE DID IN THE '30s.
SO WHY DIDN'T OLD BLACK AND WHITE PHOTOS TURN COLOR TOO?
BECAUSE THEY WERE COLOR PICTURES OF BLACK AND WHITE, REMEMBER?
THE WORLD IS A COMPLICATED PLACE, HOBBES.
WHENEVER IT SEEMS THAT WAY, I TAKE A NAP IN A TREE AND WAIT FOR DINNER.

Prolog: EBNF1

```
<program> ::= <clause list> <query> | <query>
<clause list> ::= <clause> | <clause list> <clause>
<clause> ::= <predicate> . | <predicate> :- <predicate list>.
<predicate list> ::= <predicate> |
                     <predicate list> , <predicate>
<predicate> ::= <atom> | <atom> ( <term list> )
<term list> ::= <term> | <term list> , <term>
<term> ::= <numeral> | <atom> | <variable> | <structure>
<structure> ::= <atom> ( <term list> )
<query> ::= ?- <predicate list>.
```

Prolog: EBNF2

```
<atom> ::= <small atom> | ' <string> '
<small atom> ::= <lowercase letter> |
                 <small atom> <character>
<variable> ::= <uppercase letter> | <variable> <character>
<lowercase letter> ::= a | b | c | ... | x | y | z
<uppercase letter> ::= A | B | C | ... | X | Y | Z | _
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<character> ::= <lowercase letter> | <uppercase letter> |
                <digit> | <special>
<special> ::= + | - | * | / | \ | ^ | ~ | : | . | ? |   |
              # | $ | &
<string> ::= <character> | <string> <character>
```

Algorithm

Algorithm 18.1

$\text{INTERPRET0}\ (P, G) \stackrel{df}{=}$

$$\left\{\begin{array}{l}
\quad \textbf{requires: } \text{A propositional logic program } P \text{ and propositional goal } G \\
goals := \{G\} \\
\textbf{while } goals \neq \emptyset \\
\quad \textbf{do } \left\{\begin{array}{l}
\text{Choose some goal } A \in goals \\
\text{Choose a clause } A' \leftarrow B_1, \ldots, B_k \in P : A \equiv A' \\
\textbf{if } A' \text{ does not exist} \\
\quad \textbf{then exit} \\
\\
\quad \textbf{else } goals := (goals - \{A\}) \cup \{B_1, \ldots, B_k\}
\end{array}\right. \\
\textbf{if } goals = \emptyset \\
\quad \textbf{then return } (yes) \\
\quad \textbf{else return } (no) \\
\quad \textbf{ensures: } yes \text{ if } P \vdash G \text{ else } no
\end{array}\right.$$

Algorithm

Algorithm 18.2

$\text{INTERPRET1}\ (P, G) \stackrel{df}{=}$

$$
\left\{
\begin{array}{l}
\quad \textbf{requires: } \text{A logic program } P \text{ and goal } G \\
\text{Standardize variables apart in } \mathbf{P} \cup \{G\} \\
goalStack := emptyStack \\
\theta := \mathbf{1} \\
push(goalStack, \theta G) \\
\textbf{while } \neg empty(goalStack) \\
\quad \textbf{do } \left\{
\begin{array}{l}
A := pop(goalStack) \\
\textbf{if } \exists A' \leftarrow B_1, \ldots, B_k \in P : unifiable(A, A') \\
\quad \textbf{then } \left\{
\begin{array}{l}
\tau := \text{UNIFY } (A, A') \; //algorithm\ \textbf{??} \\
\theta := \tau \circ \theta
\end{array}
\right. \\
\quad \textbf{else exit} \\
\textbf{if } k > 0 \\
\quad \textbf{then } push(goalStack, \theta B_k, \ldots, \theta B_1)
\end{array}
\right. \\
\textbf{if } empty(goalStack) \\
\quad \textbf{then return } (\theta) \\
\quad \textbf{else return } (no) \\
\quad \textbf{ensures: } \text{if } P \vdash G \text{ then } \theta \text{ else } no
\end{array}
\right.
$$

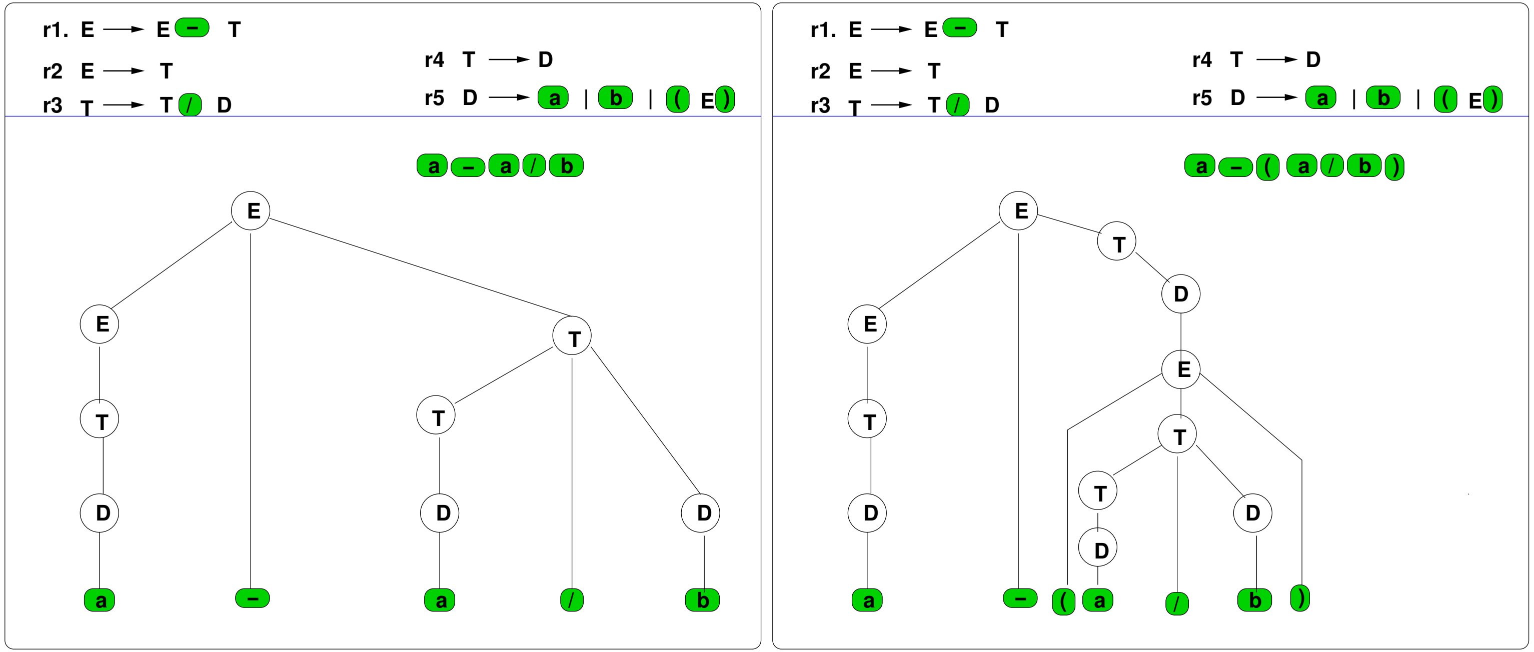


Figure 5: Derivation trees or Concrete parse trees example 5.3

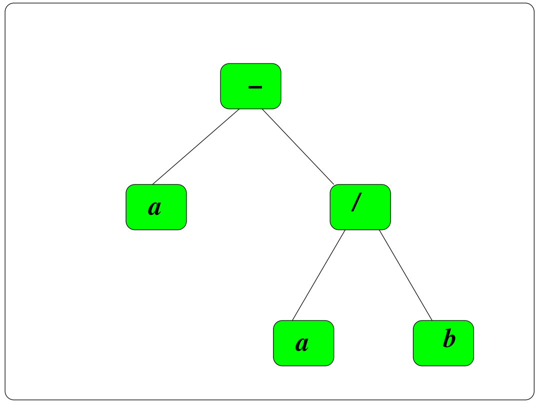


Figure 6: Abstract syntax tree (AST) for the sentences in fig. 5

Figure 7: Case $m > 0$ and $n > 0$

$$
\begin{tikzcd}
 & & M \arrow[dr, "*"] & & \\
 & M_1 \arrow[ur, "*"] \arrow[dr, "*"] & & Q \arrow[dr, "*"] & \\
L \arrow[ur, "1"] \arrow[dr, "1"'] & & P \arrow[ur, "*"] \arrow[dr, "*"] & & S \\
 & N_1 \arrow[ur, "*"] \arrow[dr] & & R \arrow[ur, "*"] & \\
 & & N \arrow[ur, "*"] & &
\end{tikzcd}
$$