Programming Languages
http://www.cse.iitd.ac.in/~sak/courses/pl/2017-18.index.html

S. Arun-Kumar
Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

March 17, 2018
Contents

1 The Programming Languages Geneology 3

2 Introduction 13

3 Scanning or Lexical Analysis 31
   3.1 Regular Expressions .................................................. 41
   3.2 Nondeterministic Finite Automata (NFA) .......................... 59
   3.3 Deterministic Finite Automata (DFA) ............................. 92

4 Parsing or Syntax Analysis 105
   4.1 Context-Free Grammars ................................................ 108
   4.2 Ambiguity ................................................................. 120
   4.3 Shift-Reduce Parsing .................................................... 139
   4.4 Bottom-Up Parsing ....................................................... 175
   4.5 Recursive Descent Parsing ............................................ 193
   4.6 Specification of Syntax: Extended Backus-Naur Form .......... 197
      4.6.1 The Extended Backus-Naur Form (EBNF) .................... 197

5 Attributes & Semantic Analysis 204
6 Abstract Syntax 249
7 The Pure Untyped Lambda Calculus: Basics 263
8 Notions of Reduction 275
9 Confluence Definitions 284
10 The Church-Rosser Property 295
11 Confluence Characterizations 301
12 An Applied Lambda-Calculus 309
  12.1 FL with recursion 311
  12.2 FL with type rules 327
13 An Imperative Language 340
  13.1 The Operational Semantics of Expressions 342
    13.1.1 l-values, r-values, aliasing and indirect addressing 346
  13.2 The Operational Semantics of Commands 352
  13.3 The Semantics of Expressions in FL 362
  13.4 The Operational Semantics of Declarations 369
13.5 The Operational Semantics of Subroutines .......................................................... 398
13.6 Runtime Structure ................................................................................................ 407
1. The Programming Languages Geneology
The Landscape of General PLs
The Usage of General PLs
The Major Features of General PLs

- **Static Memory alloc.**
- **Static scope**
- **Static types**
- **Runtime stack**
- **Heap alloc.**
- **Dynamic scope**
- **Dynamic Memory alloc.**
- **Dynamic memory alloc.**
- **Statically scoped**
- **Untyped**
- **Typed**
- **Imperative**
- **Object-oriented**
- **Functional**
- **Declarative**
FORTRAN

• The very first high-level programming language
• Still used in scientific computation
• Static memory allocation
• Very highly compute oriented
• Runs very fast because of static memory allocation
• Parameter passing by reference
COBOL

- A business oriented language
- 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
- 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 syntax of a programming language
- Formal syntax defined by BNF (an extension of context-free grammars)
- First language to implement recursion
- Introduction of block-structure and nested scoping
- Dynamic memory allocation
- Introduced the call-by-name parameter mechanism
Pascal

- ALGOL-like language meant for teaching structured programming
- Introduction of new data structures – records, enumerated types, subrange 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
- 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
2. Introduction
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 theory, data structures and algorithms.
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. \text{\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


Source and Target

In general a **compiler** for a **source** language $S$ written in some language $C$ translates code to a **target** language $T$.

**Source** $S$ could be

- a programming language, or
- a description language (e.g. Verilog, VHDL), or
- a markup language (e.g. XML, HTML, SGML, \LaTeX)

**Target** $T$ could be

- 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.

We will be primarily concerned with compiling from a **high-level** programming language (**source**) to **low-level** code.
The Compiling Process

In general the process of compiling involves at least three languages

1. The language $S$ of source programs in which the users of the compiler write code.

2. The language $C$ in which the compiler itself is written. The assumption is that unless the compiler itself is written in machine language there is already a compiler or an interpreter for $C$.

3. The language $T$ into which the compiler translates the user programs.

Besides these three languages there could be several other intermediate languages $I_1, I_2, \ldots$ (also called intermediate representations) into which the source program could be translated in the process of compiling or interpreting the source programs written in $S$. In modern compilers, for portability, modularity and reasons of code improvement, there is usually at least one intermediate representation.
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 $S$ 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.
The Big Picture: 1

stream of characters

stream of tokens

SCANNER
The Big Picture: 2

```
stream of characters

SCANNER

stream of tokens

PARSER

parse tree
```
The Big Picture: 3

- **Scanner**: stream of characters → stream of tokens
- **Parser**: stream of tokens → parse tree
- **Semantic Analyzer**: parse tree → abstract syntax tree
The Big Picture: 4

SCANNER

stream of characters

stream of tokens

PARSER

parse tree

SEMANTIC ANALYZER

abstract syntax tree

I.R. CODE GENERATOR

intermediate representation
The Big Picture: 5

```
stream of characters

SCANNER

stream of tokens

PARSER

parse tree

SEMANTIC ANALYZER

abstract syntax tree

I.R. CODE GENERATOR

intermediate representation

OPTIMIZER

optimized intermediate representation
```
The Big Picture: 6

The diagram illustrates the process of converting a stream of characters into target code. It consists of the following steps:

1. **Scanner**: Converts a stream of characters into a stream of tokens.
2. **Parser**: Generates a parse tree from the stream of tokens.
3. **Semantic Analyzer**: Produces an abstract syntax tree from the parse tree.
4. **Intermediate Representation Code Generator**: Creates an intermediate representation from the abstract syntax tree.
5. **Optimizer**: Generates an optimized intermediate representation from the intermediate representation.
6. **Code Generator**: Converts the optimized intermediate representation into target code.

The diagram shows the flow of information from one step to the next, illustrating how the various components work together to produce the final target code.
The Big Picture: 7

The diagram illustrates the components of a compiler, including:

- **Scanner**: Converts a stream of characters into a stream of tokens.
- **Parser**: Converts the stream of tokens into a parse tree.
- **Semantic Analyzer**: Determines the meaning of the parse tree.
- **Intermediate Representation (I.R.) Code Generator**: Converts the abstract syntax tree into an intermediate representation.
- **Optimizer**: Optimizes the intermediate representation.
- **Code Generator**: Converts the optimized intermediate representation into target code.
- **Symbol Table Manager**: Manages symbol tables.
- **Error Handler**: Handles syntax errors and other errors.

The process starts with a stream of characters, which is scanned into tokens. These tokens are then parsed into a parse tree, which is analyzed semantically. The output of the semantic analysis is an abstract syntax tree, which is converted into an intermediate representation. The intermediate representation is then optimized, and finally, the target code is generated.
The Big Picture: 8

Scanner → Parser → Semantic Analysis → Symbol Table Manager

Parser → Semantic Analyzer

Semantic Analyzer → Intermediate Code Generator

Intermediate Code Generator → Optimizer

Optimizer → Code Generator

Code Generator → Symbol Table Manager

Symbol Table Manager → Error Handler

Error Handler → Scanner

Scanner → Stream of Characters

Parser → Stream of Tokens

Semantic Analyzer → Parse Tree

Intermediate Code Generator → Abstract Syntax Tree

Optimzier → Optimized Intermediate Representation

Code Generator → Target Code
3. Scanning or Lexical Analysis

Lexical Analysis
Language

• Every language is built from a finite alphabet of symbols. The alphabet of a programming language consists of the symbols of the ASCII set.

• Each language has a vocabulary consisting of words. Each word is a string of symbols drawn from the alphabet.

• Each language has a finite set of punctuation symbols, which separate phrases, clauses and sentences.

• The phrases, clauses and sentences of a programming language are expressions, commands, functions, procedures and programs.
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 groups of meaningful words, and other symbols.
- Each such group of characters is then classified as belonging to a certain token type.
- Certain sequences of characters are not tokens and are completely ignored (or skipped) by the compiler.
Tokens and Non-tokens

Tokens Typical tokens are

- **Identifiers**: Names of variables, constants, procedures, functions etc.
- **Keywords/Reserved words**: void, public, main
- **Operators**: +, *, /
- **Punctuation**: ,, :, .
- **Brackets**: (, ), [, ], begin, end, case, esac

Non-tokens Typical non-tokens

- **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

- Takes a stream of characters and identifies tokens from the lexemes.
- Eliminates comments and redundant whitespace.
- Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting to the user.
Scanning: 2

- **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 like +, -, *).
- **Examples of lexemes.**
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives
Scanning: 3

• 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 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.
Lecture 03
Regular Expressions
Specifying Lexical Rules

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

- there are an infinite number of legally correct programs in any programming language and
- we require finite descriptions/specifications of the lexical rules so that they can cover the infinite number of legal programs.

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

- Each regular expression is a *finite* sequence of symbols.
- A regular expression may be used to describe an *infinite* collection of strings.

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

\[ [A-Za-z][A-Za-z0-9_]^* \]
Concatenations

Consider a (finite) alphabet (of symbols) $A$.

- Any set of strings built up from the symbols of $A$ is called a language.
- Given any two strings $x$ and $y$ in a language, $x.y$ or simply $xy$ is the concatenation of the two strings.

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

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

**Example** Let $X = \{\text{Mengesha, Gemechis}\}$ and $Y = \{\text{Mamo, Bekele, Selassie}\}$

$XY = \{\text{MengeshaMamo, MengeshaBekele, MengeshaSelassie, GemechisMamo, GemechisBekele, GemechisSelassie}\}$
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 $A$ of symbols. Each regular expression $r$ denotes a set of strings $L(r)$. $L(r)$ 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 $\{a\}$.

(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,

$$L(rs) = L(r)L(s)$$
Epsilon and Alternation

Epsilon $\epsilon$ denotes the language with a single element the empty string (""") i.e.

$$L(\epsilon) = \{"\"\}$$

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.

$$L(r | s) = L(r) \cup L(s)$$

Example $L(\text{Menelik}|\text{Selassie}|\epsilon) = \{\text{Menelik, Selassie, """"}\}$. 
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.

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

Then

$$x^* = \{x^n \mid n \geq 0\}$$
String Repetitions Example

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

$$x^0 = ""$$
$$x^1 = \text{Selassie}$$
$$x^2 = \text{SelassieSelassie}$$

\vdots

$$x^5 = \text{SelassieSelassieSelassieSelassieSelassieSelassie}$$

\vdots

Then $x^*$ is the language consisting of all strings that are finite repetitions of the string 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.

\[
\begin{align*}
X^0 &= "" \\
X^1 &= X \\
X^2 &= X.X \\
&\vdots \\
X^{n+1} &= X.X^n = X^n.X \\
&\vdots \\
\end{align*}
\]

Then

\[
X^* = \bigcup_{n\geq 0} X^n
\]
Language Iteration Example

Example Let $X = \{\text{Mengesha, Gemechis}\}$. Then

$$X^0 = \{\"\"\}\n$$

$$X^1 = \{\text{Mengesha, Gemechis}\}\n$$

$$X^2 = \{\text{MengeshaMengesha, GemechisMengesha, GemechisGemechis, GemechisGemechisGemechis}\}\n$$

$$X^3 = \{\text{MengeshaMengeshaMengesha, GemechisMengeshaMengesha, GemechisGemechisMengesha, GemechisGemechisGemechis, GemechisGemechisGemechisGemechis}\}\n$$

$$\vdots\n$$

$$X^{n+1} = X.X^n\n$$

$$\vdots$$
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 \textbf{Kleene closure} of \( r \), denoted \( r^* \) specifies the language \( (\mathcal{L}(r))^* \).
In general
\[
r^* = r^0 \mid r^1 \mid \ldots \mid r^{n+1} \mid \ldots
\]
denotes an \textbf{infinite union} of languages.

\textbf{Question 1.} For what regular expression \( r \) will \( r^* \) specify a \textit{finite} set?

\textbf{Question 2.} How many strings will be in the language specified by \((a \mid b \mid c)^n\)?

\textbf{Question 3.} Give an informal description of the language specified by \((a \mid b \mid c)^*\)?

\textbf{Question 4.} Give a regular expression which specifies the language \( \{a^k \mid k > 100\} \).
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

$$r^+ = r.r^*$$

We also have the identity

$$r^* = \epsilon \mid r^+$$

**Question 5.** Simplify the expression $r^*.r^*$, i.e. give a simpler regular expression which specifies the same language.

**Question 6.** Simplify the expression $r^+.r^+$. 
Range Specifications

We may specify ranges of various kinds as follows.

• \([a–c] = a \mid b \mid c\). Hence the expression of Question 3 may be specified as \([a–c]^*\).

• Multiple ranges: \([a–c0–3] = [a–c] \mid [0–3]\)

Question 6. Try to understand what the regular expression for identifiers really specifies.

Question 7. Modify the regular expression so that all identifiers start only with upper-case letters.

Question 8. 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.
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 3.1 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 3.2 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 3.3 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 3.4** 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 3.5** 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 $(a^b)^c$?
3.2. Nondeterministic Finite Automata (NFA)
Nondeterministic Finite Automata

A regular expression is useful in defining a finite state automaton. An automaton is a machine (simple program) which can be used to recognize all valid lexical tokens of a language. A nondeterministic finite automaton (NFA) $N$ over a finite alphabet $\mathcal{A}$ 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 $\rightarrow \subseteq Q \times (\mathcal{A} \cup \{\varepsilon\}) \times Q$. Equivalently

$$\rightarrow: Q \times (\mathcal{A} \cup \{\varepsilon\}) \rightarrow 2^Q$$

is a function that for each source state $q \in Q$ and symbol $a \in \mathcal{A}$ associates a set of target states.
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 $\varepsilon$ transitions represent going to a new target state *without* reading any input symbol.
- The NFA may be nondeterministic because of
  - one or more $\varepsilon$ 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 $\varepsilon$ (and going to different states).
Acceptance of NFA

• For any alphabet $A$, $A^*$ denotes the set of all (finite-length) strings of symbols from $A$.

• Given a string $x = a_1a_2\ldots a_n \in A^*$, an accepting sequence is a sequence of transitions

\[
q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_1} \xrightarrow{\varepsilon} \cdots q_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_2} \cdots \xrightarrow{\varepsilon} a_n \xrightarrow{\varepsilon} \cdots q_n
\]

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

\[
q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_1} \xrightarrow{\varepsilon} \cdots q'_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_2} \cdots \xrightarrow{\varepsilon} a_n \xrightarrow{\varepsilon} \cdots q'_n
\]

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.

- $L(N)$ 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:

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
<th>( N_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 )</td>
<td>( { a_f } )</td>
<td>( \emptyset \cdots \emptyset \emptyset )</td>
</tr>
<tr>
<td>( a_f )</td>
<td>( \emptyset )</td>
<td>( \emptyset \cdots \emptyset \emptyset )</td>
</tr>
</tbody>
</table>

Notice that all the cells except one have empty targets.
Regular Expressions to NFAs: 2

\[ N_\varepsilon \]  |  Input Symbol
---|---
State  |  a  |  \( \varepsilon \)  \\
\( \varepsilon_0 \)  |  \( \emptyset \)  |  \( \emptyset \ldots \emptyset \)  |  \{ \( \varepsilon_f \) \}  \\
\( \varepsilon_f \)  |  \( \emptyset \)  |  \( \emptyset \ldots \emptyset \)  |  \emptyset  \\

\[ \varepsilon \]
Regular Expressions to NFAs: 3

\[
\begin{array}{c|ccc}
N_r|s & \text{Input Symbol} \\
\text{State} & a & \cdots & \epsilon \\
\hline
r|s_0 & \emptyset & \cdots & \{r_0, s_0\} \\
r_0 & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots \\
r_f & \cdots & \cdots & \{r|s_f\} \\
s_0 & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots \\
s_f & \cdots & \cdots & \{r|s_f\} \\
r|s_f & \emptyset & \cdots & \emptyset \\
\end{array}
\]
Regular Expressions to NFAs:4

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

![Diagram of NFA for regular expression]

<table>
<thead>
<tr>
<th>$N_{r^*}$</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>a</td>
</tr>
<tr>
<td>$r^*_0$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$r_0$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$r_f$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$r^*_f$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Regular expressions vs. NFAs

• It is obvious that for each regular expression $r$, the corresponding NFA $N_r$ is correct by construction i.e.
  \[ L(N_r) = L(r) \]

• 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 $\Sigma$
  – or at most 2 outgoing transitions on $\varepsilon$

• Hence $N_r$ has at most $2|r|$ states and $4|r|$ transitions.
Example

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

- Assume the alphabet \(A = \{a, b\}\).
- We follow the steps of the construction as given in Constructing NFA to Regular Expressions to NFAs:
- 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 $(a|b)^*abb$
Example:-5

Steps in NFA for \((a|b)^*abb\)
Example:-4

Steps in NFA for \((a|b)^*abb\)
Example:-3

Steps in NFA for \((a|b)^*abb\)
Example:-2

Steps in NFA for $(a|b)^*abb$
Example:- 1

Steps in NFA for \((a|b)^*abb\)
Example-final

Steps in NFA for $(a|b)^*abb$
Extensions

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 to construct \( N_{r\{k, n\}} \) given \( N_r \).

4. Consider a new regular expression operator \( ^\wedge \) defined by

\[
\mathcal{L}(^\wedge r) = A^* - \mathcal{L}(r)
\]

What is the automaton \( N_{^\wedge r} \) given \( N_r \)?
Lecture 05
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 $\varepsilon$ 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 a $\varepsilon$ 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 $\varepsilon$ transitions.
- The $\varepsilon$ 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 $\varepsilon$, we need to eliminate them without changing the language recognized by the automaton.

• Two consecutive $\varepsilon$ transitions are the same as one. In fact any number of $\varepsilon$ transitions are the same as one. So as a first step we compute all finite sequences of $\varepsilon$ transitions and collapse them into a single $\varepsilon$ transition.

• Two states $q, q'$ are equivalent if there are only $\varepsilon$ transitions between them. This is called the $\varepsilon$-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 $\varepsilon$ transitions.

---

**Algorithm 1 ε–Closure**

**Require:** $T$ a set of states of the NFA

**Ensure:** $T_\varepsilon = \varepsilon$–Closure($T$).

1. $U := T$
2. repeat
3. $U_{old} := U$
4. $U := U_{old} \cup \{q' \mid q' \notin U, \exists q \in U_{old} : q \xrightarrow{\varepsilon} q'\}$
5. until $U = U_{old}$
6. $T_\varepsilon = U$
7. return $T_\varepsilon$
Analysis of $\varepsilon$-Closure

- $U$ can only grow in size through each iteration.
- The set $U$ 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.

**Algorithm 2 Recognition using NFA**

**Require:** A string $x \in A^*$.

**Ensure:** Boolean

$$S := \varepsilon\text{-}\text{Closure}([q_0]).$$

$$a := \text{nextchar}(x)$$

**while** $a \neq \text{end}\_of\_string$ **do**

$$S := \varepsilon\text{-}\text{Closure}(S \xrightarrow{a})$$

$$a := \text{nextchar}(x)$$

**end while**

**return** $S \cap F \neq \emptyset$

In the above algorithm we extend our notation for targets of transitions to include sets of sources. Thus

$$S \xrightarrow{a} = \{q' \mid \exists q \in S : q \xrightarrow{a} q'\}$$
Analysis of Recognition using NFA

- Even if $\varepsilon$-closure is computed as a call from within the algorithm, the time taken to recognize a string is bounded by $O(|x|.|Q_{Nr}|)$ where $|Q_{Nr}|$ is the number of states in $N_r$.

- The space required for the automaton is at most $O(|r|)$.

- Given that $\varepsilon$-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 $\varepsilon$-closures are pre-computed one may then work towards a Deterministic automaton to reduce the space required.
Lecture 06
Conversion of NFAs to DFAs
Deterministic Finite Automata

• A deterministic finite automaton (DFA) is a NFA in which
  1. there are no transitions on $\varepsilon$ and
  2. $\rightarrow$ yields a *at most one* target state for each source state and symbol from $A$ i.e. $\rightarrow$ is no longer a relation but a *function* of the form

\[ \rightarrow: Q \times A \rightarrow Q \]

• Clearly if every regular expression had a DFA which accepts the same language, all backtracking could be avoided.
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 \(\varepsilon\) 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 states or
   (b) a set consisting of a single state or
   (c) a set consisting of more than one state or
Transition Tables of DFAs

In the case of a deterministic automaton
1. there are no $\varepsilon$ transitions, and
2. for each state $q$ and letter $a$
   
   (a) either there is no transition
   
   (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 $N = \langle Q_N, A \cup \{\varepsilon\}, s_N, F_N, \rightarrow_N \rangle$ be a NFA with

- $Q_N$ the set of states of the NFA
- $A$ the alphabet
- $s_N \in Q_N$ the start state of the NFA
- $F_N$ the final or accepting states of the NFA and
- $\rightarrow_N \subseteq Q_N \times A \times Q_N$ the transition relation.

We would like to construct a DFA $D = \langle Q_D, A, s_D, F_D, \rightarrow_D \rangle$ where

- $Q_D$ the set of states of the DFA
- $A$ the alphabet
- $s_D \in Q_D$ the start state of the DFA
- $F_D$ the final or accepting states of the DFA and
- $\rightarrow_D \subseteq Q_D \times A \times Q_D$ the transition function of the DFA.

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

• The $\varepsilon$-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 $\varepsilon$s.

• 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 $\varepsilon$ transitions for the same input symbol.

• A major source of non-determinism is the presence of $\varepsilon$ transitions. The use of $\varepsilon$–Closure creates a cluster of similar states.

• 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 though to be similar.
NFA to DFA construction

Algorithm 3 Construction of DFA from NFA

Require: NFA $N = \langle Q_N, A \cup \{\varepsilon\}, s_N, F_N, \xrightarrow{N} \rangle$

Ensure: DFA $D = \langle Q_D, A, s_D, F_D, \xrightarrow{D} \rangle$ with $L(N) = L(D)$

1. $s_D := \varepsilon$-Closure($\{s_N\}$);
2. $Q_D := \{s_D\}; F_D := \emptyset; \xrightarrow{D} := \emptyset$
3. $U := \{s_D\}$ \{U is the set of unvisited states of the DFA\}
4. while $U \neq \emptyset$ do
5.  Choose any $q_D \in U; U := U - \{q_D\}$
6.  for all $a \in A$ do
7.    $q_D' := \varepsilon$-Closure($q_D \xrightarrow{a} N$) \{Note: $q_D \subseteq Q_N$\}
8.    $\xrightarrow{D} := \xrightarrow{D} \cup q_D \xrightarrow{a} q_D'$
9.    if $q_D' \cap F_N \neq \emptyset$ then
10.       $F_D := F_D \cup \{q_D'\}$
11.  end if
12.  if $q_D' \notin Q_D$ then
13.     $Q_D := Q_D \cup \{q_D'\}$
14.     $U := U \cup \{q_D'\}$
15.  end if
16. end for
17. end while
Example-NFA

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

and apply the NFA to DFA construction algorithm
Determinising

\( N(a|b)^\ast abb \)

\[ EC_0 = \varepsilon \rightharpoonup \text{Closure}(0) = \{0, 1, 2, 3, 7\} \]

2 \( \xrightarrow{a} \) \( N \) 4 and 7 \( \xrightarrow{a} \) \( N \) 8. So \( EC_0 \xrightarrow{a} D \varepsilon \rightharpoonup \text{Closure}(4, 8) = EC_{4,8} \). Similarly

\[ EC_0 \xrightarrow{b} D \varepsilon \rightharpoonup \text{Closure}(5) = EC_5 \]

\[ EC_{4,8} = \varepsilon \rightharpoonup \text{Closure}(4, 8) = \{4, 6, 7, 1, 2, 3, 8\} \]

\[ EC_5 = \varepsilon \rightharpoonup \text{Closure}(5) = \{5, 6, 7, 1, 2, 3\} \]

\[ EC_5 \xrightarrow{a} D \varepsilon \rightharpoonup \text{Closure}(4, 8) = EC_{4,8} \] and \( EC_5 \xrightarrow{b} D \varepsilon \rightharpoonup \text{Closure}(5) \)

\[ EC_{4,8} \xrightarrow{a} D \varepsilon \rightharpoonup \text{Closure}(4, 8) = EC_{4,8} \] and \( EC_{4,8} \xrightarrow{b} D \varepsilon \rightharpoonup \text{Closure}(5, 9) = EC_{5,9} \)

\[ EC_{5,9} = \varepsilon \rightharpoonup \text{Closure}(5, 9) = \{5, 6, 7, 1, 2, 3, 9\} \]

\[ EC_{5,9} \xrightarrow{a} D \varepsilon \rightharpoonup \text{Closure}(4, 8) = EC_{4,8} \] and \( EC_{5,9} \xrightarrow{b} D \varepsilon \rightharpoonup \text{Closure}(5, 10) = EC_{5,10} \)

\[ EC_{5,10} = \varepsilon \rightharpoonup \text{Closure}(5, 10) = \{5, 6, 7, 1, 2, 3, 10\} \]

\[ EC_{5,10} \xrightarrow{a} D \varepsilon \rightharpoonup \text{Closure}(4, 8) \] and \( EC_{5,10} \xrightarrow{b} \varepsilon \rightharpoonup \text{Closure}(5) \)
Final DFA

\[ D_{(a|b)^*abb} \]

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( EC_0 )</td>
<td></td>
</tr>
<tr>
<td>( EC_{4,8} )</td>
<td>( EC_{4,8} ) ( EC_5 )</td>
</tr>
<tr>
<td>( EC_{4,8} )</td>
<td>( EC_{4,8} ) ( EC_{5,9} )</td>
</tr>
<tr>
<td>( EC_5 )</td>
<td>( EC_{4,8} ) ( EC_5 )</td>
</tr>
<tr>
<td>( EC_{5,9} )</td>
<td>( EC_{4,8} ) ( EC_{5,10} )</td>
</tr>
<tr>
<td>( EC_{5,10} )</td>
<td>( EC_{4,8} ) ( EC_{5,9} )</td>
</tr>
</tbody>
</table>
The Big Picture
Lexical Analysis: Problems

1. Write a regular expression to specify all numbers in binary form that are multiples 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.
Formal languages: Definition, Recognition, Generation

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

**Definition** Regular expressions is another formal language used to represent 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.
4.1. Context-Free Grammars

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 \( \alpha \rightarrow \beta \) 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.

Note. Every sentence is also a sentential form. We use small greek letters (\( \alpha, \beta \) etc.) to denote sentential forms and small roman letters at the end of the roman alphabet (\( x, y, z \) etc.) to denote sentences.
Definition 4.3 A grammar $G = \langle N, T, P, S \rangle$ is called context-free if:

- each production is of the form $X \rightarrow \alpha$, where
  - $X \in N$ is a nonterminal and
  - $\alpha \in (N \cup T)^*$ is a sentential form.
- The production is terminal if $\alpha$ is a sentence.
CFG: Example 1

\[ G = \langle \{S\}, \{a,b\}, P, S\rangle \], where \( S \rightarrow ab \) and \( S \rightarrow aSb \) 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

\[
\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \alpha_n
\]  

(1)

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

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

Definition 4.5 A derivation is complete if \( \alpha_n \in T^* \). Then \( \alpha_n \) is said to have been generated by the grammar.
Language Generation

Definition 4.6

Definition 4.7 *The language generated* by a context-free grammar is the set of sentences that can be generated.

Example 4.8 \( L(G) \), the language generated by \( G \) is \( \{a^n b^n | n > 0 \} \).

Actually can be proved by induction on the length and structure of derivations.
Regular Grammars

Definition 4.9 A production rule of a context-free grammar is

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

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

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

Definition 4.10 A regular grammar is a context-free grammar whose productions are either only right linear or only left linear.
CFG: Empty word

\[G = \langle \{S\}, \{a, b\}, P, S \rangle, \text{ where } S \rightarrow SS \mid aSb \mid \varepsilon\]
generates all sequences of matching nested parentheses, including the empty word \(\varepsilon\).

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 \textit{God alone knows what!}

\[S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow SS \Rightarrow \ldots\]

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 non-terminal was used to get the next member of the sequence.
• Of course, generation of the empty word $\varepsilon$ must be shown explicitly in the tree.
CFG: Derivation trees 2

Derivation tree of \(abaabb\)
CFG: Derivation trees 3

Another Derivation tree of abaabb
Yet another Derivation tree of abaabb
4.2. Ambiguity

Ambiguity Disambiguation
Ambiguity: 1

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

\[
\begin{align*}
E & \rightarrow I \mid C \mid E+E \mid E*E \\
I & \rightarrow y \mid z \\
C & \rightarrow 4
\end{align*}
\]

Consider the sentence \( y + 4 * z \).
Ambiguity: 2

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

\[
\begin{align*}
E & \rightarrow I \mid C \mid E + E \mid E \ast E \\
I & \rightarrow y \mid z \\
C & \rightarrow 4
\end{align*}
\]

Consider the sentence $y + 4 \ast z$. 
Ambiguity: 3

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

\[
\begin{align*}
E & \rightarrow I \mid C \mid E + E \mid E * E \\
I & \rightarrow y \mid z \\
C & \rightarrow 4
\end{align*}
\]

Consider the sentence \( y + 4 * z \).
Ambiguity: 4

\[ G_1 = \langle \{E, I, C\}, \{y, z, 4, \ast, +\}, P_1, \{E\} \rangle \] where \( P_1 \) consists of the following productions.

\[
E \rightarrow I \mid C \mid E+E \mid E\ast E \\
I \rightarrow y \mid z \\
C \rightarrow 4
\]

Consider the sentence \( y + 4 \ast z \).
Ambiguity: 5

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

\[
\begin{align*}
E & \rightarrow I \mid C \mid E+E \mid E*E \\
I & \rightarrow y \mid z \\
C & \rightarrow 4
\end{align*}
\]

Consider the sentence \( y + 4 \ast z \).
Left-most Derivation 1

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

\[
\begin{align*}
E & \quad \Rightarrow \\
E+E & \quad \Rightarrow \\
I+E & \quad \Rightarrow \\
y+E & \quad \Rightarrow \\
y+E*E & \quad \Rightarrow \\
y+C*E & \quad \Rightarrow \\
y+4*E & \quad \Rightarrow \\
y+4*I & \quad \Rightarrow \\
y + 4 * z
\end{align*}
\]
Left-most Derivation 2

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

$$
E \quad \Rightarrow \\
E*E \quad \Rightarrow \\
E+E*E \quad \Rightarrow \\
I+E*E \quad \Rightarrow \\
y+E*E \quad \Rightarrow \\
y+C*E \quad \Rightarrow \\
y + 4*E \quad \Rightarrow \\
y + 4*I \quad \Rightarrow \\
y + 4 * z
$$
Right-most Derivation 1

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

$$
\begin{align*}
E & \Rightarrow \\
E+E & \Rightarrow \\
E+E*E & \Rightarrow \\
E+E*I & \Rightarrow \\
E+E*z & \Rightarrow \\
E+C*z & \Rightarrow \\
E+4*z & \Rightarrow \\
I+4*z & \Rightarrow \\
y + 4*z & \Rightarrow 
\end{align*}
$$
Right-most Derivation 2

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

$$
E \quad \Rightarrow \\
E*E \quad \Rightarrow \\
E*I \quad \Rightarrow \\
E*z \quad \Rightarrow \\
E+E*z \quad \Rightarrow \\
E+C*z \quad \Rightarrow \\
E+4*z \quad \Rightarrow \\
I+4*z \quad \Rightarrow \\
y + 4*z
$$
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*
Disambiguation

The only way to remove ambiguity (without changing the language generated) is to change the grammar by introducing some more non-terminal symbols and changing the production rules. Consider the grammar

\[ G_1' = \langle N', \{y, z, 4, *, +\}, P', \{E\} \rangle \]

where \( N' = N \cup \{T, F\} \) with the following production rules \( P' \).

\[
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow I \mid C \\
I \rightarrow y \mid z \\
C \rightarrow 4
\]

and compare it with the grammar \( G_1 \)
Left-most Derivation 1’

The left-most derivation of \( y + 4 \times z \) is then as follows.

\[
\begin{align*}
E & \Rightarrow \\
E + T & \Rightarrow \\
I + T & \Rightarrow \\
y + T & \Rightarrow \\
y + T \times F & \Rightarrow \\
y + T \times F & \Rightarrow \\
y + F \times F & \Rightarrow \\
y + C \times F & \Rightarrow \\
y + 4 \times F & \Rightarrow \\
y + 4 \times I & \Rightarrow \\
y + 4 \times z & 
\end{align*}
\]
Left-most Derivations

Compare it with the Left-most Derivation 1.

\[ G_1. \quad E \Rightarrow E+E \Rightarrow I+E \Rightarrow y+E \Rightarrow y+E*E \Rightarrow \]
\[ y+C*E \Rightarrow y+4*E \Rightarrow y+4*I \Rightarrow y+4*z \]

\[ G'_1. \quad E \Rightarrow E+T \Rightarrow I+T \Rightarrow y+T \Rightarrow y+T*F \Rightarrow y+T*F \Rightarrow y+F*F \Rightarrow \]
\[ y+C*F \Rightarrow y+4*F \Rightarrow y+4*I \Rightarrow y+4*z \]

There is no derivation in \( G'_1 \) corresponding to Left-most Derivation 2 (Why not?).
Right-most Derivation 1’

Right-most derivation of \( y + 4 \times z \) corresponding to the \textit{first} derivation tree.

\[
E \quad \Rightarrow \\
E + T \quad \Rightarrow \\
E + T \ast F \quad \Rightarrow \\
E + T \ast I \quad \Rightarrow \\
E + T \ast z \quad \Rightarrow \\
E + C \ast z \quad \Rightarrow \\
E + 4 \ast z \quad \Rightarrow \\
F + 4 \ast z \quad \Rightarrow \\
I + 4 \ast z \quad \Rightarrow \\
+ 4 \ast z \quad \Rightarrow \\
y + 4 \ast z
\]

Compare it with the \textbf{Right-most Derivation 1}.

There is no derivation corresponding to \textbf{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 4.11** Compare the following fully parenthesized grammar $G_2$ (which has the extra terminal symbols ( and )) with the grammar $G_1$ without parentheses

$$E \rightarrow I \mid C \mid (E+E) \mid (E*E)$$

$$I \rightarrow y \mid z$$

$$C \rightarrow 4$$

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**. $\ast$ has higher precedence than $\pm$.

**Associativity.** $\ast$ and $\pm$ are both left associative operators.
Context-Free Grammars: Problems

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 $(, )$, 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 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.
4.3. Shift-Reduce Parsing

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.
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

\[
\begin{align*}
y + 4 \ast z & \iff \\
I + 4 \ast z & \iff \\
E + 4 \ast z & \iff \\
E + C \ast z & \iff \\
E + E \ast z & \iff \\
E \ast z & \iff \\
E \ast I & \iff \\
E \ast E & \iff \\
E & \iff 
\end{align*}
\]
Bottom-Up Parsing Strategy

The main problem is to match parentheses of arbitrary nesting depths. This requires a stack 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 $\alpha$ in the parsing stack to $X$. 
r1. E → E - T
r2. E → T
r3. T → T (D)
( D)

r5. D → a / b ( E)
Principle:
Reduce whenever possible.
Shift only when reduce is impossible.
Parsing: 2

\[
\begin{align*}
    r1. & \quad E \rightarrow E \cdot T \\
    r2. & \quad E \rightarrow T \\
    r3. & \quad T \rightarrow T \cup D \\
    r4. & \quad T \rightarrow D \\
    r5. & \quad D \rightarrow a \mid b \mid ( E ) \\
\end{align*}
\]

Reduce by r5
r1. $E \rightarrow E \bullet T$

$r2 \quad E \rightarrow T$

$r3 \quad T \rightarrow T \cup D$

$r4 \quad T \rightarrow D$

$r5 \quad D \rightarrow a \mid b \mid (E)$

Reduce by $r4$
Parsing: 4

r1. $E \rightarrow E - T$

r2 $E \rightarrow T$

r3 $T \rightarrow T \{\} D$

r4 $T \rightarrow D$

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

Reduce by r2
Parsing: 5

r1. E → E T
r2. E → T
r3. T → T ( ) D
r4. T → D
r5. D → a | b | ( E )

Shift
Parsing: 6

r1. \( E \rightarrow E - T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \lor D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a | b | ( E ) \)

Shift

\( \lor \) \( b \)

\( a \)

\( E \)
Parsing: 7

r1. E → E T
r2. E → T
r3. T → T ( D
r4. T → D
r5. D → a | b | ( E )

Reduce by r5
Parsing: 8

r1. E → E T
r2. E → T
r3. T → T ( D
r4. T → D
r5. D → a | b | ( E )

Reduce by r4
Parsing: 8a

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \cdot D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r4
Parsing: 9a

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a | b | (E)$

Reduce by $r1$
Parsing: 10a

1. \( E \rightarrow E T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T ( \) D
4. \( T \rightarrow D \)
5. \( D \rightarrow a \mid b \mid ( E ) \)

Shift

\( b \)
Parsing: 11a

r1. $E \rightarrow E\textbf{−}T$

r2. $E \rightarrow T$

r3. $T \rightarrow T\textbf{} D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid ( E )$

Shift
Parsing: 12a

r1. $E \rightarrow E \cdot T$
r2. $E \rightarrow T$
r3. $T \rightarrow T \cdot D$
r4. $T \rightarrow D$
r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r5
Parsing: 13a

\[
\begin{align*}
\text{r1. } & E \rightarrow E \rightarrow T \\
\text{r2. } & E \rightarrow T \\
\text{r3. } & T \rightarrow T \cup D \\
\text{r4. } & T \rightarrow D \\
\text{r5. } & D \rightarrow a \mid b \mid (E) \\
\end{align*}
\]

Reduce by r4
Parsing: 14a

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \uparrow D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid ( E )$

Stuck!

Reduce by r2

Get back!
Parsing: 14b

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: 13b

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \cdot D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Get back!

Reduce by r4
Parsing: 12b

1. $E \rightarrow E \cdot T$
2. $E \rightarrow T$
3. $T \rightarrow T \cup D$
4. $T \rightarrow D$
5. $D \rightarrow a \mid b \mid (E)$

Reduce by r5

Get back!
Parsing: 11b

r1. \[ E \rightarrow E \cdot T \]
r2. \[ E \rightarrow T \]
r3. \[ T \rightarrow T(\mathbb{L})\ D \]
r4. \[ T \rightarrow D \]
r5. \[ D \rightarrow a \ | \ b \ | \ (E) \]

Get back!

Shift
Parsing: 10b

r1. \( E \rightarrow E \, T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \, ( \, D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid ( \, E \) \)

Get back!

Shift
Parsing: 9b

Get back to where you once belonged!

Reduce by r1
Parsing: 8b

Principle:
Reduce whenever possible, but depending upon lookahead.

Modified
Shift instead of reduce here!
Reducing by rule r4:
Parsing: 9

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T ( D$

r4. $T \rightarrow D$

r5. $D \rightarrow a | b | ( E )$

Shift
Parsing: 10

r1. \( E \rightarrow E \cdot T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \cdot D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a \mid b \mid (E) \)

Shift
Parsing: 11

r1. $E \rightarrow E - T$
r2. $E \rightarrow T$
r3. $T \rightarrow T \cup D$
r4. $T \rightarrow D$
r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r5
Parsing: 12

r1. $E \rightarrow E \text{−} T$
r2. $E \rightarrow T$
r3. $T \rightarrow T \text{−} D$
r4. $T \rightarrow D$
r5. $D \rightarrow \text{a} | \text{b} | (E)$

Reduce by $r3$
r1. $E \rightarrow E\ T$

r2. $E \rightarrow T$

r3. $T \rightarrow T\ \{\}\ D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \ | \ b \ | \ (E)\$

Reduce by r1
4.4. Bottom-Up Parsing

Bottom-Up Parsing
Parse Trees: 0

r1. \( E \rightarrow E \rightarrow T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T^* \rightarrow D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid ( \rightarrow E \rightarrow ) \)

shift-reduce parsing: 0
Parse Trees: 1

\[
\begin{align*}
\text{r1. } & E & \rightarrow & E & T \\
\text{r2. } & E & \rightarrow & T \\
\text{r3. } & T & \rightarrow & T & D \\
\text{r4. } & T & \rightarrow & D \\
\text{r5. } & D & \rightarrow & a & | & b & | & (E) \\
\end{align*}
\]
Parse Trees: 2

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a | b | ( E )$

shift-reduce parsing: 2
Parse Trees: 3

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

shift-reduce parsing: 3
Parse Trees: 3a

\[
\begin{align*}
    r1. & \quad E \rightarrow E - T \\
    r2. & \quad E \rightarrow T \\
    r3. & \quad T \rightarrow T / D \\
    r4. & \quad T \rightarrow D \\
    r5. & \quad D \rightarrow a \mid b \mid (E)
\end{align*}
\]

shift-reduce parsing
Parse Trees: 3b

r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

shift-reduce parsing
Parse Trees: 4

r1. \( E \rightarrow E \quad T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \quad D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a \mid b \mid (E) \)

shift-reduce parsing
Parse Trees: 5

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T(\cdot D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

shift-reduce parsing
Parse Trees: 5a

r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \# D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid ( E ) \)

shift-reduce parsing
Parse Trees: 5b

r1. \[ E \rightarrow E \mathbin{\text{−}} T \]
r2. \[ E \rightarrow T \]
r3. \[ T \rightarrow T \mathbin{\text{/}} D \]
r4. \[ T \rightarrow D \]
r5. \[ D \rightarrow a \mid b \mid ( \text{E} ) \]

shift-reduce parsing
Parse Trees: 6

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a | b | ( E )$

shift-reduce parsing
Parse Trees: 7

r1. $E \rightarrow E \mathbf{T}$

r2. $E \rightarrow T$

r3. $T \rightarrow T \mathbf{D}$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

shift-reduce parsing
Parse Trees: 8

\[
\begin{align*}
&\text{r1. } E \rightarrow E - T \\
&\text{r2. } E \rightarrow T \\
&\text{r3. } T \rightarrow T \cdot D \\
&\text{r4. } T \rightarrow D \\
&\text{r5. } D \rightarrow a \mid b \mid (E) \\
\end{align*}
\]

shift-reduce parsing
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 possibility 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
4.5. Recursive Descent Parsing

Recursive Descent Parsing

- Suitable for grammars that are LL(1)
- 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
Caveats with RDP: Left Recursion

Any direct or indirect 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. In particular,

- Production rules cannot be left-recursive i.e. they should not be of the form $A \rightarrow A\alpha$. This would result in an infinite recursion with no input token consumed.

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

**Example 4.12**

\[
A \rightarrow B\beta \\
B \rightarrow A\alpha
\]

*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 4.13** A set of productions of the form

\[ A \rightarrow aB\beta \mid aC\gamma \]

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

\[ A \rightarrow aA' \]
\[ A' \rightarrow B\beta \mid C\gamma \]

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

The grammar used in shift-reduce parsing 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:

$$E \rightarrow TE'$$
$$E' \rightarrow -TE' | \varepsilon$$
$$T \rightarrow DT'$$
$$T' \rightarrow /DT' | \varepsilon$$
$$D \rightarrow a | b | (E)$$

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

4.6.1. The Extended Backus-Naur Form (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.

**Start symbol.** The rules are written usually in a “top-down fashion” and 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 usually enclosed in double-quotes when written in monochrome (we shall additionally colour-code them).
Balanced Parentheses: CFG

Example 4.14 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

\[
S \rightarrow \epsilon, \\
S \rightarrow (S)S, \\
S \rightarrow [S]S, \\
S \rightarrow \{S\}S
\]
Balanced Parentheses: EBNF

Example 4.15 $BP_3$ may be expressed in EBNF as follows:

\[
\begin{align*}
\text{BracketSeq} & \::= \{ \text{Bracket} \} . \\
\text{Bracket} & \::= \text{LeftParen} \text{BracketSeq} \text{RightParen} \mid \\
& \quad \text{LeftSqrbracket} \text{BracketSeq} \text{RightSqrbracket} \mid \\
& \quad \text{LeftBrace} \text{BracketSeq} \text{RightBrace} . \\
\text{LeftParen} & \::= \text{"("} . \\
\text{RightParen} & \::= \text{"")"} . \\
\text{LeftSqrbracket} & \::= \text{"["} . \\
\text{RightSqrbracket} & \::= \text{"]"} . \\
\text{LeftBrace} & \::= \text{"{"} . \\
\text{RightBrace} & \::= \text{"}" .
\end{align*}
\]
EBNF in EBNF

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

```
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} .
```
EBNF: Character Set

The character set used in EBNF is described below.

**Character** ::= Letter | Digit | SpecialChar

**Letter** ::= UpperCase | LowerCase


**LowerCase** ::= “a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”

**Digit** ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

**SpecialChar** ::= “!” | “)” | “#” | “$” | “%” | “&” | “)” | “(” | “*” | “+” | “,” | “-” | “.” | “/” | “:” | “;” | “<” | “=” | “>” | “?” | “@” | “[” | “\” | “]” | “^” | “_” | “’” | “{” | “|” | “}” | “~”
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.
- Pascal has been defined using both the text-version of EBNF and through syntax diagrams.
- EBNF also has a diagrammatic rendering. The grammar of SML has been produced by a set of 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.
Exercise 4.1

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.
5. Attributes & Semantic Analysis
Semantic Analysis: 1

• Every Programming language can be used to program any computable function, assuming of course, it has
  – unbounded memory, and
  – unbounded time

• The parser of a programming language provides the framework within which 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.

• However, context-free grammars are not powerful enough to represent all computable functions. Example, the language \{a^n b^n c^n | n > 0\}. 
Semantic Analysis: 2

- There are context-sensitive aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - correspondence between formal and actual parameters
  - type consistency between declaration and use.
  - scope and visibility issues with respect to identifiers in a program.
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.
Attributes

An attribute can represent anything we choose e.g.

- a string
- a number
- 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.
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 parsing at the stage when a reduction needs to be performed by the given rule. Hence along with the reduction the corresponding semantic rules are also executed.

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 $a_1, \ldots, a_k$ are attributes belonging to the grammar symbols of the production and

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

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

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

An attribute grammar is a syntax-directed definition in which the functions in semantic rules can have no side-effects.

In general for different occurrences of non-terminal symbols in the same 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

Determining the values of arithmetic expressions. Consider a simple attribute \textit{val} associated with an expression

\[
E_0 \rightarrow E_1 - T \quad \triangleright \quad E_0.val := E_1.val - T.val
\]

\[
E \rightarrow T 
\quad \triangleright \quad E.val := T.val
\]

\[
T_0 \rightarrow T_1 / F
\quad \triangleright \quad T_0.val := T_1.val / F.val
\]

\[
T \rightarrow F
\quad \triangleright \quad T.val := F.val
\]

\[
F \rightarrow (E)
\quad \triangleright \quad F.val := E.val
\]

\[
F \rightarrow n
\quad \triangleright \quad F.val := n.val
\]
Attributes: Basic Assumptions

- Terminal symbols are assumed to have only synthesized attributes. Their attributes are all supplied by the lexical analyser during scanning.
- The start symbol of the 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: 0
Synthesized Attributes: 1
Synthesized Attributes: 2
Synthesized Attributes: 3
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

\[
E_0 \rightarrow E_1 - T \quad \rightarrow E_0.val := \text{sub}(E_1.val, T.val)
\]

\[
E \rightarrow T \quad \rightarrow E.val := T.val
\]

\[
T_0 \rightarrow T_1 / F \quad \rightarrow T_0.val := \text{div}(T_1.val, F.val)
\]

\[
T \rightarrow F \quad \rightarrow T.val := F.val
\]

\[
F \rightarrow (E) \quad \rightarrow F.val := E.val
\]

\[
F \rightarrow n \quad \rightarrow F.val := n.val
\]
Evaluation of Synthesized Attributes

During 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 $Z \rightarrow Y_1 \ldots Y_k$ compute the attribute values of $Z$ from the attribute values of $Y_1, \ldots, Y_k$ 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.

**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.
Synthesized and Inherited Attributes

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 $a_1, \ldots, a_k$ are attributes belonging to the grammar symbols of the production and

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

Inherited: $a$ is an inherited attribute of one of the grammar symbols of $\alpha$ (denoted $B.b$ if $b$ is an attribute of $B$).

In each case the attribute $b$ is said to depend upon the the attributes $a_1, \ldots, a_k$. 
Inherited Attributes: 0

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

\[
D \rightarrow TL \\
L \rightarrow LL | I \\
T \rightarrow \text{int} | \text{float} \\
I \rightarrow x | y | z
\]
Inherited Attributes: 1

C-style declarations generating \( \text{int } x, y, z \).

\[
D \rightarrow TL \\
L \rightarrow L, L I \mid I \\
T \rightarrow \text{int } \mid \text{float} \\
I \rightarrow x \mid y \mid z
\]
Inherited Attributes: 2

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

\[
D \rightarrow TL \\
L \rightarrow LI | I \\
T \rightarrow \text{int} | \text{float} \\
I \rightarrow x | y | z
\]
Inherited Attributes: 3

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

\[
D \rightarrow TL \\
L \rightarrow L,I \mid I \\
T \rightarrow \text{int} \mid \text{float} \\
I \rightarrow x \mid y \mid z
\]
Inherited Attributes: 4

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

\[
D \rightarrow TL \\
L \rightarrow LI | I \\
T \rightarrow \text{int} \ | \ \text{float} \\
I \rightarrow x \ | \ y \ | \ z
\]
Inherited Attributes: 5
Inherited Attributes: 6

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

\[
D \rightarrow TL \\
L \rightarrow L, I | I \\
I \rightarrow x | y | z
\]
Inherited Attributes: 7

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

$$D \rightarrow TL$$

$$L \rightarrow L, I \mid I$$

$$T \rightarrow \text{int} \mid \text{float}$$

$$I \rightarrow x \mid y \mid z$$
An Attribute Grammar

\[ D \rightarrow TL \quad \Rightarrow \quad L.in := T.type \]

\[ T \rightarrow \text{int} \quad \Rightarrow \quad T.type := \text{int.int} \]

\[ T \rightarrow \text{float} \quad \Rightarrow \quad T.type := \text{float.float} \]

\[ L_0 \rightarrow L_1.I \quad \Rightarrow \quad L_1 := L_0.in \]

\[ L \rightarrow I \quad \Rightarrow \quad L.in := L.in \]

\[ I \rightarrow \text{id} \quad \Rightarrow \quad \text{id.type} := I.in \]
L-attributed Definitions

A grammar is \textit{L-attributed} if for each production of the form $\gamma \rightarrow X_1 \ldots X_k$, each inherited attribute of the symbol $X_j$, $1 \leq j \leq k$ depends only on

1. the inherited attributes of the symbol $\gamma$ and

2. the synthesized or inherited attributes of $X_1, \ldots, X_{j-1}$. 
Why L-attributedness?

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.

- if the value of $X_j.inh$ depends upon the attributes of one or more of the symbols $X_{j+1}, \ldots, X_k$ then the computation of $X_j.inh$ cannot be performed just before the reduction by the rule $Y \rightarrow X_1 \ldots X_k$ during parsing. Instead it may have to be postponed till the end of parsing.

- 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.

- 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.

\[
\begin{align*}
D & \rightarrow L:T \triangleright L.in := T.type \\
T & \rightarrow \text{int} \triangleright T.type := \text{int}.int \\
T & \rightarrow \text{real} \triangleright T.type := \text{real}.real \\
L_0 & \rightarrow L_1,I \triangleright L_1 := L_0.in \\
L & \rightarrow I \triangleright I.in := L.in \\
I & \rightarrow \text{id} \triangleright \text{id}.type := I.in
\end{align*}
\]

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

In many languages like ML which allow higher order functions as values, a definition not being L-attributed may not be of serious concern. 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. The language of the grammar of Pascal and ML declarations can be generated as follows:

\[
D \rightarrow \text{id}L \triangleright \text{addtype(id, L.type)}
\]
\[
L \rightarrow :T \triangleright L.in := T.type
\]
\[
L \rightarrow ,\text{id}L \triangleright L_0.type := L_1.type;
\]
\[
\text{addtype(id.L}_1.type)
\]
\[
T \rightarrow \text{int} \triangleright T.type := \text{int.int}
\]
\[
T \rightarrow \text{real} \triangleright T.type := \text{real.real}
\]
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.
Dependency Graph Construction

Algorithm 4 Attribute Dependency Graph Construction

Require: A parse tree of a CFG and the list of attributes
Ensure: A dependency graph

for all nodes $n$ of the parse tree do
  for all attributes $a$ of node $n$ do
    Create an attribute node $n.a$
  end for
end for

for all nodes $n$ of the parse tree do
  for all semantic rules $a := f(b_1, \ldots, b_k)$ do
    for all $i : 1 \leq i \leq k$ do
      Create a directed edge $b_i \rightarrow a$
    end for
  end for
end for
6. Abstract Syntax
Abstract Syntax Trees

The construction of ASTs from concrete parse trees is another 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.
Abstract Syntax: 0

\[
E \rightarrow E - T \mid T \\
T \rightarrow T / F \mid F \\
F \rightarrow n \mid (E)
\]

Suppose we want to evaluate an expression \((4 - 1)/2\). What we actually want is a tree that looks like this:
Evaluation: 0
Evaluation: 1
Evaluation: 2

```
1

2

3
```
Evaluation: 3

Diagram:

1
/
3
/
2
But what we actually get during parsing is a tree that looks like ...
Abstract Syntax: 1

...THIS!
Abstract Syntax

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

\[
E_0 \rightarrow E_1-T \quad \triangleright \quad E_0.ptr := \text{makeBinaryNode}(\ -, E_1.ptr, T.ptr)
\]

\[
E \rightarrow T 
\quad \triangleright \quad E.ptr := T.ptr
\]

\[
T_0 \rightarrow T_1/F 
\quad \triangleright \quad T_0.ptr := \text{makeBinaryNode}(/, T_1.ptr, F.ptr)
\]

\[
T \rightarrow F 
\quad \triangleright \quad T.ptr := F.ptr
\]

\[
F \rightarrow (E) 
\quad \triangleright \quad F.ptr := E.ptr
\]

\[
F \rightarrow n 
\quad \triangleright \quad F.ptr := \text{makeLeaf}(n.val)
\]

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.

\textbf{makeLeaf(literal)} : Creates a node with label \texttt{literal} and returns the AST.

\textbf{makeBinaryNode(opr, opd1, opd2)} : Creates a tree with root label \texttt{opr} (with sub-trees \texttt{opd1} and \texttt{opd2}).

Now we may associate a \textit{synthesized} attribute called \texttt{ast} with each terminal and nonterminal symbol which points to the root of the subtree created for it.
Abstract Syntax: Functional

\[ E_0 \rightarrow E_1 \cdot T \quad \Rightarrow \quad E_0.ast := makeBinaryNode( -, E_1.ast, T.ast) \]

\[ E \rightarrow T \quad \Rightarrow \quad E.ast := T.ast \]

\[ T_0 \rightarrow T_1 / F \quad \Rightarrow \quad T_0.ast := makeBinaryNode( /, T_1.ast, F.ast) \]

\[ T \rightarrow F \quad \Rightarrow \quad T.ast := F.ast \]

\[ F \rightarrow (E) \quad \Rightarrow \quad F.ast := E.ast \]

\[ F \rightarrow n \quad \Rightarrow \quad F.ast := makeLeaf(n.val) \]

The Big Picture
7. The Pure Untyped Lambda Calculus: Basics
Pure Untyped $\lambda$-Calculus: Syntax

The language $\Lambda$ of pure untyped $\lambda$-terms is the smallest set of terms built up from an infinite set $V$ of variables and closed under the following productions

$$L, M, N ::= x \quad \text{Variable}$$
$$\quad \mid \lambda x[L] \quad \text{Abstraction}$$
$$\quad \mid (L M) \quad \text{Application}$$

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.
Free and Bound Variables

Definition 7.1 For any term $N$ the set of free variables and the set of all variables are defined by induction on the structure of terms.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$FV(N)$</th>
<th>$Var(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>${x}$</td>
<td>${x}$</td>
</tr>
<tr>
<td>$\lambda x[L]$</td>
<td>$FV(L) - {x}$</td>
<td>$Var(L) \cup {x}$</td>
</tr>
<tr>
<td>$(L M)$</td>
<td>$FV(L) \cup FV(M)$</td>
<td>$Var(L) \cup Var(M)$</td>
</tr>
</tbody>
</table>

- 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 $\lambda x[L]$.
- $\Lambda_0 \subseteq \Lambda$ is the set of closed $\lambda$-terms (i.e. terms with no free variables).
Notational Conventions

To minimize use of brackets unambiguously

1. $\lambda x_1 x_2 \ldots x_m [L]$ denotes $\lambda x_1 [\lambda x_2 [\ldots \lambda x_m [L] \ldots ]]$ i.e. $L$ is the scope of each of the variables $x_1, x_2, \ldots x_m$.

2. $(L_1 L_2 \ldots L_m)$ denotes $(\ldots (L_1 L_2) \ldots L_m)$ i.e. application is left-associative.
Substitution

Definition 7.2 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:

\[
\begin{align*}
\{N/x\}x & \equiv N \\
\{N/x\}y & \equiv y & \text{if } y \neq x \\
\{N/x\}\lambda x[L] & \equiv \lambda x[L] \\
\{N/x\}\lambda y[L] & \equiv \lambda y[\{N/x\}L] & \text{if } y \neq x \text{ and } y \notin FV(N) \\
\{N/x\}\lambda y[L] & \equiv \lambda z[\{N/z\}{z/x}L] & \text{if } y \neq x \text{ and } y \in FV(N) \text{ and } z \text{ is 'fresh'} \\
\{N/x\}(L M) & \equiv (\{N/x\}L \{N/x\}M)
\end{align*}
\]

- In the above definition it is necessary to ensure that the free variables of \( N \) continue to remain free after substitution.
- The phrase “\( z \) is 'fresh'” may be taken to mean \( z \notin FV(N) \cup Var(L) \).
- \( z \) could be fresh even if \( z \in BV(N) \).
\( \alpha \)-equivalence

**Definition 7.3** (\( \alpha \)-equivalence) \( \lambda x[L] \equiv_\alpha \lambda y[y/x]L \) provided \( y \notin \text{Var}(L) \).

- Here again if \( y \in FV(L) \) it must not be captured by a change of bound variables.
Untyped $\lambda$-Calculus: Basic $\beta$-Reduction

Definition 7.4

- Any (sub-)term of the form $(\lambda x[L] M)$ is called a $\beta$-redex
- Basic $\beta$-reduction is the relation on $\Lambda$
  \[
  \rightarrow_\beta \overset{df}{=} \{((\lambda x[L] M), \{M/x\}L') \mid L' \equiv_\alpha L, L', L, M \in \Lambda\}
  \]
- It is usually represented by the axiom
  \[
  (\lambda x[L] M) \rightarrow_\beta \{M/x\}L'
  \]
  where $L' \equiv_\alpha L$.\[\text{ (2)}\]
Untyped $\lambda$-Calculus: 1-step $\beta$-Reduction

**Definition 7.5** A 1-step $\beta$-reduction $\rightarrow^{1}_{\beta}$ is the smallest relation (under the $\subseteq$ ordering) on $\Lambda$ such that

\[
\begin{align*}
\beta_1 \quad & L \rightarrow^{1}_{\beta} M \\
\hline 
L \rightarrow^{1}_{\beta} M 
\end{align*}
\]

\[
\begin{align*}
\beta_1 \text{Abs} & \quad \lambda x[L] \rightarrow^{1}_{\beta} \lambda x[M] \\
\hline 
L \rightarrow^{1}_{\beta} M 
\end{align*}
\]

\[
\begin{align*}
\beta_1 \text{AppL} & \quad L \rightarrow^{1}_{\beta} M \\
\hline 
(L \ N) \rightarrow^{1}_{\beta} (M \ N) 
\end{align*}
\]

\[
\begin{align*}
\beta_1 \text{AppR} & \quad L \rightarrow^{1}_{\beta} M \\
\hline 
(N \ L) \rightarrow^{1}_{\beta} (N \ M) 
\end{align*}
\]

- $\rightarrow^{1}_{\beta}$ is the **compatible closure** of basic $\beta$-reduction to all contexts.
- We will often omit the superscript $^{1}$ as understood.
Untyped $\lambda$-Calculus: $\beta$-Reduction

Definition 7.6

• For all integers $n \geq 0$, $n$-step $\beta$-reduction $\rightarrow_\beta^n$ is defined by induction on $1$-step $\beta$-reduction

\[
\begin{align*}
\beta_n \text{Basis} & \quad L \rightarrow_\beta^0 L \\
\beta_n \text{Induction} & \quad \frac{L \rightarrow_\beta M \rightarrow_\beta^1 N}{L \rightarrow_\beta^{m+1} N} \quad (m \geq 0)
\end{align*}
\]

• $\beta$-reduction $\rightarrow_\beta^*$ is the reflexive-transitive closure of $1$-step $\beta$-reduction. That is,

\[
\begin{align*}
\beta^* & \quad \frac{L \rightarrow_\beta^n M}{L \rightarrow_\beta^* M} \quad (n \geq 0)
\end{align*}
\]
Untyped $\lambda$-Calculus: Normalization

Definition 7.7

- A term is called a $\beta$-normal form ($\beta$-nf) if it has no $\beta$-redexes.
- A term is weakly normalising ($\beta$-WN) if it reduces to a $\beta$-normal form.
- A term $L$ is strong normalising ($\beta$-SN) if it has no infinite reduction sequence $L \rightarrow^{\beta} L_1 \rightarrow^{\beta} L_2 \rightarrow^{\beta} \cdots$
Untyped $\lambda$-Calculus: Examples

Example 7.8

1. $K \overset{df}{=} \lambda x\ y[x],\ I \overset{df}{=} \lambda x[x],\ S \overset{df}{=} \lambda x\ y\ z[((x\ z)\ (y\ z))],\ \omega \overset{df}{=} \lambda x[(x\ x)]$ are all $\beta$-nfs.

2. $\Omega \overset{df}{=} (\omega\ \omega)$ has no $\beta$-nf. Hence it is neither weakly nor strongly normalising.

3. $(K\ (\omega\ \omega))$ cannot reduce to any normal form because it has no finite reduction sequences. All its reductions are of the form

   $$(K\ (\omega\ \omega)) \xrightarrow{\beta} (K\ (\omega\ \omega)) \xrightarrow{\beta} (K\ (\omega\ \omega)) \xrightarrow{\beta} \cdots$$

   or at some point it could transform to

   $$(K\ (\omega\ \omega)) \xrightarrow{\beta} \lambda y[(\omega\ \omega)] \xrightarrow{\beta} \lambda y[(\omega\ \omega)] \xrightarrow{\beta} \cdots$$

4. $((K\ \omega)\ \Omega)$ is weakly normalising because it can reduce to the normal form $\omega$ but it is not strongly normalising because it also has an infinite reduction sequence

   $$((K\ \omega)\ \Omega) \xrightarrow{\beta} ((K\ \omega)\ \Omega) \xrightarrow{\beta} \cdots$$
Examples of Strong Normalization

Example 7.9

1. \(((K \omega) \omega)\) is strongly normalising because it reduces to the normal form \(\omega\) in two \(\beta\)-reduction steps.

2. Consider the term \(((S K) K)\). Its reduction sequences go as follows:

\[
((S K) K) \rightarrow^{1}_{\beta} \lambda z[((K z) (K z))] \rightarrow^{1}_{\beta} \lambda z[z] \equiv I
\]
8. 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, in the sense that while \( (p \ 2) \) deterministically produces \( 3.2.2 + 4.2 + 1 \) which in turn simplifies deterministically to \( 21 \), 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:

\[ (p \ 2) \rightarrow 3.2.2 + 4.2 + 1 \rightarrow 21 \]

And yet they are behaviourally equivalent and mutually substitutable in all contexts (referentially transparent).

1. Reduction (specifically \( \beta \)-reduction) captures this asymmetry.
2. Since reduction produces behaviourally equal terms we have the following notion of equality.
Untyped \( \lambda \)-Calculus: \( \beta \)-Equality

**Definition 8.1** \( \beta \)-equality or \( \beta \)-conversion (denoted \( =_\beta \)) is the smallest equivalence relation containing \( \beta \)-reduction (\( \rightarrow^\ast_\beta \)).

The following are equivalent definitions.

1. \( =_\beta \) is the reflexive-symmetric-transitive closure of 1-step \( \beta \)-reduction.
2. \( =_\beta \) is the smallest relation defined by the following rules.

<table>
<thead>
<tr>
<th>( =_\beta ) Basis</th>
<th>( L \overset*{\beta}{\rightarrow} M )</th>
<th>( L =_\beta M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( =_\beta ) Reflexivity</td>
<td>( L =_\beta L )</td>
<td></td>
</tr>
<tr>
<td>( =_\beta ) Symmetry</td>
<td>( L =_\beta M )</td>
<td>( M =_\beta L )</td>
</tr>
<tr>
<td>( =_\beta ) Transitivity</td>
<td>( L =<em>\beta M, M =</em>\beta N )</td>
<td>( L =_\beta N )</td>
</tr>
</tbody>
</table>
Compatibility

**Definition 8.2** 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 $(N L) \rho (N M)$.

**Example 8.3**

1. $\equiv_\alpha$ is a compatible relation
2. $\rightarrow^{1}_\beta$ is by definition a compatible relation.
Compatibility of Beta-reduction and Beta-Equality

Theorem 8.4 $\beta$-reduction $\rightarrow^*_{\beta}$ and $\beta$-equality $=_{\beta}$ are both compatible relations.
Proof of theorem 8.4

Proof:  \((\rightarrow^*_\beta)\) Assume \(L \rightarrow^*_\beta M\). By definition of \(\beta\)-reduction \(L \rightarrow^n_\beta M\) 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 \(L \equiv L_0 \rightarrow^m_\beta L_m \rightarrow^1_\beta M\). Then by the induction hypothesis and the compatibility of \(\rightarrow^1_\beta\) we have

\[
\begin{align*}
&\text{for all } x \in V, \quad \lambda x[L] \rightarrow^m_\beta \lambda x[L_m], \quad \lambda x[L_m] \rightarrow^1_\beta \lambda x[M] \\
&\text{for all } N \in \Lambda, \quad (L N) \rightarrow^m_\beta (L_m N), \quad (L_m N) \rightarrow^1_\beta (M N) \\
&\text{for all } N \in \Lambda, \quad (N L) \rightarrow^m_\beta (N L_m), \quad (N L_m) \rightarrow^1_\beta (N M)
\end{align*}
\]

By definition of \(\rightarrow^n_\beta\)

\[
\begin{align*}
&\text{lambda}[L] \rightarrow^n_\beta \lambda x[M], \\
&(L N) \rightarrow^n_\beta (M N) \\
&(N L) \rightarrow^n_\beta (N M)
\end{align*}
\]

End \((\rightarrow^*_\beta)\)

\((=\beta)\) Assume \(L =\beta M\). We proceed by induction on the length of the proof of \(L =\beta M\) using the definition of \(\beta\)-equality.

Basis. \(n = 1\). Then either \(L \equiv M\) or \(L \rightarrow^*_\beta M\). The case of reflexivity is trivial and the case of \(L \rightarrow^*_\beta M\) 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
By $\equiv_\beta$ Symmetry

for all variables $x$, $\lambda x[M] =_\beta \lambda x[L]$
for all terms $N$, $(M N) =_\beta (L N)$
for all terms $N$, $(N M) =_\beta (N L)$

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

for all variables $x$, $\lambda x[L] =_\beta \lambda x[P]$, $\lambda x[P] =_\beta \lambda x[M]$
for all terms $N$, $(L N) =_\beta (P N)$, $(P N) =_\beta (M N)$
for all terms $N$, $(N L) =_\beta (N P)$, $(N P) =_\beta (N M)$

By $\equiv_\beta$ Transitivity

for all terms $N$, $(M N) =_\beta (N L)$

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,

$$(\lambda x[(M \, x)] \, N) =^\beta (M \, N) \text{ for all terms } N$$

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 $\eta$-reduction as the relation

$$\lambda x[(L \, x)] \rightarrow^\eta L \text{ provided } x \notin FV(L)$$

(3)
Eta-Reduction and Eta-Equality

The following notions are then defined similar to the corresponding notions for $\beta$-reduction.

- **1-step $\eta$-reduction** $\rightarrow^1_\eta$ is the closure of basic $\eta$-reduction to all contexts,
- $\rightarrow^n_\eta$ is defined by induction on 1-step $\eta$-reduction
- **$\eta$-reduction** $\rightarrow^*_\eta$ is the reflexive-transitive closure of 1-step $\eta$-reduction.
- the notions of strong and weak $\eta$ normal forms $\eta$-nf.
- the notion of $\eta$-equality or $\eta$-conversion denoted by $=_{\eta}$.
Exercise 8.1

1. Prove that \( \eta \)-reduction and \( \eta \)-equality are both compatible relations.
2. Prove that \( \eta \)-reduction is strongly normalising.
3. Define basic \( \beta \eta \)-reduction as the application of either (2) or (3). Now prove that \( \rightarrow_{\beta \eta}^{1} \), \( \rightarrow_{\beta \eta}^{*} \) and \( =_{\beta \eta} \) are all compatible relations.

9. Confluence Definitions
Confluence
Reduction Relations

**Definition 9.1** For any binary relation \( \rho \) on \( \Lambda \)

1. \( \rho^1 \) is the compatible closure of \( \rho \)
2. \( \rho^+ \) is the transitive closure of \( \rho^1 \)
3. \( \rho^* \) is the reflexive-transitive-closure of \( \rho^1 \) and is a preorder
4. \( ((\rho^1) \cup (\rho^1)^{-1})^* \) (denoted \( =_\rho \)) is the reflexive-symmetric-transitive closure of \( \rho^1 \) and is an equivalence relation.
5. \( =_\rho \) is also called the equivalence generated by \( \rho \).

We will often use \( \rightarrow \) (suitably decorated) as a reduction relation instead of \( \rho \). Then \( \rightarrow^1, \rightarrow^+, \rightarrow^* \) and \( \leftarrow^* \) denote respectively the compatible closure, the transitive closure, the reflexive transitive closure and the equivalence generated by \( \rightarrow \).
The Diamond Property

Definition 9.2 Let $\rho$ be any relation on terms. $\rho$ has the diamond property if for all $L, M, N$,

$\begin{align*}
M & \quad M \\
\rho & \quad \rho \\
L & \quad \Rightarrow \exists P : \quad P \\
\rho & \quad \rho \\
N & \quad N
\end{align*}$
Reduction Relations: Termination

Let \(\rightarrow\) be a reduction relation, \(\rightarrow^*\) the least preorder containing \(\rightarrow\) and \(\leftarrow^*\) the least equivalence relation containing \(\rightarrow^*\). Then

**Definition 9.3** \(\rightarrow\) is terminating iff there is no infinite sequence of the form

\[
L_0 \rightarrow L_1 \rightarrow \cdots
\]
Reduction: Local Confluence

**Definition 9.4** `def:localconfluence →` is locally confluent if for all `L, M, N`,

\[ N \leftarrow L \rightarrow M \Rightarrow \exists P : N \rightarrow^* P \leftarrow^* M \]

which we denote by

```
  M
  /\  \/
 /    \ /   \nL    * P
\   /   \   /
 \ /    \ /
 N
```
Reduction: Semi-confluence

**Definition 9.5** $\rightarrow$ is semi-confluent if for all $L, M, N$,

$$N \leftarrow L \rightarrow^* M \Rightarrow \exists P : N \rightarrow^* P \leftarrow^* M$$

which we denote by

![Diagram](image.png)
Reduction: Confluence

**Definition 9.6** → is confluent if for all \( L, M, N, \)

\[
N \xleftarrow{*} L \xrightarrow{*} M \Rightarrow \exists P : N \xrightarrow{*} P \xleftarrow{*} M
\]

which we denote as

\[
\begin{array}{c}
M \\
\Rightarrow \exists P \\
L \\
\Rightarrow \exists P \\
N
\end{array}
\]

**Fact 9.7** Any confluent relation is also semi-confluent.
Reduction: Church-Rosser

Definition 9.8 → is Church-Rosser if for all \( L, M \),

\[ L \overset{*}{\leftrightarrow} M \Rightarrow \exists P : L \overset{*}{\rightarrow} P \overset{*}{\leftarrow} M \]

which we denote by

\[ L \overset{*}{\leftrightarrow} M \overset{*}{\rightarrow} \exists P \]
Equivalence Characterization

Lemma 9.9

1. $\leftrightarrow^*$ is the least equivalence containing $\rightarrow$.

2. $\leftrightarrow^*$ is the least equivalence containing $\rightarrow^*$.

3. $L \leftrightarrow^* M$ 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 $i$, $0 \leq i < m$, $M_i \rightarrow M_{i+1}$ or $M_{i+1} \rightarrow M_i$. We represent this fact more succinctly as

$$L \equiv_{\alpha} M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_m \equiv_{\alpha} M$$

(4)
Proof of lemma 9.9

Proof:

1. Just prove that $\rightarrow^*$ is a subset of every equivalence that contains $\rightarrow$.
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” (4) implies $L \rightarrow^* M$ by transitivity. For the other part use induction on the length of the proof.

QED

10. The Church-Rosser Property
Parallel Beta Reduction

Definition 10.1 The parallel-β or $\parallel \beta$ reduction is the smallest relation for which the following rules hold.

$\parallel \beta_1\ App \frac{L \rightarrow^1 \parallel \beta L'}{L M \rightarrow^1 \parallel \beta (L' M')}$

$\parallel \beta_1\ Abs1 \frac{L \rightarrow^1 L'}{\lambda x[L] \rightarrow^1 \parallel \beta \lambda x[L']}$

$\parallel \beta_1\ App \frac{L \rightarrow^1 \parallel \beta L', M \rightarrow^1 \parallel \beta M'}{(L M) \rightarrow^1 \parallel \beta (L' M')}$

$\parallel \beta_1\ Abs2 \frac{L \rightarrow^1 \parallel \beta L', M \rightarrow^1 \parallel \beta M'}{(\lambda x[L] M) \rightarrow^1 \parallel \beta \{M'/x\} L'}$
Parallel Beta: The Diamond Property

Lemma 10.2

1. $L \xrightarrow{1} \beta L' \Rightarrow L \xrightarrow{1} \|\beta L'$.

2. $L \xrightarrow{1} \|\beta L' \Rightarrow L \xrightarrow{*} \beta L'$.

3. The smallest preorder containing $\xrightarrow{1} \|\beta$ is $\xrightarrow{*} = \xrightarrow{\|\beta} \beta$.

4. If $L \xrightarrow{1} \beta L'$ and $M \xrightarrow{1} \|\beta M'$ then $\{M/x\}L \xrightarrow{1} \|\beta \{M'/x\}L'$.

Proof: By induction on the structure of terms or by induction on the number of steps in any proof. QED

Theorem 10.3 $\xrightarrow{1} \|\beta$ has the diamond property.
Proof of theorem 10.3

Proof: We need to prove for all $L$

$$\iff L \rightarrow_{\beta} M \Rightarrow \exists P : N \rightarrow_{\beta} P \rightarrow_{\beta} M$$

We prove this by induction on the structure of $L$ and a case analysis of the rule applied in definition 10.1.

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 $L \equiv_{a} M$. Choose $P \equiv_{a} N$ to complete the diamond.

Case $L \equiv_{a} N$. Then choose $P \equiv_{a} M$.

Case $M \equiv_{a} N$. Then there is nothing to prove.

In the sequel we assume $N \neq_{a} L \neq_{a} M \neq_{a} N$ 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 $\beta_1Abs$ 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 $N_1 \rightarrow_{\beta} L_1 \rightarrow_{\beta} M_1$. By the induction hypothesis we have

$$\exists P_1 : N_1 \rightarrow_{\beta} P_1 \rightarrow_{\beta} M_1$$

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 $\beta_1App$ is the only rule that must have been applicable in the last step of the proofs of $N \rightarrow_{\beta} L \rightarrow_{\beta} M$. Clearly then there exist $M_1, M_2, N_1, N_2$ such that $N_1 \rightarrow_{\beta} L_1 \rightarrow_{\beta} M_1$ and $N_2 \rightarrow_{\beta} L_2 \rightarrow_{\beta} M_2$. Again by the induction hypothesis, we have

$$\exists P_1 : N_1 \rightarrow_{\beta} P_1 \rightarrow_{\beta} M_1$$

and

$$\exists P_2 : N_2 \rightarrow_{\beta} P_2 \rightarrow_{\beta} M_2$$

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 $\parallel \beta_1 \text{App}$ or $\parallel \beta_1 \text{Abs}$. Of these the sub-case when both $M$ and $N$ were obtained by applying $\parallel \beta_1 \text{App}$ is easy and similar to the previous case. That leaves us with three subcases.

**Sub-case: Both $M$ and $N$ were obtained by applying rule $\parallel \beta_1 \text{Abs}$.

Then we have

$$\{N_2/x\}N_1 \equiv N_1 \xrightarrow{1} L \equiv (\lambda x[L_1] \ L) \rightarrow_\beta^1 M \equiv \{M_2/x\}M_1$$

for some $M_1, M_2, N_1, N_2$ such that

$$N_1 \xrightarrow{1} L_1 \rightarrow_\beta^1 M_1$$

and

$$N_2 \xrightarrow{1} L_2 \rightarrow_\beta^1 M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \rightarrow_\beta^1 P_1 \xrightarrow{1} M_1$$

and

$$\exists P_2 : N_2 \rightarrow_\beta^1 P_2 \xrightarrow{1} M_2$$

and the last part of lemma 10.2 we have

$$\exists P \equiv \{P_2/x\}P_1 : N \rightarrow_\beta^1 P \xrightarrow{1} M$$

completing the proof.

**Sub-case: $M$ was obtained by applying rule $\parallel \beta_1 \text{Abs}$ and $N$ by $\parallel \beta_1 \text{App}$.**

Then we have the form

$$(\lambda x[N_1] \ N_2) \equiv N_1 \xrightarrow{1} L \equiv (\lambda x[L_1] \ L) \rightarrow_\beta^1 M \equiv \{M_2/x\}M_1$$

where again

$$N_1 \xrightarrow{1} L_1 \rightarrow_\beta^1 M_1$$

and

$$N_2 \xrightarrow{1} L_2 \rightarrow_\beta^1 M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \rightarrow_\beta^1 P_1 \xrightarrow{1} M_1$$
and
\[ \exists P_2 : N_2 \xrightarrow{1}_{|\beta} P_2 \xleftarrow{1}_{|\beta} M_2 \]
and finally we have
\[ \exists P \equiv \{ P_2/x \} P_1 : N \xrightarrow{1}_{|\beta} P \xleftarrow{1}_{|\beta} M \]
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

Theorem 10.4 $\rightarrow 1_{|\beta}$ is confluent.

Proof: We need to show that for all $L, M, N$,
\[ N \xrightarrow{1}_{|\beta} L \xrightarrow{*}_{|\beta} M \Rightarrow \exists P : N \xrightarrow{*}_{|\beta} P \xrightarrow{*}_{|\beta} M \]
We prove this by induction on the length of the sequences
\[ L \xrightarrow{1}_{|\beta} M_1 \xrightarrow{1}_{|\beta} M_2 \xrightarrow{1}_{|\beta} \cdots \xrightarrow{1}_{|\beta} M_m \equiv M \]
and
\[ L \xrightarrow{1}_{|\beta} N_1 \xrightarrow{1}_{|\beta} N_2 \xrightarrow{1}_{|\beta} \cdots \xrightarrow{1}_{|\beta} N_n \equiv N \]
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 10.3 to complete the rectangle
\[ L \xrightarrow{1}_{|\beta} M_1 \xrightarrow{1}_{|\beta} M_2 \xrightarrow{1}_{|\beta} \cdots \xrightarrow{1}_{|\beta} M_m \equiv M \]
\[ N_1 \xrightarrow{1}_{|\beta} P_{11} \xrightarrow{1}_{|\beta} P_{12} \xrightarrow{1}_{|\beta} \cdots \xrightarrow{1}_{|\beta} P_{1m} \]
\[ \vdots \]
\[ N_n \xrightarrow{1}_{|\beta} P_{n1} \xrightarrow{1}_{|\beta} P_{n2} \xrightarrow{1}_{|\beta} \cdots \xrightarrow{1}_{|\beta} P_{nm} \equiv P \]
QED
Corollary 10.5 $\xrightarrow{\beta} \text{is confluent.}$

Proof: Since $\xrightarrow{\beta} = \xrightarrow{\beta \parallel}$, it follows from theorem 10.4 that $\xrightarrow{\beta}$ is confluent. QED

Corollary 10.6 If a term reduces to a $\beta$-normal form then the normal form is unique (upto $\equiv_\alpha$).

Proof: If $N_1 \xleftarrow{\beta} L \xrightarrow{\beta} N_2$ and both $N_1, N_2$ are $\beta$-nfs, then by the corollary 10.5 they must both be $\beta$-reducible to a third element $N_3$ which is impossible if both $N_1$ and $N_2$ are $\beta$-nfs. Hence $\beta$-nfs are unique whenever they exist. QED

Corollary 10.7 $\xrightarrow{\beta} \text{is Church-Rosser.}$

Proof: Follows from corollary 10.5 and theorem 11.2. QED

11. Confluence Characterizations
The Church-Rosser Property
Confluence and Church-Rosser

Lemma 11.1 Every confluent relation is also semi-confluent

Theorem 11.2 The following statements are equivalent for any reduction relation $\rightarrow$.

1. $\rightarrow$ is Church-Rosser.
2. $\rightarrow$ is confluent.
Proof of theorem 11.2

Proof: (1 \Rightarrow 2) Assume \rightarrow is Church-Rosser and let

\[ N \leftarrow L \rightarrow * M \]

Clearly then \( N \leftarrow* M \). If \( \rightarrow \) is Church-Rosser then

\[ \exists P : N \rightarrow* P \leftarrow* M \]

which implies that it is confluent.

(2 \Rightarrow 1) Assume \( \rightarrow \) is confluent and let \( L \leftarrow* M \). We proceed by induction on the length of the chain (4).

\[ L \equiv_{\alpha} M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_m \equiv_{\alpha} M \]

**Basis.** \( m = 0 \). This case is trivial since for any \( P, L \rightarrow* P \) iff \( M \rightarrow* P \)

**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.

\[ L \equiv_{\alpha} M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_k \rightarrow / \leftarrow M_{k+1} \equiv_{\alpha} M \]

Case \( M_k \rightarrow M \). Then by the induction hypothesis and semi-confluence we have

\[ L \leftarrow* M_k \leftarrow* \exists Q \leftarrow* M \leftarrow* \exists P \]

which proves the claim.
Case $M_k \leftarrow M$. Then the claim follows from the induction hypothesis and the following diagram

```
L \leftrightarrow^* M_k \leftarrow M
\downarrow \nonumber \downarrow \nonumber \exists P
\exists \nonumber \nonumber \nonumber
```

QED

**Lemma 11.3** If a terminating relation is locally confluent then it is semi-confluent.

**Proof:** Assume $L \rightarrow M$ and $L \rightarrow^* N$. We need to show that there exists $P$ such that $M \rightarrow^* P$ and $N \rightarrow^* P$. We prove this by induction on the length of $L \rightarrow^* N$. If $L \equiv_a N$ then $P \equiv_a M$, otherwise assume $L \rightarrow N_1 \rightarrow \cdots \rightarrow N_n = N$ for some $n > 0$. By the local confluence we have there exists $P_1$ such that $M \rightarrow^* P_1$. By successively applying the induction hypothesis we get terms $P_2, \ldots, P_n$ such that $P_{j-1} \rightarrow^* P_j$ and $N_j \rightarrow^* P_j$ for each $j$, $1 \leq j \leq m$. In effect we complete the following rectangle

```
L \rightarrow N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_n \equiv M
\downarrow \downarrow \downarrow \cdots \downarrow
M \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_n
```

QED

From lemma 11.3 and theorem 11.2 we have the following theorem.

**Theorem 11.4** If a terminating relation is locally confluent then it is confluent.

**Proof:**

$\rightarrow$ on $\Lambda$ 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. $N_1 \xleftarrow{1} L \rightarrow^1 M_1 \Rightarrow \exists P : M_1 \rightarrow^* P \xleftarrow{\ast} N_1$
We need to show for any term \( L \) that
\[
N \leftarrow L \rightarrow^* M \Rightarrow \exists S : M \rightarrow^* S \leftarrow N
\] (6)

Let \( L \) be any term. Consider the graph \( G(L) = (\Gamma(L), \rightarrow^1) \) such that \( \Gamma(L) = \{ M \mid L \rightarrow^* M \} \). Since \( \rightarrow \) is a terminating reduction

**Fact 11.5** 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 \( M_0 \rightarrow^1 M_1 \rightarrow^1 \cdots \rightarrow^1 M_{k-1} \rightarrow^1 M_0 \) which implies there is also an infinite reduction sequence of the form \( L \rightarrow^* M_0 \rightarrow^k M_0 \rightarrow^k \cdots \) which is impossible.

Since there are only a finite number of sub-terms of \( L \) that may be reduced under \( \rightarrow \), for each \( L \) there is a maximum number \( p \geq 0 \), which is the length of the longest reduction sequence.

**Fact 11.6** 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 \( p \).

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 (6) holds.

**Induction Step.** Assume \( L \) is a term whose longest reduction sequence is of length \( p > 0 \). Also assume \( N \leftarrow L \rightarrow^* M \) i.e. \( \exists m, n \geq 0 : N \leftarrow^m L \rightarrow^n M \).

*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

$$N \leftarrow N_1 \leftarrow L \rightarrow^1 M_1 \rightarrow^* M$$

(7)

See figure (1). By (5), $\exists P : M_1 \rightarrow^* P \leftarrow^* N_1$. Clearly $M_1, N_1, P \in \Gamma(L) - \{L\}$. Hence by fact 11.6, $G(M_1), G(N_1)$ 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

$$P \leftarrow^* M_1 \rightarrow^* M \Rightarrow \exists Q : M \rightarrow^* Q \leftarrow^* P$$

(8)

and

$$N \leftarrow^* N_1 \rightarrow^* P \Rightarrow \exists R : P \rightarrow^* R \leftarrow^* N$$

(9)
But by (8) and (9) and the induction hypothesis we have

\[ R \leftarrow P \rightarrow^* Q \Rightarrow \exists S : Q \rightarrow^* S \leftarrow R \]  

(10)

Combining (10) with (7), (8) and (9) we get

\[ N \leftarrow L \rightarrow^* M \Rightarrow \exists S : M \rightarrow^* S \leftarrow N \]  

(11)

QED

**Theorem 11.7** If a terminating relation is locally confluent then it is Church-Rosser.

**Proof:** Follows from theorem 11.4 and theorem 11.2  

QED
12. An Applied Lambda-Calculus
12.1. FL with recursion

An Applied Lambda-Calculus
A Simple Language of Terms: FL

Let $X$ be an infinite collection of variables (names). Consider the language (actually a collection of abstract syntax trees) of terms $T_{\Sigma}(X)$ defined by the following constructors (along with their intended meanings).

<table>
<thead>
<tr>
<th>Construct</th>
<th>Arity</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>0</td>
<td>The number 0</td>
</tr>
<tr>
<td>$T$</td>
<td>0</td>
<td>The truth value true</td>
</tr>
<tr>
<td>$F$</td>
<td>0</td>
<td>The truth value false</td>
</tr>
<tr>
<td>$P$</td>
<td>1</td>
<td>The predecessor function on numbers</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>The successor function on numbers</td>
</tr>
<tr>
<td>ITE</td>
<td>3</td>
<td>The if-then-else construct (on numbers and truth values)</td>
</tr>
<tr>
<td>IZ</td>
<td>1</td>
<td>The is-zero predicate on numbers</td>
</tr>
<tr>
<td>GTZ</td>
<td>1</td>
<td>The greater-than-zero predicate on numbers</td>
</tr>
</tbody>
</table>
FL: Language, Datatype of Instruction Set?

The set of terms $T_\Sigma(X)$ may be alternatively defined by the BNF:

$$t ::= x \in X \mid Z \mid (P \ t) \mid (S \ t) \mid T \mid F \mid (\text{ITE} \langle t, t_1, t_0 \rangle) \mid (IZ \ t) \mid (GTZ \ t)$$

- 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
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 $S(t)$, $P(t)$, $IZ(t)$ and $GTZ(t)$
- $ITE(t, t_1, t_0)$ 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.
FL: 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 $\Lambda_\Sigma(X)$ over this set of terms as follows:

$$L, M ::= t \in T_\Sigma(X) \mid \lambda x[L] \mid (L M)$$
The Normal forms for Integers

We need reduction rules to simplify (non-recursive) expressions.

Zero. \( Z \) is the unique representation of the number 0 and every integer expression that is equal to 0 must be reducible to \( Z \).

Positive integers. Each positive integer \( k \) is uniquely represented by the expression \( S^k(Z) \) where the super-script \( k \) denotes a \( k \)-fold application of \( S \).

Negative integers. Each negative integer \( -k \) is uniquely represented by the expression \( P^k(Z) \) where the super-script \( k \) denotes a \( k \)-fold application of \( P \).

\( \delta \)-rules

\[
\begin{align*}
(P \, (S \, x)) & \rightarrow_{\delta} x \quad (12) \\
(S \, (P \, x)) & \rightarrow_{\delta} x \quad (13)
\end{align*}
\]
Reduction Rules for Boolean Expressions

Pure Boolean Reductions. The constructs $T$ and $F$ are the normal forms for boolean values.

$$\text{(ITE } \langle T, x, y \rangle \text{)} \xrightarrow{\delta} x \quad \text{(14)}$$
$$\text{(ITE } \langle F, x, y \rangle \text{)} \xrightarrow{\delta} y \quad \text{(15)}$$

Testing for zero.

$$\text{(IZ } Z \text{)} \xrightarrow{\delta} T \quad \text{(16)}$$
$$\text{(IZ } S^k Z \text{)} \xrightarrow{\delta} F, k > 0 \quad \text{(17)}$$
$$\text{(IZ } P^k Z \text{)} \xrightarrow{\delta} F, k > 0 \quad \text{(18)}$$
Testing for Positivity

\[
(GTZ \ Z) \xrightarrow{\delta} F \quad \text{(19)}
\]
\[
(GTZ (S^k Z)) \xrightarrow{\delta} T, k > 0 \quad \text{(20)}
\]
\[
(GTZ (P^k Z)) \xrightarrow{\delta} F, k > 0 \quad \text{(21)}
\]
Other Non-recursive Operators

We may “program” the other boolean operations as follows:

\[
\text{NOT} \overset{df}{=} \lambda x[\text{ITE} \langle x, F, T \rangle] \\
\text{AND} \overset{df}{=} \lambda \langle x, y \rangle[\text{ITE} \langle x, y, F \rangle] \\
\text{OR} \overset{df}{=} \lambda \langle x, y \rangle[\text{ITE} \langle x, T, y \rangle]
\]

We may also “program” the other integer comparison operations as follows:

\[
\text{GEZ} \overset{df}{=} \lambda x[\text{OR} \langle \langle \text{IZ} x \rangle, (\text{GTZ} x) \rangle] \\
\text{LTZ} \overset{df}{=} \lambda x[\text{NOT} (\text{GEZ} x)] \\
\text{LEZ} \overset{df}{=} \lambda x[\text{OR} \langle \langle \text{IZ} x \rangle, (\text{LTZ} x) \rangle]
\]
Recursion in the Untyped Lambda-calculus

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 12.1** A combinator $Y$ is called a fixed-point combinator if for every lambda term $L$, $\beta(Y L) = (L (Y L))$

Curry’s $\text{Y}_C$ combinator

$$Y_C \overset{df}{=} \lambda f[(C C)]$$

where

$$C \overset{df}{=} \lambda x[(f (x x))]$$

Turing’s $\text{Y}_T$ combinator

$$Y_T \overset{df}{=} (T T)$$

where

$$T \overset{df}{=} \lambda y x[(x (y y x))]$$
FL: Adding Recursion

But the various $\gamma$ combinators unfortunately will not satisfy any typing rules that we may define for the language. Instead it is more convenient to use the fixed-point property and define a new constructor with a $\delta$-rule which satisfies the fixed-point property (definition 12.1).

We extend the language FL with a new constructor

$$L ::= \ldots | (\text{REC } L)$$

and add the fixed point property as a $\delta$-rule

$$\text{(REC } L) \rightarrow_\delta (L \text{ (REC } L))$$

(22)
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 inductively.

\[
x + y = \begin{cases} 
y & \text{if } x = 0 \\
(x - 1) + (y + 1) & \text{if } x > 0 \\
(x + 1) + (y - 1) & \text{if } x < 0
\end{cases}
\]  

(23)
Using the constructors of FL we require that any (curried) definition of addition on numbers should be a solution to the following equation in FL for all (integer) expression values of \(x\) and \(y\).

\[
(plusc x y) =_{\beta\delta} \text{ITE} \langle (IZ \ x), \ y, \ \text{ITE} \langle (GTZ \ x), \ (plusc (P \ x) \ (S \ y)), \ (plusc (S \ x) \ (P \ y))\rangle \rangle
\]  

(24)

Equation (24) may be rewritten using abstraction as follows:

\[
plusc =_{\beta\delta} \lambda x \lbrack \lambda y \lbrack \text{ITE} \langle (IZ \ x), \ y, \ \text{ITE} \langle (GTZ \ x), \ (plusc (P \ x) \ (S \ y)), \ (plusc (S \ x) \ (P \ y))\rangle \rangle \rbrack \rbrack
\]  

(25)

We may think of equation (25) as an equation to be solved in the unknown variable \(plusc\).

Consider the (applied) \(\lambda\)-term obtained from the right-hand-side of equation (25) by simply abstracting the unknown \(plusc\).

\[
addc \overset{df}{=} \lambda f \lbrack \lambda x \ y \lbrack \text{ITE} \langle (IZ \ x), \ y, \ \text{ITE} \langle (GTZ \ x), \ (f (P \ x) \ (S \ y)), \ (f (S \ x) \ (P \ y))\rangle \rangle \rbrack \rbrack
\]  

(26)

Claim 12.2

\[
(\text{REC } addc) \rightarrow_{\delta} (addc (\text{REC } addc))
\]  

(27)

and hence

\[
(\text{REC } addc) =_{\beta\delta} (addc (\text{REC } addc))
\]  

(28)

Claim 12.3 \((\text{REC } addc)\) satisfies exactly the equation (25). That is

\[
((\text{REC } addc) \ x \ y) =_{\beta\delta} \text{ITE} \langle (IZ \ x), \ y, \ \text{ITE} \langle (GTZ \ x), \ ((\text{REC } addc) (P \ x) \ (S \ y)), \ ((\text{REC } addc) (S \ x) \ (P \ y))\rangle \rangle
\]  

(29)

Hence we may regard \((\text{REC } addc)\) where \(addc\) is defined by right-hand-side of definition (26) as the required solution to the equation (24) in which \(plusc\) is an unknown.
The abstraction shown in (26) and the claims (12.2) and (12.3) simply go to show that $M \equiv_\alpha \lambda f\{f/z\}L$ is a solution to the equation $z =_\beta L$, 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 12.1

1. Prove that the relation $\rightarrow_8$ 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. Prove that $Y_C$ and $Y_T$ are both fixed-point combinators.

4. 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.

5. The equation (23) is defined conditionally. However the following is equally valid for all integer values $x$ and $y$.

   \[ x + y = (x - 1) + (y + 1) \]  

   (a) Follow the steps used in the construction of $addc$ to define a new applied $addc'$ that instead uses equation (30).
   (b) Is $REC addc' =_\beta (addc' (REC addc'))$?
   (c) Is $addc =_\beta addc'$?
   (d) Is $REC addc =_\beta (REC addc')$?

6. The function $addc$ was defined in curried form. Use the pairing function in the untyped $\lambda$-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.
12.2. FL with type rules

Typing FL expressions

We have already seen that the simple language FL has

- A types is an important *attribute* of any variable, constant or expression.
- 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 FL

Besides the need for type-checking rules on $T_\Sigma(X)$ to prevent illegal constructor operations,

• rules are necessary to ensure that $\lambda$-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 Simple Language of Types

Consider the following language of types (in fully parenthesized form) defined over an infinite collection $'a \in TV$ of type variables. We also have two type constants $\texttt{int}$ and $\texttt{bool}$.

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid 'a \in TV \mid (\sigma \ast \tau) \mid (\sigma \rightarrow \tau)$$

Notes.

- $\texttt{int}$ and $\texttt{bool}$ are type constants.
- $\ast$ is the product operation on types and $\rightarrow$ is function operator on types.
- We require $\ast$ because of the possibility of defining functions of various kinds of arities in $\Lambda_{\Sigma}(X)$.
- **Precedence.** We assume $\ast$ has a higher precedence than $\rightarrow$.
- **Associativity.** $\rightarrow$ is right associative whereas $\ast$ is left associative.
- In any type expression $\tau$, $TVar(\tau)$ is the set of type variables
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 Assignment: Infrastructure

- Assume $\Gamma$ is the environment (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 FL 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 FL.

- The expressions of FL could have free variables. The type of the expression would then depend on the types assigned to the free variables.
Constraint Typing Relation

**Definition 12.4** For each term \( L \in \Lambda_\Sigma(X) \) the constraint typing relation is of the form

\[
\Gamma \vdash L : \tau \; \triangleright_T \; C
\]

where

- \( \Gamma \) is called the context and defines the stack of assumptions that may be needed to assign a type (expression) to the (sub-)expression \( L \).
- \( \tau \) 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
Typing axioms: Basic

The following axioms may be applied during the scanning and parsing phases of the compiler to assign types to the individual tokens.

\[
\begin{align*}
\text{Z} & \quad \Gamma \vdash Z : \text{int} \quad \triangleright \emptyset \quad \emptyset \\
\text{T} & \quad \Gamma \vdash T : \text{bool} \quad \triangleright \emptyset \quad \emptyset \\
\text{F} & \quad \Gamma \vdash F : \text{bool} \quad \triangleright \emptyset \quad \emptyset \\
\text{S} & \quad \Gamma \vdash S : \text{int} \rightarrow \text{int} \quad \triangleright \emptyset \quad \emptyset \\
\text{P} & \quad \Gamma \vdash P : \text{int} \rightarrow \text{int} \quad \triangleright \emptyset \quad \emptyset \\
\text{IZ} & \quad \Gamma \vdash IZ : \text{int} \rightarrow \text{bool} \quad \triangleright \emptyset \quad \emptyset \\
\text{GTZ} & \quad \Gamma \vdash GTZ : \text{int} \rightarrow \text{bool} \quad \triangleright \emptyset \quad \emptyset \\
\text{ITEI} & \quad \Gamma \vdash ITE : \text{bool} \times \text{int} \times \text{int} \rightarrow \text{int} \quad \triangleright \emptyset \quad \emptyset \\
\text{ITEB} & \quad \Gamma \vdash ITE : \text{bool} \times \text{bool} \times \text{bool} \rightarrow \text{bool} \quad \triangleright \emptyset \quad \emptyset
\end{align*}
\]

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 for FL: 2

\[
\begin{align*}
\text{Var} & \quad \Gamma \vdash x : \Gamma(x) \triangleright_{\emptyset} \emptyset \\
\text{Abs} & \quad \frac{\Gamma, x : \sigma \vdash L : \tau \triangleright_T C}{\Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \triangleright_T C} \\
\text{App} & \quad \frac{\Gamma \vdash L : \sigma \triangleright_{T_1} C_1 \quad \Gamma \vdash M : \tau \triangleright_{T_2} C_2}{\Gamma \vdash (L \ M) : \text{'}a \triangleright_{T'} C'}
\end{align*}
\]

(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.** \( \text{'a} \notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2) \)
- \( T' = T_1 \cup T_2 \cup \{\text{'a}\} \)
- \( C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow \text{'a}\} \)
Example 12.5 Consider the following simple combinator \( C = \lambda x[\lambda y[\lambda z[(x (y z))]]] \) which defines the function composition operator. Since there are three bound variables \( x, y \) and \( z \) we begin with an initial assumption \( \Gamma = x : 'a, y : 'b, z : 'c \) which assign arbitrary types to the bound variables, represented by the type variables \( 'a, 'b \) and \( 'c \) respectively. Our inference for the type of \( C \) then proceeds as follows. Note, that since \( C \) 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.

1. \( x : 'a, y : 'b, z : 'c \vdash x : 'a \triangleright \emptyset \) (Var)
2. \( x : 'a, y : 'b, z : 'c \vdash y : 'b \triangleright \emptyset \) (Var)
3. \( x : 'a, y : 'b, z : 'c \vdash z : 'c \triangleright \emptyset \) (Var)
4. \( x : 'a, y : 'b, z : 'c \vdash (y z) : 'd \triangleright (d, e) \{ 'b = 'c \rightarrow 'd \} \) (App)
5. \( x : 'a, y : 'b, z : 'c \vdash (x (y z)) : 'e \triangleright (d, e) \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) (App)
6. \( x : 'a, y : 'b \vdash \lambda z[(x (y z))] : 'c \rightarrow 'e \triangleright (d, e) \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) (Abs)
7. \( x : 'a \vdash \lambda x[\lambda y[\lambda z[(x (y z))]]] : 'b \rightarrow 'c \rightarrow 'e \triangleright (d, e) \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) (Abs)
8. \( \vdash \lambda x[\lambda y[\lambda z[(x (y z))]]] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e \triangleright (d, e) \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) (Abs)

Hence \( \lambda x[\lambda y[\lambda z[(x (y z))]]] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e \) subject to the constraints given by \( \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) which yields \( \lambda x[\lambda y[\lambda z[(x (y z))]]] : (d \rightarrow 'e) \rightarrow ('c \rightarrow 'd) \rightarrow 'c \rightarrow 'e \)
Principal Type Schemes

Definition 12.6 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 $\tau$ such that $S(\tau) = \sigma$.

- The rules yield a principal type scheme for each well-typed applied $\lambda$-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 12.2

1. The language has several constructors which behave like functions. Derive the following rules for terms in $T_\Sigma(X)$ from the basic typing axioms and the rule $\text{App}$.

\[
\frac{\Gamma \vdash t : \tau \triangleright_T C}{\Gamma \vdash (S t) : \text{int} \triangleright_T C \cup \{\tau = \text{int}\}}
\]

\[
\frac{\Gamma \vdash t_1 : \tau \triangleright_{T_1} C_1 \quad \Gamma \vdash t_0 : \nu \triangleright_{T_0} C_0}{\Gamma \vdash \text{ITE} \langle t, t_1, t_0 \rangle : \tau \triangleright_{T'} C'}
\]

where $T' = T \cup T_1 \cup T_0$ and $C' = C \cup C_1 \cup C_0 \cup \{\sigma = \text{bool}, \tau = \nu\}$

2. Use the rules to define the type of $S$?
3. How would you define a type assignment for the recursive function \texttt{addc} defined by equation (26).

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) \((K S)\)
   
   (b) \(((K S) \omega)\)
   
   (c) \(((S K) K) \omega\)
   
   (d) \((\text{ITE} \langle (\text{IZ} x), T, (K x) \rangle)\)
13. An Imperative Language

An Imperative Language
The Concept of State

• 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.

• We define the store to be a (partial) function from $Loc$ to values. $\sigma : Loc \rightarrow (\text{int} \cup \text{bool})$. $\text{Stores} = \{\sigma \mid \sigma : Loc \rightarrow (\text{int} \cup \text{bool})\}$ is the set of possible stores.

• Each variable in an imperative program is assigned a location.

• The environment is an association $\gamma$ of variable (names) to locations i.e. $\gamma : X \rightarrow Loc$.

• The (dynamic) state of a program is defined by the pair $(\gamma, \sigma)$. 

13.1. The Operational Semantics of Expressions
• l-values. $\gamma(x) = i : \text{bool}, \gamma(y) = j : \text{int}, \gamma(z) = k : \text{int}$

• r-values. $\sigma(i) = T : \text{bool}, \sigma(j) = 132456 : \text{int}, \sigma(k) = -87567 : \text{int}$
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 `i`.

• have an explicit unary dereferencing operator `!` to read the value contained in the location referenced by `x`, i.e. `!x = σ(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 performed on them!
l-values and r-values

• \( l \) is the l-value of \( w \) i.e. \( \gamma(w) = l \in \text{Loc} \)
• \( m \) is the r-value of \( w \) i.e. \( \sigma(\gamma(w)) = !w = m \in \text{Loc} \)
• \( m \) is also an l-value since \( !w : \text{int ref} \)
• \( !(\!w) = 78663 : \text{int} \) is the r-value of \( !w \)
13.1.1. 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
Modern impure functional languages (which have strong-typing facilities) usually clearly distinguish between locations and values as different types. Hence every imperative variable represents only an l-value. Its r-value is obtained by applying a dereferencing operation (the prefix operation \texttt{!}). 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 = ref 1 : int ref
- !(!z);
val it = 1 : int
- !(!x);
val it = 1 : int
-
Operational Semantics of Expressions

• Consider the language of terms defined FL. Instead of the $\delta$-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 bool = \{T, F\}.

• We assume that every (sub-)expression in the language has been typed with a unique type attribute.

• We define an expression evaluation relation $\rightarrow_e$ such that

$$\rightarrow_e \subseteq (\text{Stores} \times T_\Sigma(X)) \times (\text{Stores} \times (T_\Sigma(X) \cup \text{int} \cup \text{bool}))$$

in a given environment $\gamma$. 
13.2. The Operational Semantics of Commands

The WHILE language

- We initially define a simple language of *commands*.
- The expressions of the language are those of any term algebra $T_\Sigma(X)$.
- We simply assume there is a well-defined relation $\rightarrow_e$ for evaluating expressions in
- We define $\rightarrow_e$ for FL later.
- State changes are modelled by *assignment commands*
- Given a store $\sigma$, a variable $x$ and a value $a$, $\sigma' = [\gamma(x) \mapsto a] \sigma$ is the new store that is identical to $\sigma$ except at the location $\gamma(x)$ which now contains the value $a$ i.e. $\sigma'(\ell) = \begin{cases} \sigma(\ell) & \text{if } \ell \neq \gamma(x) \\ a & \text{otherwise} \end{cases}$
**Aliases**

**Definition 13.1** 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

\[ c_0, c_1, c \ ::= \begin{align*}
\text{skip} & \quad \text{Skip} \\
\text{x := e} & \quad \text{Assgn} \\
\{c_0\} & \quad \text{Block} \\
c_0; c_1 & \quad \text{Seq} \\
\text{if e then c}_1 \ \text{else c}_0 & \quad \text{Cond} \\
\text{while e do c} & \quad \text{While}
\end{align*} \]

where \( e \) is either an integer or boolean expression in the language FL with operational semantics as given before.
Operational Semantics: Basic Commands

**Skip**

\[ \gamma \vdash \langle \sigma, \text{skip} \rangle \longrightarrow^1_\cdot \sigma \]

**Assgn0**

\[ \gamma \vdash \langle \sigma, x := m \rangle \longrightarrow^1_\cdot [\gamma(x) \mapsto m] \sigma \]

**Assgn1**

\[ \gamma \vdash \langle \sigma, e \rangle \longrightarrow e \langle \sigma, e' \rangle \\
\gamma \vdash \langle \sigma, x := e \rangle \longrightarrow^1_\cdot \langle \sigma, x := e' \rangle \]

**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 actually changes state (**Assgn0**).
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.

\[
\begin{align*}
\text{Block0} \quad & \frac{\gamma \vdash \langle \sigma, c \rangle}{\gamma \vdash \langle \sigma, \{c\} \rangle} \quad \rightarrow^1 \quad \frac{\sigma'}{\sigma'} \\
\text{Block1} \quad & \frac{\gamma \vdash \langle \sigma, c \rangle}{\gamma \vdash \langle \sigma, \{c\} \rangle} \quad \rightarrow^1 \quad \frac{\langle \sigma', c' \rangle}{\langle \sigma', \{c'\} \rangle}
\end{align*}
\]
Operational Semantics: Sequencing

<table>
<thead>
<tr>
<th>Seq0</th>
<th>$\gamma \vdash \langle \sigma, c_0 \rangle \quad \frac{\rightarrow_1^c \sigma'}{\gamma \vdash \langle \sigma, c_0; c_1 \rangle}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\gamma \vdash \langle \sigma, c_0 \rangle \quad \frac{\rightarrow_1^c \langle \sigma', c_1 \rangle}{\gamma \vdash \langle \sigma, c_0; c_1 \rangle}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq1</th>
<th>$\gamma \vdash \langle \sigma, c_0 \rangle \quad \frac{\rightarrow_1^c \langle \sigma', c'_0 \rangle}{\gamma \vdash \langle \sigma, c_0; c_1 \rangle}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\gamma \vdash \langle \sigma, c_0; c_1 \rangle \quad \frac{\rightarrow_1^c \langle \sigma', c'_0; c_1 \rangle}{\gamma \vdash \langle \sigma, c_0; c_1 \rangle}$</td>
</tr>
</tbody>
</table>

Notice that sequencing is precisely the composition of relations. If the relation $\rightarrow_1^c$ 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

\[ \text{Cond0} \quad \gamma \vdash \langle \sigma, \text{if } F \text{ then } c_1 \text{ else } c_0 \rangle \rightarrow_c^{1} \langle \sigma, c_0 \rangle \]

\[ \text{Cond1} \quad \gamma \vdash \langle \sigma, \text{if } T \text{ then } c_1 \text{ else } c_0 \rangle \rightarrow_c^{1} \langle \sigma, c_1 \rangle \]

\[ \text{Cond2} \quad \gamma \vdash \langle \sigma, e \rangle \rightarrow_e \langle \sigma, e' \rangle \]

\[ \gamma \vdash \langle \sigma, \text{if } e \text{ then } c_1 \text{ else } c_0 \rangle \rightarrow_c^{1} \langle \sigma, \text{if } e' \text{ then } c_1 \text{ else } c_0 \rangle \]
Operational Semantics: While loop

We use the fact that the `while e do c` is really a form of recursion – actually it is a form of “tail recursion”. Hence the execution behaviour of `while e do c` is exactly that of

\[
\text{if } e \text{ then } \{c; \text{while } e \text{ do } c\} \text{ else skip}
\]  

(31)

The following rules may be derived from (31) using the rules for conditional, sequencing and skip (though the number of steps may not exactly correspond).

\[
\begin{align*}
\text{While0} & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow_e \langle \sigma, F \rangle}{\gamma \vdash \langle \sigma, \text{while } e \text{ do } c \rangle \rightarrow \frac{1}{c} \sigma} \\
\text{While1} & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow_e \langle \sigma, T \rangle}{\gamma \vdash \langle \sigma, \text{while } e \text{ do } c \rangle \rightarrow \frac{1}{c} \langle \sigma, c; \text{while } e \text{ do } c \rangle} \\
\text{While2} & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow_e \langle \sigma, e' \rangle}{\gamma \vdash \langle \sigma, \text{while } e \text{ do } c \rangle \rightarrow \frac{1}{c} \langle \sigma, \text{if } e' \text{ then } \{c; \text{while } e \text{ do } c\} \text{ else skip} \rangle}
\end{align*}
\]
Operational Semantics for FL
Evaluating FL on a machine

• We previously treated FL as simply a data-type and gave $\delta$-rules.
• Here we define a deterministic evaluation mechanism $\rightarrow_e$ 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.
Operational Semantics: Constants and Variables

Let $\sigma \in \text{States}$ be any state.

\[
\begin{align*}
T & \quad \gamma \vdash \langle \sigma, T \rangle \rightarrow_e \langle \sigma, T \rangle \\
F & \quad \gamma \vdash \langle \sigma, F \rangle \rightarrow_e \langle \sigma, F \rangle \\
Z & \quad \gamma \vdash \langle \sigma, Z \rangle \rightarrow_e \langle \sigma, 0 \rangle \\
x & \quad \gamma \vdash \langle \sigma, x \rangle \rightarrow_e \langle \sigma, \sigma(\gamma(x)) \rangle
\end{align*}
\]
Operational Semantics: Integer Expressions

\[
\begin{align*}
P & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, m \rangle}{\gamma \vdash \langle \sigma, (P e) \rangle \rightarrow e \langle \sigma, m - 1 \rangle} \quad (e, m : \text{int}) \\
S & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, m \rangle}{\gamma \vdash \langle \sigma, (P e) \rangle \rightarrow e \langle \sigma, m + 1 \rangle} \quad (e, m : \text{int}) \\
IZ0 & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, m \rangle}{\gamma \vdash \langle \sigma, (IZ e) \rangle \rightarrow e \langle \sigma, F \rangle} \quad (e, m : \text{int}, m <> 0) \\
IZ1 & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, 0 \rangle}{\gamma \vdash \langle \sigma, (IZ e) \rangle \rightarrow e \langle \sigma, T \rangle} \quad (e : \text{int}) \\
GTZ0 & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, m \rangle}{\gamma \vdash \langle \sigma, (GTZ e) \rangle \rightarrow e \langle \sigma, F \rangle} \quad (e, m : \text{int}, m <= 0) \\
GTZ1 & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, m \rangle}{\gamma \vdash \langle \sigma, (GTZ e) \rangle \rightarrow e \langle \sigma, T \rangle} \quad (e, m : \text{int}, m > 0)
\end{align*}
\]
Operational Semantics: Conditional Expressions

\[
\begin{align*}
\text{ITEI0} & : \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, F \rangle}{\gamma \vdash \langle \sigma, (\text{ITE} \langle e, e_1, e_0 \rangle) \rangle \rightarrow e \langle \sigma, e_0 \rangle} \quad (e, e_0 : \text{int}) \\
\text{ITEI1} & : \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, T \rangle}{\gamma \vdash \langle \sigma, (\text{ITE} \langle e, e_1, e_0 \rangle) \rangle \rightarrow e \langle \sigma, e_1 \rangle} \quad (e, e_0 : \text{int}) \\
\text{ITEB0} & : \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, F \rangle}{\gamma \vdash \langle \sigma, (\text{ITE} \langle e, e_1, e_0 \rangle) \rangle \rightarrow e \langle \sigma, e_0 \rangle} \quad (e, e_0 : \text{bool}) \\
\text{ITEB1} & : \quad \frac{\gamma \vdash \langle \sigma, e \rangle \rightarrow e \langle \sigma, T \rangle}{\gamma \vdash \langle \sigma, (\text{ITE} \langle e, e_1, e_0 \rangle) \rangle \rightarrow e \langle \sigma, e_1 \rangle} \quad (e, e_0 : \text{bool})
\end{align*}
\]
Local Declarations

We introduce declarations through a new syntactic category $Decls$ defined as follows:

$$d_1, d_2, d ::= \text{int } x \mid \text{bool } y \mid d_1; d_2$$

$$c ::= \cdots \mid \{d; c\}$$

- 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 \( \bot \) to denote the undefined.
- We use \( z \neq \bot \) to denote that \( z \) is well-defined.

\[
\begin{align*}
\text{Assgn0'} & \quad \frac{\gamma \vdash \langle \sigma, x \rangle \xrightarrow{e} \langle \sigma, \sigma(\gamma(x)) \rangle}{x'} (\sigma(\gamma(x)) \neq \bot) \\
\text{Assgn1'} & \quad \frac{\gamma \vdash \langle \sigma, e \rangle \xrightarrow{1_c} \langle \sigma, e' \rangle}{\gamma \vdash \langle \sigma, x := e \rangle \xrightarrow{1_c} \langle \sigma, x := e' \rangle} (\gamma(x) \neq \bot)
\end{align*}
\]
Declarations: Little Environments

The effect of a declaration is to create a little environment which is pushed onto the existing environment. The transition relation

\[ \rightarrow_d \subseteq ((\text{Env} \times \text{Stores} \times \text{Decls}) \times (\text{Env} \times \text{Stores})) \]

\[ \text{int} - x \quad \gamma \vdash \langle \sigma, \text{int} \ x \rangle \rightarrow_d \langle [x \mapsto l], [l \mapsto \bot] \sigma \rangle \quad (l \notin \text{Range}(\gamma)) \]

\[ \text{bool} - x \quad \gamma \vdash \langle \sigma, \text{bool} \ x \rangle \rightarrow_d \langle [x \mapsto l], [l \mapsto \bot] \sigma \rangle \quad (l \notin \text{Range}(\gamma)) \]
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 “[...]” in (fully-bracketed) $\lambda$-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
Disjoint Scopes

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

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

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

The diagram illustrates the overlapping scopes of functions `fun1` and `fun2`, with the `x` value being defined in the outer scope of `fun1`. The `y` value is also defined within the scope of `fun1`, creating an overlap that may lead to unintended behavior or errors in functional programming.
Spanning

```
let
  val x = 10;
  fun fun1 y =
    ...
    ...
  fun fun2 z =
    ...
    ...
  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.
Names & References

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

Back to scope names
Names & References

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

fun1 x

val z = 5;

* z

Back to scope names
Names & References

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

Back to scope names
fun1 x
val x = 10;
fun fun1  y =
let
val x = 15
in
x + y
end
val z = 5;
* z
let
val x = 10; val z = 5;
fun fun1 y = 
let
val x = 15
in
x + y
end
val z = 5;

* z

Back to scope names
fun1 x
val x = 10;
fun fun1  y =
let
val x = 15
in
x + y
end
val z = 5;
* z

Back to scope names
Names & References

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

Back to scope names
Names & References

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

Back to scope names
let

val \textcolor{red}{x} = 10; \textcolor{red}{\texttt{val}} \ x = x - 5;

fun fun1

\texttt{y} =

\begin{align*}
\texttt{let} & \quad \ldots \\
\texttt{in} & \quad \ldots \\
\texttt{end}
\end{align*}

\begin{align*}
\texttt{fun} & \quad \texttt{fun2} \quad \texttt{z} = \\
\texttt{let} & \quad \ldots \\
\texttt{in} & \quad \ldots \\
\texttt{end}
\end{align*}

fun1 (fun2 \textcolor{red}{x})

val \textcolor{red}{x} = x - 5;
in

\begin{align*}
\texttt{let} & \quad \ldots \\
\texttt{in} & \quad \ldots \\
\texttt{end}
\end{align*}

Back to scope names
Names & References

let
val \(x = 10\); val \(x = x - 5\);

fun fun1
  \(y = \ldots\)
  in
  \(\ldots\)
  end

fun fun2
  \(z = \ldots\)
  in
  \(\ldots\)
  end

fun1 (fun2 \(x\))

val \(x = x - 5\);

in
end

Back to scope names
Definition of Names

Definitions are of the form

\[ \text{qualifier name} \ldots = \text{body} \]

- \text{val name} =
- \text{fun name ( argnames )} =
- \text{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
```
• as the *body* of a *local*-declaration

```
local definitions
in  definition
end
```
Example 13.2 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.

```ml
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)
```

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

<table>
<thead>
<tr>
<th>local</th>
<th>fun fun1  y = ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fun fun2  z = ...</td>
</tr>
<tr>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>fun fun3  x = ...</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

```ocaml
local
fun fun1  y = ...
fun fun2  z = ...
in
fun fun3  x = ...
end
```

Diagram:

- `fun fun1` innermost
- `fun fun2` second level
- `fun fun3` outermost
Execution in the Modified Environment

Once a declaration has been processed a new scope $\gamma'$ is created in which the new variables are available for use in addition to everything else that was previously present in the environment $\gamma$ (unless it has been “hidden” by the use of the same name in the new scope). $\gamma'$ is pushed onto $\gamma$ to create a new environment $\gamma[\gamma']$. For any variable $x$,

$$\gamma[\gamma'](x) = \begin{cases} 
\gamma'(x) & \text{if } x \in \text{Dom}(\gamma') \\
\gamma(x) & \text{if } x \in \text{Dom}(\gamma) - \text{Dom}(\gamma') \\
\bot & \text{otherwise}
\end{cases}$$

**D − Seq**

$$
\gamma \vdash \langle \sigma, d_1 \rangle \rightarrow_d \langle \gamma_1, \sigma_1 \rangle \\
\gamma[\gamma_1] \vdash \langle \sigma_1, d_2 \rangle \rightarrow_d \langle \gamma_2, \sigma_2 \rangle \\
\gamma \vdash \langle \sigma, d_1; d_2 \rangle \rightarrow_d \langle \gamma_1[\gamma_2], \sigma_2 \rangle
$$
Semantics of Anonymous Blocks

\[
\begin{align*}
\text{Block:} & \quad \gamma \vdash \langle \sigma, d \rangle \longrightarrow^*_d \langle \gamma', \sigma' \rangle \\
\quad & \quad \quad \gamma[\gamma'] \vdash \langle \sigma', c \rangle \longrightarrow^*_c \sigma'' \\
\quad & \quad \quad \gamma \vdash \langle \sigma, \{d; c\} \rangle \longrightarrow_c \sigma'' \upharpoonright \text{Dom}(\sigma)
\end{align*}
\]

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 \( \gamma' \) 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 \( \text{Dom}(\sigma) \), 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.

\[
d_1, d_2, d ::= \cdots \mid \text{sub } P = c \\
c ::= \cdots \mid P
\]

- 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 $\sigma$ to another store $\sigma'$.
- 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.

\[
\begin{align*}
Proc_0 &= Stores \rightarrow Stores \\
Env &= \{ \gamma \mid \gamma : X \rightarrow (Loc + Proc_0)\}
\end{align*}
\]
Semantics of Parameterless Subroutines

Each procedure declaration \( \text{sub } P = c \) modifies the environment \( \gamma \) by associating the procedure name \( P \) with an entity called a procedure closure \( \text{proc}_0(c, \gamma) \), which represents the body of the procedure and the environment in which it is to be executed.

\[
\text{DSub0} \quad \gamma \vdash \langle \sigma, \text{sub } P = c \rangle \rightarrow_d \langle [P \mapsto \text{proc}_0(c, \gamma)]\gamma, \sigma \rangle
\]

\[
\text{CSub0} \quad \gamma_1 \vdash \langle \sigma, c \rangle \rightarrow_c^* \sigma' \\
\gamma \vdash \langle \sigma, P \rangle \rightarrow_c \langle [P \mapsto \text{proc}_0(c, \gamma)]\gamma, \sigma' \rangle \quad (\gamma(P) = \text{proc}_0(c, \gamma_1))
\]

If \( P \) is recursive then we modify the last rule to

\[
\text{CrecSub0} \quad \gamma_2 \vdash \langle \sigma, c \rangle \rightarrow_c^* \sigma' \\
\gamma \vdash \langle \sigma, P \rangle \rightarrow_c \langle [P \mapsto \text{proc}_0(c, \gamma)]\gamma, \sigma' \rangle \quad (\gamma(P) = \text{proc}_0(c, \gamma_1))
\]

where \( \gamma_2 = [P \mapsto \gamma(P)]\gamma_1 \).

A generalization to mutual recursion with many such procedures is, in principle easy, though notationally tedious.
Subroutines with Value Parameters

We consider the case of only a single parameter for simplicity.

\[
d_1, d_2, d ::= \cdots \mid \text{sub } P(t \ x) = c \mid \text{sub } P(\text{bool } x) = c
\]
\[
c ::= \cdots \mid P(e)
\]

\[
\begin{align*}
\text{Proc}_0 &= \text{Stores } \rightarrow \text{Stores} \\
\text{Proc}_v &= (\text{Stores } \times (\text{int } \cup \text{bool})) \rightarrow \text{Stores} \\
\text{Proc} &= \text{Proc}_0 + \text{Proc}_v \\
\text{Env} &= \{ \gamma \mid \gamma : X \rightarrow (\text{Loc } + \text{Proc}) \}
\end{align*}
\]
Semantics of Call-by-value

**DSubv**
\[
\frac{\gamma \vdash \langle \sigma, \text{sub } P(t\ x) = c \rangle \rightarrow_d \langle [P \mapsto \text{proc}_v(t\ x, c, \gamma)]\gamma, \sigma \rangle}{\text{where } t \in \{\text{int}, \text{bool}\}}
\]

\[
\frac{\text{CrecSubv}}{\gamma \vdash \langle \sigma, e \rangle \rightarrow_e^\ast v}
\]

\[
\frac{\gamma_2 \vdash \langle [l \mapsto v]_\sigma, c \rangle \rightarrow_c^\ast \sigma'}{(\gamma(P) = \text{proc}_v(t\ x, c, \gamma_1))}
\]

\[
\frac{\gamma \vdash \langle \sigma, P(e) \rangle \rightarrow_c \sigma' \upharpoonright \text{Dom}(\sigma)}{\gamma_2 = [x \mapsto l][P \mapsto \gamma(P)]\gamma_1,}
\]

\[
l \notin \text{Range}(\gamma) \cup \text{Dom}(\sigma) \quad \text{and}
\]

\[
\gamma(P) = \text{proc}_v(t\ x, c, \gamma_1).
\]
Subroutines with Reference Parameters

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.

\[
\begin{align*}
  d_1, d_2, d &::= \cdots \quad \text{sub } P(\text{ref } t \ x) = c \quad \text{sub } P(\text{ref bool } x) = c \\
  c &::= \cdots \quad P(x)
\end{align*}
\]

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.
Semantics of Call-by-Reference

We augment the definition of $Proc$ to include a new entity viz. $Proc_r$. We then have

$$
\begin{align*}
Proc_0 &= \text{Stores} \to \text{Stores} \\
Proc_v &= (\text{Stores} \times (\text{int} \cup \text{bool})) \to \text{Stores} \\
Proc_r &= (\text{Stores} \times \text{Loc}) \to \text{Stores} \\
Proc &= Proc_0 + Proc_v + Proc_r \\
Env &= \{\gamma \mid \gamma : X \to (\text{Loc} + Proc)\}
\end{align*}
$$

$$
\begin{array}{c}
\text{DSubr} \\
\frac{\gamma \vdash \langle \sigma, \text{sub} \ P(t \ x) = c \rangle \longrightarrow_d \langle [P \mapsto \text{proc}_r(t \ x, c, \gamma)] \gamma, \sigma \rangle}{\gamma \vdash \langle \sigma, \text{sub} \ P(t \ x) = c \rangle \longrightarrow_d \langle [P \mapsto \text{proc}_r(t \ x, c, \gamma)] \gamma, \sigma \rangle}
\end{array}
$$

where $t \in \{\text{int, bool}\}$

$$
\begin{array}{c}
\text{CrecSubr} \\
\frac{\gamma_2 \vdash \langle [\sigma, c] \longrightarrow_c \sigma' \rangle \text{ or } \gamma \vdash \langle \sigma, P(y) \rangle \longrightarrow_c \sigma' \text{ or } (\gamma(P) = \text{proc}_r(t \ x, c, \gamma_1))}{\gamma \vdash \langle \sigma, P(y) \rangle \longrightarrow_c \sigma' \text{ or } (\gamma(P) = \text{proc}_r(t \ x, c, \gamma_1))}
\end{array}
$$

where

- $\gamma_2 = [x \mapsto \gamma(y)][P \mapsto \gamma(P)]\gamma_1$,

Notice that no new location is needed in the call-by-reference mechanism.
13.6. Runtime Structure

A Calling Chain

Main → P1 → P2 → P21 → P21
Run-time Structure: 1
Run-time Structure: 2

Main program
  Globals
  Procedure P2
    Locals of P2
      Procedure P21
        Locals of P21
        Body of P21
        Body of P2
  Procedure P1
    Locals of P1
    Body of P1
  Main body

Main → P1

Return address to Main
  Dynamic link to Main
  Locals of P1
  Static link to Main
  Formal par of P1
  Globals
Run-time Structure: 3

Main program
- Globals
- Procedure P2
  - Locals of P2
    - Procedure P21
      - Locals of P21
      - Body of P21
    - Body of P2
- Procedure P1
  - Locals of P1
    - Body of P1
- Main body

- Return address to last of P1
  - Dynamic link to last P1
  - Locals of P2
  - Static link to last P1
  - Formal par P2
- Return address to Main
  - Dynamic link to Main
  - Locals of P1
  - Static link to Main
  - Formal par of P1
- Globals

Main → P1 → P2
Run-time Structure: 4

Main program
  - Globals
  - Procedure P2
    - Locals of P2
      - Procedure P21
        - Locals of P21
          - Body of P21
        - Body of P2
    - Procedure P1
      - Locals of P1
        - Body of P1
      - Body of P2

Main → P1 → P2 → P21
Run-time Structure: 5

Main program

Globals

Procedure P2

Locals of P2

Procedure P21

Locals of P21

Body of P2

Main body

Return address to Main
Dynamic link to Main
Locals of P1
Static link to Main
Formal par of P1

Return address to last of P1
Dynamic link to last P2
Locals of P21
Static link last P2
Formal par P21

Return address to last of P2
Dynamic link to last P21
Locals of P2
Static link last P2
Formal par P2

Return address to last of P21
Dynamic link to last P21
Locals of P21
Static link last P2
Formal par P21

Main → P1 → P2 → P21 → P21