Programming Languages
http://www.cse.iitd.ac.in/~sak/courses/pl/2018-19.index.html

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

February 8, 2019
## Contents

1. **The Programming Languages Geneology**
   
2. **Introduction**
   
3. **Scanning or Lexical Analysis**
   - 3.1 Regular Expressions
   - 3.2 Nondeterministic Finite Automata (NFA)
   - 3.3 Deterministic Finite Automata (DFA)
   
4. **Parsing or Syntax Analysis**
   - 4.1 Grammars
   - 4.2 Context-Free Grammars
   - 4.3 Ambiguity
   - 4.4 Shift-Reduce Parsing
   - 4.5 Bottom-Up Parsing
   - 4.6 Recursive Descent Parsing
   - 4.7 Specification of Syntax: Extended Backus-Naur Form
     - 4.7.1 The Extended Backus-Naur Form (EBNF)
5 Attributes & Semantic Analysis

5.1 Syntax-Directed Translation ...................................................... 232
  5.1.1 Synthesized Attributes ..................................................... 242
  5.1.2 Inherited Attributes ......................................................... 261

6 Abstract Syntax ................................................................. 276

7 Symbol Table ........................................................................ 290

8 Intermediate Representation ..................................................... 297

9 The Pure Untyped Lambda Calculus: Basics ............................. 319

10 Notions of Reduction .............................................................. 333

11 Confluence Definitions ........................................................... 343

12 The Church-Rosser Property .................................................... 354

13 Confluence Characterizations ................................................... 360

14 An Applied Lambda-Calculus ................................................... 368
  14.1 FL with recursion ........................................................... 370
15 Logic Programming and Prolog
1. The Programming Languages Geneology
The Landscape of General PLs
The Usage of General PLs

Diagram showing the evolution of programming languages from 1955 to 2000, with categories such as Scientific & Numerical Computations, Business Data Processing, Structured Programming, List processing, Theorem Proving, Symbol processing, and more. Languages such as Fortran, COBOL, Algol 60, Lisp, APL, and more are interconnected, illustrating their relationships and influences from each other.
The Major Features of General PLs
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
Prolog

• First Declarative programming language
• Uses the Horn clause subset of first-order logic
• Goal-oriented programming implementing a top-down methodology
• Implements backtracking as a language feature
• Powerful pattern-matching facilities like in functional programming
• Various dialects implement various other features such as constraint programming, higher-order functions etc.
2. Introduction
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. \LaTeX).
• Model-checking and verification software are based on compiling techniques
• Synthesis of hardware circuits requires a description language and the final code that is generated is an implementation either at the register-transfer level or gate-level design.
Books and References

1. **Appel A W.** *Modern Compiler Implementation in Java* Cambridge University Press, Revised Indian Paperback edition 2001


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

SCANNER

stream of characters

stream of tokens

PARSER

parse tree
The Big Picture: 3

---

**SCANNER**

- stream of characters
- stream of tokens

**PARSER**

- parse tree

**SEMANTIC ANALYZER**

- 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

- **SCANNER**
  - stream of characters
  - 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

Diagram:

- **SCANNER**
  - stream of characters
  - stream of tokens

- **PARSER**
  - parse tree

- **SEMANTIC ANALYZER**
  - abstract syntax tree

- **I.R. CODE GENERATOR**
  - intermediate representation

- **OPTIMIZER**
  - optimized intermediate representation

- **CODE GENERATOR**
  - target code
The Big Picture: 7

- Scanner
- Parser
- Semantic Analyzer
- I.R. Code Generator
- Optimizer
- Code Generator
- Symbol Table Manager

Stream of characters → Scanner → Stream of tokens → Parser → Parse tree → Semantic Analyzer → Abstract syntax tree → I.R. Code Generator → Optimizer → Intermediate representation → Optimized intermediate representation → Code Generator → Target code
The Big Picture: 8

Scanner | Parser | Semantic Analysis | Symbol Table
---|---|---|---
**IR** | **Optimization** | |
3. Scanning or Lexical Analysis

Lexical Analysis
Programming Language Elements

- 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.
- A programming language also has a finite set of operators.
- The phrases, clauses and sentences of a programming language are expressions, commands, functions, procedures and programs.
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

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

Non-tokens 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

During the scanning phase the compiler/interpreter

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

- Whitespace: A sequence of space, tab, newline, carriage-return, form-feed characters etc.
- Lexeme: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators or punctuation symbols)
- Examples of lexemes.
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives
Scanning: 3

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

- Token: A sequence of characters to be treated as a single unit.
- Examples of tokens.
  - Reserved words (e.g. `begin`, `end`, `struct`, `if` etc.)
  - Keywords (`integer`, `true` etc.)
  - Operators (`+`, `&&`, `++` etc)
  - Identifiers (variable names, procedure names, parameter names)
  - Literal constants (numeric, string, character constants etc.)
  - Punctuation marks (`:`, `,` etc.)
Scanning: 4

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

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

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

  1. An identifier consists of a sequence of 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.
3.1. Regular Expressions
Specifying Lexical Rules

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

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

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

- 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 $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the language specified by the regular expression $r$.

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

**Concatenation** For any two regular expressions $r$ and $s$, $r.s$ or simply $rs$ denotes the concatenation of the languages specified by $r$ and $s$. That is,

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

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

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

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.

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

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

\[
x^0 = ""
\]
\[
x^1 = \text{Selassie}
\]
\[
x^2 = \text{Selassie}\text{Selassie}
\]
\[
\vdots
\]
\[
x^5 = \text{Selassie}\text{Selassie}\text{Selassie}\text{Selassie}\text{Selassie}\text{Selassie}
\]

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

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

\[
X^0 = "\\" \\
X^1 = X \\
X^2 = X \cdot X \\
\vdots \\
X^{n+1} = X \cdot X^n = X^n \cdot X \\
\vdots
\]

Then

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

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

\[
\begin{align*}
X^0 &= \{"\}
\smallskip
X^1 &= \{\text{Mengesha, Gemechis}\}
\smallskip
X^2 &= \{\text{MengeshaMengesha, GemechisMengesha, MengeshaGemechis, GemechisGemechis}\}
\smallskip
X^3 &= \{\text{MengeshaMengeshaMengesha, GemechisMengeshaMengesha, MengeshaGemechisMengesha, GemechisGemechisMengesha, MengeshaMengeshaGemechis, GemechisMengeshaGemechis, GemechisMengeshaGemechisGemechis, GemechisGemechisGemechisGemechis}\}
\smallskip
\vdots
\end{align*}
\]

\[
X^{n+1} = X \cdot X^n
\]

\]

::
### Kleene Closure

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

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

In general,

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

denotes an infinite union of languages.

Further it is easy to show the following identities.

$$ r^* = \varepsilon r r^* $$  \hspace{1cm} (1)

$$ r^* = (r^*)^* $$  \hspace{1cm} (2)
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 (1)

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

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

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

- Multiple ranges: \([a-c0-3] = [a-c] | [0-3]\)

**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*. 
Equivalence of Regular Expressions

Definition 3.3 Let $\text{REGEXP}_A$ denote the set of regular expressions over a finite non-empty set of symbols $A$ and let $r, s \in \text{REGEXP}_A$. Then

- $r \leq_A s$ if and only if $L(r) \subseteq L(s)$ and
- they are equivalent (denoted $r =_A s$) if they specify the same language, i.e.

$$r =_A s \text{ if and only if } L(r) = L(s)$$

We have already considered various identities (e.g. (1)) giving the equivalence between different regular expressions.
Notes on bracketing and precedence of operators

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

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

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

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

**Example 3.4** 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 3.5** The regular expression $r.s|t.u$ is ambiguous because we do not know beforehand whether it represents $(r.s)|(t.u)$ or $r.(t.s|t).u$ 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 3.6** 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 3.7** 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 3.8** Another non-associative operator in arithmetic which often leaves students confused is the exponentiation operator. Consider the arithmetic expression $a^{bc}$. For $a = 2$, $b = 3$, $c = 4$ is this expression to be interpreted as $a^{(bc)}$ or as $(a^b)^c$?

**Exercise 3.1**

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

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) $\mathcal{N}$ over a finite alphabet $A$ 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 $\rightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$. Equivalently

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

is a function that for each source state $q \in Q$ and symbol $a \in A$ associates a (possibly empty) 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 a 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 to 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

$$q_0 \xrightarrow{\varepsilon} \cdots a_1 \xrightarrow{\varepsilon} \cdots q_1 \xrightarrow{\varepsilon} \cdots \cdots a_2 \xrightarrow{\varepsilon} \cdots \cdots \cdots \xrightarrow{\varepsilon} q_n$$

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

$$q_0 \xrightarrow{\varepsilon} \cdots a_1 \xrightarrow{\varepsilon} \cdots q'_1 \xrightarrow{\varepsilon} \cdots \cdots a_2 \xrightarrow{\varepsilon} \cdots \cdots \cdots \xrightarrow{\varepsilon} q'_n$$

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 \textit{accepted} or \textit{recognized} by a NFA is the set of strings that can be accepted by the NFA.

• \( \mathcal{L}(N) \) is the language accepted by the NFA \( N \).
Construction of NFAs

- We show how to construct an NFA to accept a certain language of strings from the regular expression specification of the language.

- The method of construction is by induction on the structure of the regular expression. That is, for each regular expression operator, we show how to construct the corresponding automaton assuming that the NFAs corresponding to individual components of expression have already been constructed.

- For any regular expression $r$ the corresponding NFA constructed is denoted $N_r$. Hence for the regular expression $r|s$, we construct the NFA $N_{r|s}$ using the NFAs $N_r$ and $N_s$ as the building blocks.

- Our method requires only one initial state and one final state for each automaton. Hence in the construction of $N_{r|s}$ from $N_r$ and $N_s$, the initial states and the final states of $N_r$ and $N_s$ are not initial or final unless explicitly used in that fashion.
Constructing NFA

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

![Diagram showing initial state, accepting state, typical transition, and typical NFA for $r$.]
Regular Expressions to NFAs:1

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

<table>
<thead>
<tr>
<th>$N_a$</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>a</td>
</tr>
<tr>
<td>$a_0$</td>
<td>${a_f}$</td>
</tr>
<tr>
<td>$a_f$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{c}
\varepsilon_0 \xrightarrow{\varepsilon} \varepsilon_f \\
N_\varepsilon
\end{array}
\]

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\varepsilon_0</td>
<td>0 0 \cdots 0 { \varepsilon_f }</td>
</tr>
<tr>
<td>\varepsilon_f</td>
<td>0 0 \cdots 0 0</td>
</tr>
</tbody>
</table>
### Regular Expressions to NFAs:3

#### Diagram

![Diagram showing regular expression to NFA conversion]

#### Table

| $N_{r|s}$ | Input Symbol |
|------------|--------------|
| State      | a · · · ε    |
| $r | s_0$      | ∅ · · · $\{r_0, s_0\}$ |
| $r_0$      | · · · · · · |
| $r_f$      | · · · · · · $\{r | s_f\}$ |
| $s_0$      | · · · · · · |
| $s_f$      | · · · · · · $\{r | s_f\}$ |
| $r | s_f$      | ∅ · · · ∅ |
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

\[
N_{r^*} & \quad \text{Input Symbol} \\
\text{State} & \quad a & \cdots & \varepsilon \\
\begin{array}{c}
\begin{array}{c}
\varepsilon \\
r_0^* \\
r_0 \\
\vdots \\
r_f \\
r_f^*
\end{array}
\end{array} & \begin{array}{c}
\emptyset \\
\{r_0, r_f^*\} \\
\emptyset \\
\emptyset \\
\{r_0, r_f^*\} \\
\emptyset \\
\emptyset \\
\emptyset
\end{array}
\]
Regular expressions vs. NFAs

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

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

• Each regular expression operator
  – adds at most 2 new states and
  – adds at most 4 new transitions

• Every state of each $N_r$ so constructed has
  – either 1 outgoing transition on a symbol from $A$
  – or at most 2 outgoing transitions on $\varepsilon$

• 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:5
- For ease of understanding we use the regular expression itself (subscripted by $0$ and $f$ respectively) to name the two new states created by the regular expression operator.
Example:-6

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

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

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

4. Consider a new regular expression operator $\hat{r}$ defined by

$$L(\hat{r}) = A^* - L(r)$$

What is the automaton $N_{\hat{r}}$ given $N_r$?
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 an $\varepsilon$ 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.
$\varepsilon$-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 $\varepsilon$-CLOSE

Require: $T$ a set of states of the NFA
Ensure: $T_{\varepsilon} = \varepsilon\text{-CLOSE}(T)$.

1. $U := T$
2. repeat
3. $U_{old} := U$
4. $U := U_{old} \cup \{q' | q' \notin U, \exists q \in U_{old} : q \xrightarrow{\varepsilon} q'\}$
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  

\[
S := \varepsilon\text{-CLOSURE}\{q_0\}. \\
a := \text{nextchar}(x) \\
\text{while } a \neq \text{end\_of\_string} \text{ do} \\
S := \varepsilon\text{-CLOSURE}(S \xrightarrow{a}) \\
a := \text{nextchar}(x) \\
\text{end while} \\
\text{return } S \cap F \neq \emptyset
\]

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 $O(|x| \cdot |Q_{N_r}|)$ where $|Q_{N_r}|$ is the number of states in $N_r$.

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

- Given that $\varepsilon$-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.
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. the transition relation is no longer a relation but a function$^a$

$$\delta : Q \times A \rightarrow Q$$

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

$^a$Also in the case of the NFA the relation $\rightarrow$ may not define a transition from every state on every letter
Transition Tables of NFAs

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

1. for each state there could be $\varepsilon$ 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 target states or
   (b) a set of target states consisting of a single state or
   (c) a set of target states consisting of more than one state
Transition Tables of DFAs

In the case of a deterministic automaton
1. there are no $\varepsilon$ transitions, and
2. for each state $q$ and letter $a$
   (a) either there is no transition (in which case we add a new “sink” state which is a non-accepting state)
   (b) or there is a transition to a unique state $q'$.

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

Let \( N = \langle Q_N, A \cup \{\varepsilon\}, s_N, F_N, \rightarrow_N \rangle \) 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
- \( F_N \subseteq Q_N \) the 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
- \( \delta_D : Q_D \times A \rightarrow Q_D \) 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 thought to be similar.
NFA to DFA construction

Algorithm 3 Construction of DFA from NFA

Require: NFA $N = \langle Q_N, A \cup \{\varepsilon\}, s_N, F_N, \rightarrow_N \rangle$
Ensure: DFA $D = \langle Q_D, A, s_D, F_D, \delta_D \rangle$ with $\mathcal{L}(N) = \mathcal{L}(D)$

1. $s_D := \varepsilon$-CLOSURE($\{s_N\}$);
2. $Q_D := \{s_D\}$; $F_D := \emptyset$; $\delta_D := \emptyset$
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($q_D a \rightarrow_N$) \{Note: $q_D \subseteq Q_N$\}
8. $\delta_D(q_D, a) := q'_D$
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

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

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

\[ 2 \xrightarrow{a}_N 4 \text{ and } 7 \xrightarrow{a}_N 8. \]

So \[ EC_0 \xrightarrow{a}_D \varepsilon\text{-CLOSURE}(4, 8) = EC_{4, 8}. \]

Similarly

\[ EC_0 \xrightarrow{b}_D \varepsilon\text{-CLOSURE}(5) = EC_5 \]

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

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

\[ EC_5 \xrightarrow{a}_D \varepsilon\text{-CLOSURE}(4, 8) = EC_{4, 8} \text{ and } EC_5 \xrightarrow{b}_D \varepsilon\text{-CLOSURE}(5) \]

\[ EC_{4, 8} \xrightarrow{a}_D \varepsilon\text{-CLOSURE}(4, 8) = EC_{4, 8} \text{ and } EC_{4, 8} \xrightarrow{b}_D \varepsilon\text{-CLOSURE}(5) \]

\[ \varepsilon\text{-CLOSURE}(5, 9) = EC_{5, 9} \]

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

\[ EC_{5, 9} \xrightarrow{a}_D \varepsilon\text{-CLOSURE}(4, 8) = EC_{4, 8} \text{ and } EC_{5, 9} \xrightarrow{b}_D \varepsilon\text{-CLOSURE}(5) \]

\[ \varepsilon\text{-CLOSURE}(5, 10) = EC_{5, 10} \]

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

\[ EC_{5, 10} \xrightarrow{a}_D \varepsilon\text{-CLOSURE}(4, 8) \text{ and } EC_{5, 10} \xrightarrow{b}_D \varepsilon\text{-CLOSURE}(5) \]
Final DFA

\[ D_{(ab)^*abb} \]

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( EC_0 )</td>
<td>( EC_{4,8} )</td>
<td>( EC_5 )</td>
<td></td>
</tr>
<tr>
<td>( EC_{4,8} )</td>
<td>( EC_{4,8} )</td>
<td>( EC_{5,9} )</td>
<td></td>
</tr>
<tr>
<td>( EC_5 )</td>
<td>( EC_{4,8} )</td>
<td>( EC_5 )</td>
<td></td>
</tr>
<tr>
<td>( EC_{5,9} )</td>
<td>( EC_{4,8} )</td>
<td>( EC_{5,10} )</td>
<td></td>
</tr>
<tr>
<td>( EC_{5,10} )</td>
<td>( EC_{4,8} )</td>
<td>( EC_{5,9} )</td>
<td></td>
</tr>
</tbody>
</table>
Scanning: 5

The Big Picture
Exercise 3.2

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 treat such commands also in your DFA where one command is a prefix of another.
4. Parsing or Syntax Analysis

4.1. Grammars

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.
### 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.
Context-Free Grammars: Definition

Definition 4.3 A grammar $G = \langle N, T, P, S \rangle$ is called context-free if

- each production is of the form $X \rightarrow \alpha$, 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:

- \[ S \Rightarrow ab \]
- \[ S \Rightarrow aSb \Rightarrow aabb \]
- \[ S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb \]
- \[ S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbb \]

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

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

(3)

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

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

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

Definition 4.8 A production rule of a context-free grammar is
Right Linear: if it is of the form $X \rightarrow a$ 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 4.9 A regular grammar is a context-free grammar whose productions are either only right linear or only left linear.
Consider the **DFA** with the states renamed as shown below. We could easily convert the DFA to a right linear grammar which generates the language accepted by the DFA.
CFG: Empty word

\[ G = \langle \{S\}, \{a, b\}, P, S\rangle, \text{ where } S \rightarrow SS \mid aSb \mid \varepsilon \]
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!}

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

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 $\epsilon$ must be shown explicitly in the tree.
CFG: Derivation trees 2

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

Another Derivation tree of

\[ \text{abaabb} \]
Yet another Derivation tree of \(abaabb\)
4.3. 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.

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

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

\[ G_1 = \langle \{E, I, C\}, \{y, z, 4, *, +\}, P_1, \{E\} \rangle \]

where \( P_1 \) consists of the following productions.

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

Consider the sentence \( y + 4 * z \).
Ambiguity: 3

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

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

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

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

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

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

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

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

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

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

$$
E 
\Rightarrow 
E + E 
\Rightarrow 
I + E 
\Rightarrow 
y + E 
\Rightarrow 
y + E \ast E 
\Rightarrow 
y + C \ast E 
\Rightarrow 
y + 4 \ast E 
\Rightarrow 
y + 4 \ast I 
\Rightarrow 
y + 4 \ast z
$$
Left-most Derivation 2

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

\[
\begin{align*}
E & \Rightarrow \\
E\ast E & \Rightarrow \\
E + E\ast E & \Rightarrow \\
I + E\ast E & \Rightarrow \\
y + E\ast E & \Rightarrow \\
y + C\ast E & \Rightarrow \\
y + 4\ast E & \Rightarrow \\
y + 4\ast I & \Rightarrow \\
y + 4 \ast z & \Rightarrow 
\end{align*}
\]
Right-most Derivation 1

Right-most derivation of $y+4*z$ corresponding to the \textit{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.

\[
\begin{align*}
E & \Rightarrow \\
E*E & \Rightarrow \\
E*I & \Rightarrow \\
E*z & \Rightarrow \\
E+E*z & \Rightarrow \\
E+C*z & \Rightarrow \\
E+4*z & \Rightarrow \\
I+4*z & \Rightarrow \\
y+4*z & \Rightarrow
\end{align*}
\]
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 $P'$.

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

and compare it with the grammar $G_1$
Left-most Derivation 1’

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

$$
\begin{align*}
E & \Rightarrow \\
E+T & \Rightarrow \\
I+T & \Rightarrow \\
y+T & \Rightarrow \\
y+T*F & \Rightarrow \\
y+T*F & \Rightarrow \\
y+F*F & \Rightarrow \\
y+C*F & \Rightarrow \\
y+4*F & \Rightarrow \\
y+4*I & \Rightarrow \\
y + 4 \times z
\end{align*}
$$
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 \( y+4*z \) corresponding to the first derivation tree.

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

Compare it with the Right-most Derivation 1.
There is no derivation corresponding to Right-most Derivation 2.
Disambiguation by Parenthesization

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

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

\[
E \rightarrow I \mid C \mid (E+E) \mid (E*E) \\
I \rightarrow y \mid z \\
C \rightarrow 4
\]

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 ababaaaba 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 $+,-,\times,/,\%$ and the bracketing symbols ( and ), assuming the usual rules of precedence among operators that you have learned in school.

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

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

7. The language specified by a regular expression can also be generated by a context-free grammar.
   (a) Design a context-free grammar to generate all floating-point numbers allowed by the C language.
   (b) Design a context-free grammar to generate all numbers in binary form that are not multiples of 4.
   (c) Write a regular expression to specify all numbers in binary form that are multiples of 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.
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

\[
\begin{align*}
y + 4 \ast z & \Leftarrow \\
I + 4 \ast z & \Leftarrow \\
E + 4 \ast z & \Leftarrow \\
E + C \ast z & \Leftarrow \\
E + E \ast z & \Leftarrow \\
E \ast z & \Leftarrow \\
E \ast I & \Leftarrow \\
E \ast E & \Leftarrow \\
E & \Leftarrow 
\end{align*}
\]
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 \( X \).
Parsing: 0

r1. E → E − T
r2. E → T
r3. T → T ⊕ D
r4. T → D
r5. D → a | b | ( E )
Principle:
Reduce whenever possible.
Shift only when reduce is impossible
Parsing: 2

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

r2 $E \rightarrow T$

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

r4 $T \rightarrow D$

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

Reduce by r5
Parsing: 3

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: 4

r1. $E \rightarrow E \cdot T$

r2 $E \rightarrow T$

r3 $T \rightarrow T \oplus D$

r4 $T \rightarrow D$

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

Reduce by $r2$
Parsing: 5

r1. $E \to E T$

r2. $E \to T$

r3. $T \to T \lor D$

r4. $T \to D$

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

Shift
Parsing: 6

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

Shift
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. $E \longrightarrow E \mathbf{a} \mathbf{b} T$

r2. $E \longrightarrow T$

r3. $T \longrightarrow T \mathbf{a} \mathbf{b} D$

r4. $T \longrightarrow D$

r5. $D \longrightarrow \mathbf{a} | \mathbf{b} | ( \mathbf{E} \mathbf{a} )$

Reduce by r4
Parsing: 8a

r1. $E \rightarrow E \rightarrow T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \rightarrow D$

r4. $T \rightarrow D$

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

Reduce by r4
Parsing: 9a

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 r1
Parsing: 10a

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

Shift
Parsing: 11a

1. E → E T
2. E → T
3. T → T D
4. T → D
5. D → a | b | ( E )
Parsing: 12a

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

Reduce by r5
Parsing: 13a

r1. \( E \rightarrow E \quad T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \cup D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \ | \ b \ | \ ( E ) \)

Reduce by r4
Parsing: 14a

r1. \[ E \rightarrow E - T \]
r2. \[ E \rightarrow T \]
r3. \[ T \rightarrow T \cup D \]
r4. \[ T \rightarrow D \]
r5. \[ D \rightarrow a | b | ( E ) \]

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 )

Reduce by r2

Get back!
Parsing: 13b

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

Reduce by r4

Get back!
Parsing: 12b

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

Get back!

Reduce by r5
Parsing: 11b

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \cup \diamond D$

r4. $T \rightarrow D$

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

Get back!  Shift
Parsing: 10b

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!

Shift
Parsing: 9b

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

Get back to where you once belonged!

Reduce by r1
Parsing: 8b

Principle:
Reduce whenever possible, but depending upon lookahead.

Shift instead of reduce here!

Shift–reduce conflict

Reduce by r4
Parsing: 8

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

Reduce by r4
Parsing: 9

1. E → E T
2. E → T
3. T → T D
4. T → D
5. D → a | b | ( E )

Shift
Parsing: 10

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

r2. $E \rightarrow \mathbf{T}$

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

r4. $T \rightarrow D$

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

Shift
Parsing: 11

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

Reduce by r5
Parsing: 12

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

Reduce by r3
Parsing: 13

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

Reduce by r1
4.5. Bottom-Up Parsing

Bottom-Up Parsing
Parse Trees: 0

r1. E → E − T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )
Parse Trees: 1

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

shift-reduce parsing: 1
Parse Trees: 2

Parse trees for the given grammar:

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. $E \rightarrow E \cdot T$

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: 3
Parse Trees: 3a

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \div D$

r4. $T \rightarrow D$

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

shift-reduce parsing
Parse Trees: 3b

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

shift-reduce parsing
Parse Trees: 4

r1. \( E \rightarrow E \mathord{-} T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \mathord{\langle} D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mathord{|} b \mathord{|} (E) \)

shift-reduce parsing
Parse Trees: 5

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

r2. \( E \rightarrow T \)

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

r4. \( T \rightarrow D \)

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

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

shift-reduce parsing
Parse Trees: 5b

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: 6

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: 7

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

shift-reduce parsing
Parse Trees: 8

r1. \[ E \rightarrow E - T \]

r2. \[ E \rightarrow T \]

r3. \[ T \rightarrow T / D \]

r4. \[ T \rightarrow D \]

r5. \[ D \rightarrow \text{a} \mid \text{b} \mid (E) \]

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
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 \textit{indirectly} left recursive. For instance, the following is \textit{indirect} left-recursion of cycle length 2.

\begin{example}
\begin{align*}
A & \rightarrow B\beta \\
B & \rightarrow A\alpha
\end{align*}
\end{example}

\textit{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 4.12** 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.
4.7. Specification of Syntax: Extended Backus-Naur Form

Specification of Syntax: EBNF
The EBNF specification of a programming language is a collection of rules that defines the (context-free) grammar of the language. It specifies the formation rules for the correct grammatical construction of the phrases of the language.

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 4.13 A context-free grammar for balanced parentheses (including the empty string) over the terminal alphabet \{ (, [, , }\} could be given as $BP_3 = \langle \{S\}, \{ (, [, , \}, \{S\} \rangle$, 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 4.14 $BP_3$ may be expressed in EBNF as follows:

- **BracketSeq** ::= \{Bracket\}.
- **Bracket** ::= LeftParen BracketSeq RightParen | LeftSqbracket BracketSeq RightSqbracket | LeftBrace BracketSeq RightBrace.
- **LeftParen** ::= “(”.
- **RightParen** ::= “)”. 
- **LeftSqbracket** ::= “[”.
- **RightSqbracket** ::= “]”.
- **LeftBrace** ::= “{”.
- **RightBrace** ::= “}”.


EBNF in EBNF

EBNF has its own grammar which is again context-free. Hence EBNF (4.7.1) 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.

- EBNF also has a diagrammatic rendering called syntax diagrams or railroad diagrams. The grammar of SML has been produced by a set of syntax diagrams.

- Pascal has been defined using both the text-version of EBNF and through syntax diagrams.

- While the text form of EBNF helps in parsing, the diagrammatic rendering is only for the purpose of readability.

- EBNF is a specification language that almost all modern programming languages use to define the grammar of the programming language.
Syntax Specifications

- BNF of C
- BNF of Java
- EBNF of Pascal
- Pascal Syntax diagrams
- BNF of Standard ML
- BNF of Datalog
- BNF of Prolog
Syntax of Standard ML
Tobias Nipkow and Larry Paulson

PROGRAMS AND MODULES

Program

TopLevelDeclaration ;

TopLevelDeclaration

Expression
ObjectDeclaration
SignatureDeclaration
FunctorDeclaration

ObjectDeclaration

Declaration
structure Ident : Signature = Structure
and
local ObjectDeclaration in ObjectDeclaration end

;
Syntax Diagrams of SML: 2

Signature Declaration

Functor Declaration

Functor Binding

Functor Arguments

Structure

Signature
DECLARATIONS

Declaration

val Pattern = Expression

rec

and

fun FunHeading = Expression

- Type

1

and

type TypeBinding

datatype DatatypeBinding

withtype TypeBinding

abstype DatatypeBinding

withtype TypeBinding

with Declaration end

exception Name of Type

- CompoundName

and

local Declaration in Declaration end

open CompoundIdent

infix

infixr

nonfix

Digit

Ident

:
Syntax Diagrams of SML: 5

### FunHeading

- **Name**
- **AtomicPattern**

  - **InfixOperator**
  - **AtomicPattern**

### TypeBinding

- **TypeVarList**
- **Ident**
- **=**
- **Type**

- **and**

### DatatypeBinding

- **TypeVarList**
- **Ident**
- **of**
- **Type**

- **and**

### TypeVarList

- **TypeVar**

  - **(**
  - **TypeVar**
  - **)**
EXPRESSIONS

Expression

InfixExpression

Expression : Type

Expression andalso Expression

Expression handle Match

raise Expression

if Expression then Expression else Expression

while Expression do Expression

case Expression of Match

fn Match

InfixExpression

AtomicExpression

InfixExpression InfixOperator InfixExpression
**Matches and Patterns**

**Match**

```
Pattern ∋ Expression
```

**Pattern**

```
AtomicPattern
```

```
ModuleName AtomicPattern
Pattern InfixOperator Pattern
Pattern Type Pattern
Name Type as Pattern
```
1. AtomicPattern
   - /
lad
   - /
   - /
   - /
   - /
   - /
CompoundName
   - Constant
   - (Pattern
   - )
   - [Pattern
   - ]
   - 
   - FieldPattern

FieldPattern

...
TYPES

Type

LEXICAL MATTERS: IDENTIFIERS, CONSTANTS, COMMENTS

CompoundIdent

CompoundName

Name

InfixOperator

*(any Ident that has been declared to be infix)*
Syntax Diagrams of SML: 10

1. Constant
   Numeral
   /
   /
   /
   /
   E
   Numeral
   /
   /
   /
   /
   /
   StringEscape
   \n
2. StringEscape
   n
   t
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   Space
   Tab
   Newline
   Formfeed

3. Numeral
   /
   /
   /
   /
   /
   Digit

4. TypeVar
   /
   /
   /
   /
   /
   AlphanumericIdent
Syntax Diagrams of SML: 11

1. alphanumericIdent
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
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
Attributes & Semantic Analysis
Semantic Analysis: 0

1. Every programming language can be used to program any computable function, assuming of course, it has
   - unbounded memory, and
   - unbounded time

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

   **Example 5.1** The language $\{a^n b^n c^n | n > 0\}$ is not context-free.

3. Semantic analysis is an essential step to generating IR-code, since it requires the computation of certain **bits and pieces of information** called attributes (which include information to be entered into the symbol table or useful for error-handling)

4. These attributes are usually **context-sensitive** in nature. They need to be computed and if necessary propagated during parsing from wherever they are available.
Semantic Analysis: 1

The parser of a programming language provides the framework within which the IR-code or even the target code is to be generated.

The parser also provides a structuring mechanism that divides the task of code generation into bits and pieces determined by the individual nonterminals and production rules.

The parser provides the framework from within which the semantic analysis (which includes the bits and pieces of information that are required for code generation) is performed.
Semantic Analysis: 2

- There are context-sensitive aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - type consistency between declaration and use.
  - correspondence between formal and actual parameters (example 5.1 is an abstraction where $a^n$ represents a function or procedure declaration with $n$ formal parameters and $b^n$ and $c^n$ represent two calls to the same procedure in which the number of actual parameters should equal $n$).
  - scope and visibility issues with respect to identifiers in a program.
5.1. Syntax-Directed Translation

Attributes

An attribute can represent anything we choose e.g.

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

The value of an attribute at a parse-tree node is defined by the semantic rule associated with the production used at that node.
Syntax-Directed Definitions (SDD)

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

1. various attributes

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

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

There are two kinds of attributes that one can envisage.

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

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

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

Inherited attributes are convenient for expressing the dependence of a language construct on the *context* in which it appears.
What is Syntax-directed?

- A syntax-directed definition is a generalisation of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called synthesized and inherited attributes.

- The various attributes are computed by so-called semantic rules associated with each production of the grammar which allows the computation of the various attributes.

- These semantic rules are in general executed during bottom-up (SR) parsing at the stage when a reduction needs to be performed by the given rule and top-down (RDP) parsing in the procedure before the next call or return from the procedure.

- 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 $X$ and/or the grammar symbols of $\alpha$.

**Definition 5.2** Given a production $X \rightarrow \alpha$, an attribute $a$ is **synthesized**: a synthesized attribute of $X$ (denoted $X.a$) or **inherited**: an inherited attribute of one of the grammar symbols of $\alpha$ (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.
- The attribute grammar also specifies how the attributes are propagated through the grammar, by using *graph dependency* between the productions.
- In general, **different occurrences** of the **same** non-terminal symbol in each production will be distinguished by appropriate subscripts when defining the semantic rules associated with the rule.

The following **example** illustrates the concept of a syntax-directed definition using synthesized attributes.
Attribute Grammars: Example

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

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

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

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

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

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

\[
F \rightarrow n \quad \triangleright \quad F\.\text{val} := n\.\text{val}
\]

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

Evaluating the expression \((4 - 1)/2\) generated by the grammar for subtraction and division
Synthesized Attributes: 1

Synthesized Attributes

4 3 2 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
Synthesized Attributes: 13
Synthesized Attributes: 14
An Attribute Grammar

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

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

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

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

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

\[ F \rightarrow n \quad \Rightarrow \quad F.val := n.val \]
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.
5.1.2. Inherited Attributes

Inherited Attributes: 0

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

\[
D \rightarrow T \quad L \\
T \rightarrow \text{int} \mid \text{float} \\
L \rightarrow L, I \mid I \\
I \rightarrow x \mid y \mid z
\]

\[
\text{D} \quad \text{T} \quad \text{L} \\
\text{L} \quad \text{I} \\
\text{I} \\
\text{x} \quad \text{y} \quad \text{z} \\
\text{int}
\]
Inherited Attributes: 1

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

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

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

\[
D \to T \ L \\
L \to L, I \mid I \\
T \to \text{int } \mid \text{float} \\
I \to x \mid y \mid z
\]
Inherited Attributes: 3

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

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

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

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

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

$$D \rightarrow T \mid L \quad T \rightarrow \text{int } \mid \text{float}$$

$$L \rightarrow L, I \mid I \quad I \rightarrow x \mid y \mid z$$
Inherited Attributes: 6

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

\[
D \rightarrow T \ L \\
T \rightarrow \text{int} \mid \text{float} \\
L \rightarrow L, I \mid I \\
I \rightarrow x \mid y \mid z
\]
Inherited Attributes: 7

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

\[
D \rightarrow T \ L \\
T \rightarrow \text{int} \ | \ \text{float} \\
L \rightarrow L, I \ | \ I \\
I \rightarrow x \ | \ y \ | \ z
\]
Attribute Grammar: Inherited

\[
D \rightarrow TL \quad \triangleright \quad L.in := T.type
\]
\[
T \rightarrow \text{int} \quad \triangleright \quad T.type := \text{int}.\text{int}
\]
\[
T \rightarrow \text{float} \quad \triangleright \quad T.type := \text{float}.\text{float}
\]
\[
L_0 \rightarrow L_1, I \quad \triangleright \quad L_1 := L_0.in
\]
\[
L \rightarrow I \quad \triangleright \quad I.in := L.in
\]
\[
I \rightarrow \text{id} \quad \triangleright \quad \text{id}.\text{type} := I.in
\]
L-attributed Definitions

Definition 5.3 A grammar is L-attributed if for each production of the form $Y \rightarrow X_1 \ldots X_k$, each inherited attribute of the symbol $X_j$, $1 \leq j \leq k$ depends only on

1. the inherited attributes of the symbol $Y$ 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.

\[
D \rightarrow L:T \quad \triangleright \quad L.in := T.type \\
T \rightarrow \text{int} \quad \triangleright \quad T.type := \text{int}.\text{int} \\
T \rightarrow \text{real} \quad \triangleright \quad T.type := \text{real}.\text{real} \\
L_0 \rightarrow L_1,I \quad \triangleright \quad L_1 := L_0.\text{in} \\
L \rightarrow I \quad \triangleright \quad I.in := L.in \\
I \rightarrow \text{id} \quad \triangleright \quad \text{id}.type := I.in
\]

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:

\[
D \to \text{id}L \rar addtype(\text{id}, L\.type) \\
L \to :T \rar L\.in := T\.type \\
L \to ,\text{id} L \rar L_0\.type := L_1\.type; \\
\quad addtype(\text{id}.L_1\.type) \\
T \to \text{int} \rar T\.type := \text{int}.\text{int} \\
T \to \text{real} \rar T\.type := \text{real}.\text{real}
\]
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

```
  1
 --
  2
  |
 4 1
```
Evaluation: 1
Evaluation: 2
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

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

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

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

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

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

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

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

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

\[ E \rightarrow T \quad \triangleright \quad E.p\text{tr} := T.p\text{tr} \]

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

\[ T \rightarrow F \quad \triangleright \quad T.p\text{tr} := F.p\text{tr} \]

\[ F \rightarrow (E) \quad \triangleright \quad F.p\text{tr} := E.p\text{tr} \]

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

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.

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

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

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

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

\[ E \rightarrow T \quad \triangleright \quad E.\text{ast} := T.\text{ast} \]

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

\[ T \rightarrow F \quad \triangleright \quad T.\text{ast} := F.\text{ast} \]

\[ F \rightarrow (E) \quad \triangleright \quad F.\text{ast} := E.\text{ast} \]

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

The Big Picture
7. Symbol Table
Symbol Table
Symbol Table: 1

- The store house of context-sensitive and run-time information about every identifier in the source program.
- All accesses relating to an identifier require to first find the attributes of the identifier from the symbol table.
- Usually organized as a hash table – provides fast access.
- Compiler-generated temporaries may also be stored in the symbol table.
Symbol Table: 2

Attributes stored in a symbol table for each identifier:

- type
- size
- scope/visibility information
- base address
- addresses to location of auxiliary symbol tables (in case of records, procedures, classes)
- address of the location containing the string which actually names the identifier and its length in the string pool
Symbol Table:3

- A symbol table exists throughout the compilation and run-time.
- Major operations required of a symbol table:
  - insertion
  - search
  - deletions are **purely logical** (depending on scope and visibility) and not physical
- Keywords are often stored in the symbol table before the compilation process begins.
Symbol Table:4

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

**Scanning**: Insertion of new identifiers.

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

**Semantic analysis:**

- Determination of types of identifiers from declarations
- type checking to ensure that operands are used in type-valid contexts.
- Checking scope, visibility violations.
Symbol Table: 5

**IR generation:** Memory allocation and relative\(^a\) address calculation.

**Optimization:** All memory accesses through symbol table

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

**The Big picture**

\(^a\)\textit{i.e. relative} to a base address that is known only at run-time
8. Intermediate Representation
Intermediate Representation
Intermediate Representation

Intermediate representations are important for reasons of portability i.e. platform (hardware and OS) independence.

• *(more or less) independent* of specific features of the high-level language.
  
  Example. Java byte-code for any high-level language.

• *(more or less) independent* of specific features of any particular target architecture (e.g. number of registers, memory size)
  
  – number of registers
  – memory size
  – word length
IR Properties: 1

1. It is fairly low-level containing instructions common to all target architectures and assembly languages.
   How low can you stoop? ...

2. It contains some fairly high-level instructions that are common to most high-level programming languages.
   How high can you rise?

3. To ensure portability
   - an unbounded number of variables and memory locations
   - no commitment to Representational Issues

4. To ensure type-safety
   - memory locations are also typed according to the data they may contain,
   - no commitment is made regarding word boundaries, and the structure of individual data items.

Next
IR: Representation?

- No commitment to word boundaries or byte boundaries
- No commitment to representation of
  - int vs. float,
  - float vs. double,
  - packed vs. unpacked,
  - strings – where and how?

Back to IR Properties:1
IR: How low can you stoop?

- most arithmetic and logical operations, load and store instructions etc.
- so as to be interpreted easily,
- the interpreter is fairly small,
- execution speeds are high,
- to have fixed length instructions (where each operand position has a specific meaning).

Back to IR Properties:1
IR: How high can you rise?

• typed variables,
• temporary variables instead of registers.
• array-indexing,
• random access to record fields,
• parameter-passing,
• pointers and pointer management
• no limits on memory addresses

Back to IR Properties:1
A typical instruction set: 1

Three address code: A suite of instructions. Each instruction has at most 3 operands.

- an opcode representing an operation with at most 2 operands
- two operands on which the binary operation is performed
- a target operand, which accumulates the result of the (binary) operation.

If an operation requires less than 3 operands then one or more of the operands is made null.
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
A typical instruction set: 2.1

- **Assignments (LOAD-STORE)**
  - \( x := y \ bop \ z \), where \( \text{bop} \) is a binary operation
  - \( x := uop \ y \), where \( \text{uop} \) is a unary operation
  - \( x := y \), load, store, copy or register transfer

- **Jumps (conditional and unconditional)**
- **Procedures and parameters**
- **Arrays and array-indexing**
- **Pointer Referencing and Dereferencing**
A typical instruction set: 2.2

• Assignments (LOAD-STORE)
• Jumps (conditional and unconditional)
  – goto L – Unconditional jump,
  – x relop y goto L – Conditional jump, where relop is a relational operator
• Procedures and parameters
• Arrays and array-indexing
• Pointer Referencing and Dereferencing
A typical instruction set: 2.3

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
  - `call p n`, where `n` is the number of parameters
  - `return y`, return value from a procedures call
  - `param x`, parameter declaration
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
A typical instruction set: 2.4

• Assignments (LOAD-STORE)
• Jumps (conditional and unconditional)
• Procedures and parameters
• Arrays and array-indexing
  – \( x := a[i] \) – array indexing for \( r\)-value
  – \( a[j] := y \) – array indexing for \( l\)-value

Note: The two opcodes are different depending on whether \( l\)-value or \( r\)-value is desired. \( x \) and \( y \) are always simple variables

• Pointer Referencing and Dereferencing
A typical instruction set: 2.5

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
  - $x := \hat{y}$ – referencing: set $x$ to point to $y$
  - $x := *y$ – dereferencing: copy contents of location pointed to by $y$ into $x$
  - $*x := y$ – dereferencing: copy $r$-value of $y$ into the location pointed to by $x$

Picture
IR: Generation Basics

- Can be generated by recursive traversal of the abstract syntax tree.
- Can be generated by syntax-directed translation as follows:
  For every non-terminal symbol $N$ in the grammar of the source language there exist two attributes
  $N$.place, which denotes the address of a temporary variable where the result of the execution of the generated code is stored
  $N$.code, which is the actual code segment generated.
- In addition a global counter for the instructions generated is maintained as part of the generation process.
- It is independent of the source language but can express target machine operations without committing to too much detail.
IR: Infrastructure 1

Given an abstract syntax tree $T$, with $T$ also denoting its root node.

$T\.place$ address of temporary variable where result of execution of the $T$ is stored.

$newtemp$ returns a fresh variable name and also installs it in the symbol table along with relevant information

$T\.code$ the actual sequence of instructions generated for the tree $T$.

$newlabel$ returns a label to mark an instruction in the generated code which may be the target of a jump.

$emit$ emits an instructions (regarded as a string).
IR: Infrastructure 2

Colour and font coding of IR code generation process.

- **Green**: Nodes of the Abstract Syntax Tree
- **Brown**: Characters and strings of the Intermediate Representation
- **Red**: Variables and data structures of the *language* in which the IR code generator is written
- **Blue**: Names of relevant *procedures* used in IR code generation.
- **Black**: All other stuff.
IR: Example 1

\[ E \quad \rightarrow \quad id \quad \triangleleft \]

\[ E.place \quad := \quad id.place; \]
\[ E.code \quad := \quad emit() \]

\[ E_0 \quad \rightarrow \quad E_1 \; - \; E_2 \quad \triangleleft \]

\[ E_0.place \quad := \quad \text{newtemp}; \]
\[ E_0.code \quad := \quad E_1.code \]
\[ \quad |\quad E_2.code \]
\[ \quad |\quad \text{emit}(E_0.place := E_1.place - E_2.place) \]
IR: Example 2

\[ S \rightarrow id := E \]

\[ S.code := E.code \]
\[ \triangledown emit(id.place:=E.place) \]

\[ S_0 \rightarrow while E do S_1 \]

\[ S_0.begin := newlabel; \]
\[ S_0.after := newlabel; \]
\[ S_0.code := emit(S_0.begin:) \]
\[ \triangledown E.code \]
\[ \triangledown emit(\text{if} E.place= 0 \text{ goto } S_0.after) \]
\[ \triangledown S_1.code \]
\[ \triangledown emit(\text{goto } S_0.begin) \]
\[ \triangledown emit(S_0.after:) \]
IR: Example 3

\[ S \quad \rightarrow \quad id := E \]

\[ S.code \quad := \quad E.code \]
\[ \quad || \quad emit(id.place := E.place) \]

\[ S_0 \quad \rightarrow \quad \text{while } E \text{ do } S_1 \]

\[ S_0.begin \quad := \quad \text{newlabel;} \]
\[ S_0.after \quad := \quad \text{newlabel;} \]
\[ S_0.code \quad := \quad \text{emit}(S_0.begin:) \]
\[ \quad || \quad E.code \]
\[ \quad || \quad \text{emit}(\text{if } E.place = 0 \text{ goto } S_0.after) \]
\[ \quad || \quad S_1.code \]
\[ \quad || \quad \text{emit}(\text{goto} S_0.begin) \]
\[ \quad || \quad \text{emit}(S_0.after:) \]
IR: Generation End

While generating the intermediate representation, it is sometimes necessary to generate jumps into code that has not been generated as yet (hence the address of the label is unknown). This usually happens while processing

• forward jumps

• short-circuit evaluation of boolean expressions

It is usual in such circumstances to either fill up the empty label entries in a second pass over the the code or through a process of backpatching (which is the maintenance of lists of jumps to the same instruction number), wherein the blank entries are filled in once the sequence number of the target instruction becomes known.
9. 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

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

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

$\begin{array}{|c|c|c|}
\hline
N & FV(N) & Var(N) \\
\hline
x & \{x\} & \{x\} \\
\lambda x[L] & FV(L) - \{x\} & Var(L) \cup \{x\} \\
(L M) & FV(L) \cup FV(M) & Var(L) \cup Var(M) \\
\hline
\end{array}$

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

Definition 9.2 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:

\[
\begin{align*}
\{N/x\}x & \equiv N \\
\{N/x\}y & \equiv y \quad \text{if } y \not\equiv x \\
\{N/x\}\lambda x[L] & \equiv \lambda x[L] \\
\{N/x\}\lambda y[L] & \equiv \lambda y[\{N/x\}L] \quad \text{if } y \not\equiv x \text{ and } y \not\in \text{FV}(N) \\
\{N/x\}\lambda y[L] & \equiv \lambda z[\{N/x\}\{z/y\}L] \quad \text{if } y \not\equiv x \text{ and } y \in \text{FV}(N) \text{ and } z \text{ is } 'fresh' \\
\{N/x\}(LM) & \equiv (\{N/x\}L \{N/x\}M)
\end{align*}
\]

- In the above definition it is necessary to ensure that the free variables of $N$ continue to remain free after substitution i.e. none of the free variables of $N$ should be "captured" as a result of the substitution.
- The phrase "$z$ is 'fresh'" may be taken to mean $z \not\in \text{FV}(N) \cup \text{Var}(L)$.
- $z$ could be fresh even if $z \in \text{BV}(N)$.
Compatibility

Definition 9.3 A binary relation $\rho \subseteq \Lambda \times \Lambda$ is said to be compatible if $L \rho M$ implies

1. for all variables $x$, $\lambda x[L] \rho \lambda x[M]$ and
2. for all terms $N$, $(L N) \rho (M N)$ and $(N L) \rho (N M)$. 
Compatible Closure

Definition 9.4 The compatible closure of a relation \( \rho \subseteq \Lambda \times \Lambda \) is the smallest (under the \( \subseteq \) ordering) relation \( \rho^c \subseteq \Lambda \times \Lambda \) such that

\[
\begin{align*}
\rho & \vdash \frac{L \rho M}{L \rho^c M} \\
\rho & \vdash \frac{\rho^c\lambda x[L]}{\rho^c\lambda x[M]} \\
\rho_{\text{AppL}} & \vdash \frac{L \rho^c M}{(L N) \rho^c (M N)} \\
\rho_{\text{AppR}} & \vdash \frac{L \rho^c M}{(N L) \rho^c (N M)}
\end{align*}
\]

Lemma 9.5

1. \( \rho^c \supseteq \rho \).
2. The compatible closure of any relation is compatible.
3. If \( \rho \) is compatible then \( \rho^c = \rho \).

Example 9.6

1. \( \equiv_\alpha \) is a compatible relation
2. \( \rightarrow^1_\beta \) is by definition a compatible relation.
\[ \alpha \text{-equivalence} \]

**Definition 9.7 (\(\alpha\)-equivalence)** \(\equiv_{\alpha} \subseteq \Lambda \times \Lambda\) is the **compatible closure** of the relation \(\{(\lambda x[L] \equiv_{\alpha} \lambda y[y/x]L) \mid y \not\in FV(L)\}\).

- 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 9.8

• 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 $L' \equiv_\alpha L$. 

(4)
Untyped $\lambda$-Calculus: 1-step $\beta$-Reduction

Definition 9.9 A 1-step $\beta$-reduction $\rightarrow^1_\beta$ is the smallest relation (under the $\subseteq$ ordering) on $\Lambda$ such that

\[
\begin{align*}
\beta_1 & \quad \frac{L \rightarrow_\beta M}{L \rightarrow^1_\beta M} \\
\beta_1\text{Abs} & \quad \frac{L \rightarrow^1_\beta M}{\lambda x[L] \rightarrow^1_\beta \lambda x[M]} \\
\beta_1\text{AppL} & \quad \frac{L \rightarrow^1_\beta M}{(L \, N) \rightarrow^1_\beta (M \, N)} \\
\beta_1\text{AppR} & \quad \frac{L \rightarrow^1_\beta M}{(N \, L) \rightarrow^1_\beta (N \, M)}
\end{align*}
\]

- $\rightarrow^1_\beta$ 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 9.10

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

\[
\begin{align*}
\beta_n \text{Basis} & : L \rightarrow^0_\beta L \\
\beta_n \text{Induction} & : L \rightarrow^m_\beta M \rightarrow^1_\beta N \\
& \quad \Rightarrow L \rightarrow^{m+1}_\beta N \quad (m \geq 0)
\end{align*}
\]

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

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

Definition 9.11

- 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 can reduce to a $\beta$-normal form.
- A term $L$ is strongly normalising ($\beta$-SN) if it has no infinite reduction sequence $L \rightarrow^1_\beta L_1 \rightarrow^1_\beta L_2 \rightarrow^1_\beta \cdots$
Untyped $\lambda$-Calculus: Examples

Example 9.12

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

$$(K (\omega \omega)) \rightarrow^{1}_{\beta} (K (\omega \omega)) \rightarrow^{1}_{\beta} (K (\omega \omega)) \rightarrow^{1}_{\beta} \cdots$$

or at some point it could transform to

$$(K (\omega \omega)) \rightarrow^{1}_{\beta} \lambda y[(\omega \omega)] \rightarrow^{1}_{\beta} \lambda y[(\omega \omega)] \rightarrow^{1}_{\beta} \cdots$$

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

$$( (K \omega) \Omega) \rightarrow^{1}_{\beta} ((K \omega) \Omega) \rightarrow^{1}_{\beta} \cdots$$
Examples of Strong Normalization

Example 9.13

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:

\[
((S K) K) \rightarrow^\beta_1 \lambda z[((K z)(K z))] \rightarrow^\beta_1 \lambda z[z] \equiv I
\]
10. Notions of Reduction
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:

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

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 10.1 \( \beta \)-equality or \( \beta \)-conversion (denoted \( =_\beta \)) is the smallest equivalence relation containing \( \beta \)-reduction (\( \to^*_\beta \)).

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.

\[
\begin{align*}
=\beta & \text{ Basis } \quad \frac{L \to^*_\beta M}{L =_\beta M} \\
=\beta & \text{ Reflexivity } \quad \frac{L =_\beta L}{L =_\beta L} \\
=\beta & \text{ Symmetry } \quad \frac{L =_\beta M}{M =_\beta L} \\
=\beta & \text{ Transitivity } \quad \frac{L =_\beta M, M =_\beta N}{L =_\beta N}
\end{align*}
\]
Example 10.2 Consider Curry’s paradoxical combinator

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

For any term \( L \) we have

\[
(Y_C \ L) \xrightarrow{\beta} (\lambda x[(L \ (x \ x))] \ \lambda x[(L \ (x \ x))]) \\
\equiv (\lambda y[(L \ (y \ y))] \ \lambda x[(L \ (x \ x))]) \\
\xrightarrow{\beta} (L \ (\lambda x[(L \ (x \ x))] \ \lambda x[(L \ (x \ x))])) \\
= (L \ (Y_C \ L))
\]

Hence \( (Y_C \ L) = (L \ (Y_C \ L)) \). However \( (L \ (Y_C \ L)) \) will never \( \beta \)-reduce to \( (Y_C \ L) \).
Compatibility of Beta-reduction and Beta-Equality

Theorem 10.3 $\beta$-reduction $\rightarrow^\ast_\beta$ and $\beta$-equality $=^\beta$ are both compatible relations.
Proof of theorem 10.3

Proof: \( \rightarrow^* \) 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).**

\[
\text{The proof holds for all } k, \ 0 \leq k \leq m \text{ for some } m \geq 0.
\]

**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 \( \rightarrow^n_\beta \)

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

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 $=_\beta \text{Symmetry}$ or $=_\beta \text{Transitivity}$ on some previous steps.

**Case ($=_\beta \text{Symmetry}$).** Then the $(n - 1)$-st step proved $M =_\beta L$. By the induction hypothesis and then by applying $=_\beta \text{Symmetry}$ to each case we get

$$
\begin{align*}
\text{for all variables } x, & \quad \lambda x[M] =_\beta \lambda x[L] \\
\text{for all terms } N, & \quad (M N) =_\beta (L N) \\
\text{for all terms } N, & \quad (N M) =_\beta (N L)
\end{align*}
$$

By $=_\beta \text{Symmetry}$

$$
\begin{align*}
\lambda x[L] =_\beta \lambda x[M] \\
(L N) =_\beta (M N) \\
(N M) =_\beta (N L)
\end{align*}
$$

**Case ($=_\beta \text{Transitivity}$).** 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 $=_\beta \text{Transitivity}$ we get

$$
\begin{align*}
\text{for all variables } x, & \quad \lambda x[L] =_\beta \lambda x[P], \quad \lambda x[P] =_\beta \lambda x[M] \\
\text{for all terms } N, & \quad (L N) =_\beta (P N), \quad (P N) =_\beta (M N) \\
\text{for all terms } N, & \quad (N L) =_\beta (N P), \quad (N P) =_\beta (N M)
\end{align*}
$$

By $=_\beta \text{Transitivity}$

$$
\begin{align*}
\lambda x[L] =_\beta \lambda x[M] \\
(L N) =_\beta (M N) \\
(N L) =_\beta (N P)
\end{align*}
$$

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,

$$\left(\lambda x[(M \, x)] \, N\right) =^\beta (M \, N)$$
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

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

(5)
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 10.1

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 (4) or (5). Now prove that $\rightarrow^1_{\beta\eta}$, $\rightarrow^*_{\beta\eta}$ and $=_{\beta\eta}$ are all compatible relations.

11. Confluence Definitions
Confluence
Reduction Relations

Definition 11.1 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 $\leftrightarrow^*$ denote respectively the compatible closure, the transitive closure, the reflexive transitive closure and the equivalence generated by $\rightarrow$. 
The Diamond Property

Definition 11.2 Let $\rho$ be any relation on terms. $\rho$ has the diamond property if for all $L, M, N$, 

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

Let $\rightarrow$ be a reduction relation, $\rightarrow^*$ the least preorder containing $\rightarrow$ and $\leftrightarrow^*$ the least equivalence relation containing $\rightarrow^*$. Then

**Definition 11.3** $\rightarrow$ is terminating iff there is no infinite sequence of the form

$$L_0 \rightarrow L_1 \rightarrow \cdots$$
Reduction: Local Confluence

Definition 11.4 \( \text{def:localconfluence} \rightarrow \) 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

\[
\begin{array}{ccc}
M & \rightarrow & \ast \\
\downarrow & & \downarrow \\
L & \Rightarrow & \exists P \\
\downarrow & & \downarrow \\
N & \rightarrow & \ast
\end{array}
\]
Reduction: Semi-confluence

Definition 11.5 $\rightarrow$ 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](attachment:image.png)
Reduction: Confluence

Definition 11.6 $\rightarrow$ is confluent if for all $L, M, N$,

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

which we denote as

\[
\begin{array}{c}
& M \\
\downarrow^* & \Downarrow^* \\
L & \Rightarrow \exists P \\
\downarrow^* & \Uparrow^* \\
N &
\end{array}
\]

Fact 11.7 Any confluent relation is also semi-confluent.
Reduction: Church-Rosser

Definition 11.8 $\rightarrow$ is Church-Rosser if for all $L, M$,

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

which we denote by

$$L \leftrightarrow^* M$$
Equivalence Characterization

Lemma 11.9

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

$$L \equiv_{\alpha} M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_m \equiv_{\alpha} M \quad (6)$$
Proof of lemma 11.9

Proof:

1. Just prove that $\leftrightarrow^*$ 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” (6) implies $L \leftrightarrow^* M$ by transitivity. For the other part use induction on the length of the proof.

QED

12. The Church-Rosser Property
Parallel Beta Reduction

**Definition 12.1** The parallel-$\beta$ or $\parallel\beta$ reduction is the smallest relation for which the following rules hold.

1. **$\parallel_1\beta$**
   
   $$
   \frac{L \rightarrow^1_\beta L'}{L \rightarrow^1_\beta L'}
   $$

2. **$\parallel_1\text{Abs1}$**
   
   $$
   \frac{L \rightarrow^1_\beta L'}{\lambda x[L] \rightarrow^1_\beta \lambda x[L']}
   $$

3. **$\parallel_1\text{App}$**
   
   $$
   \frac{L \rightarrow^1_\beta L', M \rightarrow^1_\beta M'}{(L M) \rightarrow^1_\beta (L' M')}
   $$

4. **$\parallel_1\text{Abs2}$**
   
   $$
   \frac{L \rightarrow^1_\beta L', M \rightarrow^1_\beta M'}{\lambda x[L] M \rightarrow^1_\beta \{M'/x\}}
   $$
Parallel Beta: The Diamond Property

**Lemma 12.2**

1. \( L \xrightarrow{\beta^1} L' \Rightarrow L \xrightarrow{1\beta} L' \).

2. \( L \xrightarrow{1\beta} L' \Rightarrow L \xrightarrow{\beta^*} L' \).

3. The smallest preorder containing \( \xrightarrow{1\beta} \) is \( \xrightarrow{\beta^*} = \xrightarrow{\beta^*} \).

4. If \( L \xrightarrow{\beta^1} L' \) 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. \( \text{QED} \)

**Theorem 12.3** \( \xrightarrow{1\beta} \) has the **diamond property**.
Proof of theorem 12.3

Proof: We need to prove for all \( L \)

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

We prove this by induction on the structure of \( L \) and a case analysis of the rule applied in definition 12.1.

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 \( L \equiv_{\alpha} M \). Choose \( P \equiv_{\alpha} N \) to complete the diamond.

Case \( L \equiv_{\alpha} N \). Then choose \( P \equiv_{\alpha} M \).

Case \( M \equiv_{\alpha} N \). Then there is nothing to prove.

In the sequel we assume \( N \not\equiv_{\alpha} L \not\equiv_{\alpha} M \not\equiv_{\alpha} N \) and proceed by induction on the structure of \( L \).

Case \( L \equiv \lambda x [L_1] \). Then clearly \( M \) and \( N \) were both obtained in proofs whose last step was an application of rule \( \xrightarrow{\beta} \nabla \text{Abs} \) 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 \( N_1 \xrightarrow{\beta} L_1 \xrightarrow{\beta} M_1 \). By the induction hypothesis we have

\[ \exists P_1 : N_1 \xrightarrow{\beta} P_1 \xrightarrow{\beta} M_1 \]

Hence by choosing \( P \equiv \lambda x [P_1] \) we obtain the desired result.

Case \( L \equiv (L_1 L_2) \) and \( L_1 \) is not an abstraction.

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

\[ \exists P_1 : N_1 \xrightarrow{\beta} P_1 \xrightarrow{\beta} M_1 \]

and

\[ \exists P_2 : N_2 \xrightarrow{\beta} P_2 \xrightarrow{\beta} M_2 \]

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

**Sub-case: Both $M$ and $N$ were obtained by applying rule $\beta_1 Abs2$.**

Then we have

$$\{N_2/x\}N_1 \equiv N_1 \stackrel{1}{\beta} L \equiv (\lambda x[L_1] L_2) \rightarrow_\beta^1 M \equiv \{M_2/x\}M_1$$

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

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

and

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

By the induction hypothesis

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

and

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

and the last part of lemma 12.2 we have

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

completing the proof.

**Sub-case: $M$ was obtained by applying rule $\beta_1 Abs2$ and $N$ by $\beta_1 App$.**

Then we have the form

$$(\lambda x[N_1] N_2) \equiv N_1 \stackrel{1}{\beta} L \equiv (\lambda x[L_1] L_2) \rightarrow_\beta^1 M \equiv \{M_2/x\}M_1$$

where again

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

and

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

By the induction hypothesis

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

and

$$\exists P_2 : N_2 \rightarrow_\beta^1 P_2 \stackrel{1}{\beta} M_2$$
and finally we have
\[ \exists P \equiv \{P_2/x\}P_1 : N \rightarrow^1_{||\beta} P \rightarrow^1_{||\beta} M \]
completing the proof.

Sub-case: \( M \) was obtained by applying rule \( ||\beta_1\text{App} \) and \( N \) by \( ||\beta_1\text{Abs2} \).
Similar to the previous sub-case.

QED

**Theorem 12.4** \( \rightarrow^1_{||\beta} \) is confluent.

**Proof:** We need to show that for all \( L, M, N, \)
\[ N \rightarrow^*_{||\beta} L \rightarrow^*_{||\beta} M \Rightarrow \exists P : N \rightarrow^*_{||\beta} P \rightarrow^*_{||\beta} M \]
We prove this by induction on the length of the sequences
\[ L \rightarrow^1_{||\beta} M_1 \rightarrow^1_{||\beta} M_2 \rightarrow^1_{||\beta} \cdots \rightarrow^1_{||\beta} M_m \equiv M \]
and
\[ L \rightarrow^1_{||\beta} N_1 \rightarrow^1_{||\beta} N_2 \rightarrow^1_{||\beta} \cdots \rightarrow^1_{||\beta} N_n \equiv N \]
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 12.3 to complete the rectangle

\[
\begin{array}{cccccc}
L & \rightarrow^1_{||\beta} & M_1 & \rightarrow^1_{||\beta} & M_2 & \rightarrow^1_{||\beta} \cdots \rightarrow^1_{||\beta} M_m & \equiv M \\
||\beta_1 & \downarrow & ||\beta_1 & \downarrow & ||\beta_1 & \downarrow & \cdots & ||\beta_1 \\
N_1 & \rightarrow^1_{||\beta} & P_{11} & \rightarrow^1_{||\beta} & P_{12} & \rightarrow^1_{||\beta} \cdots & \rightarrow^1_{||\beta} P_{1m} \\
||\beta_1 & \downarrow & ||\beta_1 & \downarrow & ||\beta_1 & \downarrow & \cdots & ||\beta_1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
||\beta_1 & \downarrow & ||\beta_1 & \downarrow & ||\beta_1 & \downarrow & \cdots & ||\beta_1 \\
N_n & \rightarrow^1_{||\beta} & P_{n1} & \rightarrow^1_{||\beta} & P_{n2} & \rightarrow^1_{||\beta} \cdots & \rightarrow^1_{||\beta} P_{nm} & \equiv P
\end{array}
\]

QED

**Corollary 12.5** \( \rightarrow^1_{\beta} \) is confluent.
Proof: Since $\rightarrow^*_{\beta} = \rightarrow^*_||\beta$ it follows from theorem 12.4 that $\rightarrow^1_{\beta}$ is confluent.

**Corollary 12.6** If a term reduces to a $\beta$-normal form then the normal form is unique (upto $\equiv_\alpha$).

Proof: If $N_1 \rightarrow^*_{\beta} N_2$ and both $N_1$ and $N_2$ are $\beta$-nfs, then by the corollary 12.5 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 12.7** $\rightarrow^1_{\beta}$ is Church-Rosser.

Proof: Follows from corollary 12.5 and theorem 13.2. QED

### 13. Confluence Characterizations
The Church-Rosser Property
Confluence and Church-Rosser

Lemma 13.1 *Every confluent relation is also semi-confluent*

Theorem 13.2 *The following statements are equivalent for any reduction relation $\rightarrow$.*

1. $\rightarrow$ is *Church-Rosser*.
2. $\rightarrow$ is *confluent*. 
**Proof of theorem 13.2**

*Proof: (1 ⇒ 2) Assume \(\rightarrow\) is Church-Rosser and let \(N \leftarrow^* M\)

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 ⇒ 1)* Assume \(\rightarrow\) is confluent and let \(L \leftarrow^* M\). We proceed by induction on the length of the chain (6).

\(L \equiv M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_m \equiv M\)

**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.

\(L \equiv M_0 \rightarrow / \leftarrow M_1 \rightarrow / \leftarrow \cdots \rightarrow / \leftarrow M_k \rightarrow / \leftarrow M_{k+1} \equiv M\)

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

\[
\begin{array}{c}
L \\ \leftarrow^* \\
\downarrow \quad \quad \quad \quad \quad \downarrow \\
M_k \\
\leftarrow^* \\
\leftarrow \quad \quad \quad \quad \quad \leftarrow \\
M \\
\leftarrow^* \\
\leftarrow \\
\leftarrow \quad \quad \quad \quad \quad \leftarrow \\
P \\
\end{array}
\]

which proves the claim.

**Case \(M_k \leftarrow M\).** Then the claim follows from the induction hypothesis and the following diagram

\[
\begin{array}{c}
L \\ \leftarrow^* \\
\downarrow \quad \quad \quad \quad \downarrow \\
M_k \leftarrow M \\
\leftarrow^* \\
\leftarrow \quad \quad \quad \quad \leftarrow \\
P \\
\end{array}
\]
Lemma 13.3 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 \( L \equiv_\alpha N \) then \( P \equiv_\alpha M \), 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

\[
\begin{array}{cccccc}
L & \rightarrow & N_1 & \rightarrow & \cdots & \rightarrow & N_n \equiv M \\
\downarrow & \downarrow & \downarrow & \cdots & \downarrow \\
M & \rightarrow & P_1 & \rightarrow & \cdots & \rightarrow & P_n
\end{array}
\]

QED

From lemma 13.3 and theorem 13.2 we have the following theorem.

Theorem 13.4 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. \( N_1 \leftarrow L \rightarrow^1 M_1 \Rightarrow \exists P : M_1 \rightarrow^* P \leftarrow N_1 \) (7)

We need to show for any term \( L \) that

\( N \leftarrow L \rightarrow^* M \Rightarrow \exists S : M \rightarrow^* S \leftarrow N \) (8)

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 13.5 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 13.6** 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 (8) holds.

**Induction Step.** Assume $L$ is a term whose longest reduction sequence is of length $p > 0$. Also assume $N \not\rightarrow^* L \rightarrow^* M$ i.e. $\exists m, n \geq 0 : N \not\rightarrow^m L \rightarrow^m M$.

- **Case $m = 0$.** If $m = 0$ then $M \equiv_\alpha L$ and hence $S \equiv_\alpha N$.
- **Case $n = 0$.** Then $N \equiv_\alpha L$ and we have $S \equiv_\alpha M$.
- **Case $m, n > 0$.** Then consider $M_1$ and $N_1$ such that

  $N \not\rightarrow^* N_1 \leftarrow^1 L \rightarrow^1 M_1 \rightarrow^* M$ \hspace{1cm} (9)

See figure (1). By (7), $\exists P : M_1 \rightarrow^* P \not\rightarrow N_1$. Clearly $M_1, N_1, P \in \Gamma(L) - \{L\}$. Hence by fact 13.6, $G(M_1), G(N_1)$ 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

$P \not\rightarrow^* M_1 \rightarrow^* M \Rightarrow \exists Q : M \rightarrow^* Q \not\rightarrow P \hspace{1cm} (10)$

and

$N \not\rightarrow^* N_1 \leftarrow^* P \Rightarrow \exists R : P \rightarrow^* R \not\rightarrow N \hspace{1cm} (11)$
But by (10) and (11) and the induction hypothesis we have

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

(12)

Combining (12) with (9), (10) and (11) we get

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

(13)

QED

**Theorem 13.7** If a terminating relation is locally confluent then it is Church-Rosser.
Proof: Follows from theorem 13.4 and theorem 13.2
14.1. 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:

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

• 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
• Notice that the constructor $(ITE \langle t, t_1, t_0 \rangle)$ is overloaded.
Extending the language

To make this simple language safe we require

**Type-checking**: to ensure that arbitrary expressions are not mixed in ways they are not “intended” to be used. For example

- $t$ cannot be a boolean expression in $S(t)$, $P(t)$, $IZ(t)$ 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.
FL0: 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:

$$L, M, N ::= t \in T_\Sigma(X) \mid \lambda x [L] \mid (L M)$$

(15)

This is two-level grammar combining the term grammar (14) with $\lambda$-abstraction and $\lambda$-application. While this makes it possible to use the operators of $T_\Sigma(X)$ as part of functions ($\lambda$-expressions), it does not allow us to use the operators of $T_\Sigma(X)$ outside of $\lambda$-abstractions and $\lambda$-applications.
FL0: Lack of Higher-order Power?

Example 14.1 The grammar (15) does not allow us to define expressions such as the following:

1. the successor of the result of an application \((S (L M))\)

2. higher order conditionals e.g. \(\lambda x[(\text{ITE} \langle (L x), (M x), (N x) \rangle)]\) where \((L x)\) yields a boolean value for an argument of the appropriate type.

3. In general, it does not allow the constructors to be applied to \(\lambda\)-expressions.
FL: Higher order functions

We need to *flatten* the grammar of (15) to allow \( \lambda \)-terms also to be used as arguments of the constructors of the term-grammar (14).

\[
L, M, N ::= x \in X \mid Z \\
\mid (P L) \mid (S L) \\
\mid T \mid F \\
\mid (IZ L) \mid (GTZ L) \\
\mid (ITE \langle L, M, N \rangle) \\
\mid \lambda x[L] \mid (L M)
\] (16)
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

\[
(P \ (S \ x)) \rightarrow_{\delta} x \\
(S \ (P \ x)) \rightarrow_{\delta} x
\]
Reduction Rules for Boolean Expressions

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

\[
(\text{ITE} \langle T, x, y \rangle) \rightarrow_\delta x \tag{19}
\]
\[
(\text{ITE} \langle F, x, y \rangle) \rightarrow_\delta y \tag{20}
\]

Testing for zero.

\[
(\text{IZ } Z) \rightarrow_\delta T \tag{21}
\]
\[
(\text{IZ } (S^k Z)) \rightarrow_\delta F, k > 0 \tag{22}
\]
\[
(\text{IZ } (P^k Z)) \rightarrow_\delta F, k > 0 \tag{23}
\]
Testing for Positivity

\[(\text{GTZ } Z) \rightarrow_{\delta} \text{ F} \quad (24)\]

\[(\text{GTZ } (S^k Z)) \rightarrow_{\delta} \text{ T}, k > 0 \quad (25)\]

\[(\text{GTZ } (P^k Z)) \rightarrow_{\delta} \text{ F}, k > 0 \quad (26)\]
Other Non-recursive Operators

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

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

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

\[
\begin{align*}
\text{GEZ} & \overset{df}{=} \lambda x[\text{OR} \langle (\text{IZ} x), (\text{GTZ} x) \rangle] \\
\text{LTZ} & \overset{df}{=} \lambda x[\text{NOT} (\text{GEZ} x)] \\
\text{LEZ} & \overset{df}{=} \lambda x[\text{OR} \langle (\text{IZ} x), (\text{LTZ} x) \rangle]
\end{align*}
\]
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 14.2** A combinator $Y$ is called a fixed-point combinator if for every lambda term $L$, $(Y L) = \beta (L (Y L))$

Curry’s $Y$ combinator ($Y_C$)

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

where

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

Turing’s $Y$ combinator ($Y_T$)

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

where

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

But the various $Y$ 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 14.2).

We extend the language FL with a new constructor

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

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

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

(27)
Recursion Example: Addition

Consider addition on integers as a binary operation to be defined in this language. We use the following properties of addition on the integers to define it by induction on the first argument.

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

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

Equation (29) may be rewritten using abstraction as follows:

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

We may think of equation (30) as an equation to be solved in the unknown variable $\text{plusc}$. Consider the (applied) $\lambda$-term obtained from the right-hand-side of equation (30) by simply abstracting the unknown $\text{plusc}$.

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

Claim 14.3

\[
(\text{REC} \; \text{adcc}) \longrightarrow_{\delta} (\text{adcc} \; (\text{REC} \; \text{adcc}))
\]  

and hence

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

Claim 14.4 $(\text{REC} \; \text{adcc})$ satisfies exactly the equation (30). That is

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

Hence we may regard $(\text{REC} \; \text{adcc})$ where $\text{adcc}$ is defined by right-hand-side of definition (31) as the required solution to the equation (29) in which $\text{plusc}$ is an unknown.
The abstraction shown in (31) and the claims (14.3) and (14.4) simply go to show that $M \equiv_{\alpha} \lambda f[\{f/z\}L]$ is a solution to the equation $z =_{\beta\delta} L$, whenever such a solution does exist. Further, the claims also show that we may “unfold” the recursion (on demand) by simply performing the substitution $\{L/z\}L$ for each free occurrence of $z$ within $L$. 
Exercise 14.1

1. Prove that the relation $\rightarrow_\delta$ is confluent.

2. The language FL does not have any operators that take boolean arguments and yields integer values. Define a standard conversion function $\text{B2I}$ 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 $\text{add}$ 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 $\text{mult}$ on integers and prove that it satisfies the equation(s) for products.

5. The equation (28) is defined conditionally. However the following is equally valid for all integer values $x$ and $y$.
   \[ x + y = (x - 1) + (y + 1) \] (35)
   
   (a) Follow the steps used in the construction of $\text{addc}$ to define a new applied $\text{addc}'$ that instead uses equation (35).
   
   (b) Is $(\text{REC} \text{addc}') =_{\beta\delta} (\text{addc}' (\text{REC} \text{addc}'))$?
   
   (c) Is $\text{addc} =_{\beta\delta} \text{addc}'$?
   
   (d) Is $(\text{REC} \text{addc}) =_{\beta\delta} (\text{REC} \text{addc}')$?

6. The function $\text{addc}$ 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.
Typing FL expressions

We have already seen that the simple language FL has

● two kinds of expressions: integer expressions and boolean expressions,

● there are also constructors which take integer expressions as arguments and yield boolean values

● there are also function types which allow various kinds of functions to be defined on boolean expressions and integer expressions.
The Need for typing in FL

- A type is an important attribute of any variable, constant or expression, since every such object can only be used in certain kinds of expressions.

- Besides the need for type-checking rules on $T_\Sigma(X)$ 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 the function operator on types.
- We require \( \ast \) because of the possibility of defining functions of various kinds of arities in \( \Lambda_X \).
- 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 14.5  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.

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

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

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

(Conditions 1. and 2.)

where

- **Condition 1.** $T_1 \cap T_2 = T_1 \cap TV ar(\tau) = T_2 \cap TV ar(\sigma) = \emptyset$

- **Condition 2.** $\mathcal{L}_a \not\in T_1 \cup T_2 \cup TV ar(\sigma) \cup TV ar(\tau) \cup TV ar(C_1) \cup TV ar(C_2)$.

- $T' = T_1 \cup T_2 \cup \{\mathcal{L}_a\}$

- $C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow \mathcal{L}_a\}$
Example 14.6 Consider the following simple combinator \( C \overset{\text{df}}{=} \lambda x[\lambda y[\lambda z[(x (y z))]]] \) 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. \( x : 'a, y : 'b, z : 'c \vdash x : 'a \doveq \emptyset \) \hspace{1cm} (Var)
2. \( x : 'a, y : 'b, z : 'c \vdash y : 'b \doveq \emptyset \) \hspace{1cm} (Var)
3. \( x : 'a, y : 'b, z : 'c \vdash z : 'c \doveq \emptyset \) \hspace{1cm} (Var)
4. \( x : 'a, y : 'b, z : 'c \vdash (y z) : 'd \doveq \{ 'd \} \{ 'b = 'c \rightarrow 'd \} \) \hspace{1cm} (App)
5. \( x : 'a, y : 'b, z : 'c \vdash (x (y z)) : 'e \doveq \{ 'd, 'e \} \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) \hspace{1cm} (App)
6. \( x : 'a, y : 'b \vdash \lambda z[(x (y z))] : 'c \rightarrow 'e \doveq \{ 'd, 'e \} \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) \hspace{1cm} (Abs)
7. \( x : 'a \vdash \lambda y[\lambda z[(x (y z))]] : 'b \rightarrow 'c \rightarrow 'e \doveq \{ 'd, 'e \} \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) \hspace{1cm} (Abs)
8. \( \vdash \lambda x[\lambda y[\lambda z[(x (y z))]]] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e \doveq \{ 'd, 'e \} \{ 'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e \} \) \hspace{1cm} (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 14.7 A solution for $\Gamma \vdash L : \tau \ni_T C$ 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 14.2

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 $\text{App}$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type Expression</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Sx$</td>
<td>$\Gamma \vdash t : \tau \triangleright_T C$</td>
<td>$\Gamma \vdash (S\ t) : \text{int} \triangleright_T C \cup {\tau = \text{int}}$</td>
</tr>
<tr>
<td>$Px$</td>
<td>$\Gamma \vdash t : \tau \triangleright_T C$</td>
<td>$\Gamma \vdash (P\ t) : \text{int} \triangleright_T C \cup {\tau = \text{int}}$</td>
</tr>
<tr>
<td>$IZx$</td>
<td>$\Gamma \vdash t : \tau \triangleright_T C$</td>
<td>$\Gamma \vdash (IZ\ t) : \text{bool} \triangleright_T C \cup {\tau = \text{int}}$</td>
</tr>
<tr>
<td>$GTZx$</td>
<td>$\Gamma \vdash t : \tau \triangleright_T C$</td>
<td>$\Gamma \vdash (GTZ\ t) : \text{bool} \triangleright_T C \cup {\tau = \text{int}}$</td>
</tr>
<tr>
<td>$ITEx$</td>
<td>$\Gamma \vdash s : \sigma \triangleright_T C$</td>
<td>$\Gamma \vdash t_1 : \tau_1 \triangleright_{T_1} C_1$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash t_0 : \upsilon \triangleright_{T_0} C_0$</td>
<td>$(T \cap T_1 = T_1 \cap T_0 = T_0 \cap T = \emptyset)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash (ITE\ \langle t, t_1, t_0 \rangle) : \tau \triangleright_{T'} C'$</td>
</tr>
</tbody>
</table>

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

2. Use the rules to define the type of the combinators $K$ and $S$?
3. How would you define a type assignment for the recursive function \( \text{addc} \) defined by equation (31).

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) \((\text{ITE } \langle \text{IZ } x, T, (K x) \rangle)\)
15. Logic Programming and Prolog

Prolog: Abstract Interpreter

**Algorithm 5** A simple abstract interpreter for Prolog

**Require:** A Prolog program $P$ and ground goal $G$

**Ensure:** yes if $P \vdash G$ else no

1: resolvent := $\{G\}$
2: while $\sim$ empty(resolvent) do
3:   Choose goal $A$ from resolvent
4:   Choose a *ground* instance of some clause $A' \leftarrow B_1, \ldots, B_k$ from $P$ such that $A \equiv A'$
5:   if $A'$ does not exist then
6:     exit loop
7:   end if
8:   resolvent := (resolvent $\setminus \{A\}$) $\cup \{B_1, \ldots, B_k\}$
9: end while
10: if empty(resolvent) then
11:    return yes
12: else
13:    return no
14: end if