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 24, 2019
## Contents

1 The Programming Languages Geneology 3

2 Introduction 14

3 Scanning or Lexical Analysis 32
   3.1 Regular Expressions ........................................ 42
   3.2 Nondeterministic Finite Automata (NFA) .................. 61
   3.3 Deterministic Finite Automata (DFA) ...................... 94

4 Parsing or Syntax Analysis 108
   4.1 Grammars .................................................... 108
   4.2 Context-Free Grammars .................................... 112
   4.3 Ambiguity .................................................. 127
   4.4 Shift-Reduce Parsing ...................................... 147
   4.5 Bottom-Up Parsing ....................................... 213
   4.6 Simple LR Parsing ....................................... 231
   4.7 Recursive Descent Parsing ............................... 273
   4.8 Specification of Syntax: Extended Backus-Naur Form . 279
      4.8.1 The Extended Backus-Naur Form (EBNF) .......... 280
5 Attributes & Semantic Analysis
5.1 Syntax-Directed Translation ................................................................. 306
  5.1.1 Synthesized Attributes ................................................................. 316
  5.1.2 Inherited Attributes ........................................................................ 335

6 Abstract Syntax ....................................................................................... 350

7 Symbol Table .......................................................................................... 364

8 Intermediate Representation .................................................................... 371

9 The Pure Untyped Lambda Calculus: Basics ........................................ 393

10 Notions of Reduction ............................................................................. 409

11 Logic Programming and Prolog ............................................................ 409
1. The Programming Languages Geneology
The Landscape of General PLs
The Usage of General PLs

The diagram illustrates the evolution of programming languages from 1955 to 2000, highlighting key milestones and categorizing them into different domains such as Scientific & Numerical Computations, Business Data Processing, Structured Programming, and Theorem Proving. It also identifies languages like Fortran, COBOL, Algol 60, Lisp, and APL, among others, and associates them with specific years and features, such as Symbol processing, Business Data Processing, and Systems Programming. The diagram emphasizes the progression from imperative to object-oriented and functional paradigms, showing how languages like C, C++, and Python have evolved over time.
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

SCANNER

stream of tokens
```
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

1. **SCANNER**
   - stream of characters
2. **PARSER**
   - stream of tokens
   - parse tree
3. **SEMANTIC ANALYZER**
   - abstract syntax tree
4. **I.R. CODE GENERATOR**
   - intermediate representation
5. **OPTIMIZER**
   - optimized intermediate representation
The Big Picture: 6

stream of characters

SCANNER

stream of tokens

PARSER

parse tree

SEMANTIC ANALYZER

abstract syntax tree

I.R. CODE GENERATOR

intermediate representation

OPTIMIZER

optimized intermediate representation

CODE GENERATOR

target code
The Big Picture: 7

SCANNER

PARSER

SEMANTIC ANALYZER

I.R. CODE GENERATOR

OPTIMIZER

CODE GENERATOR

SYMBOL TABLE MANAGER

ERROR-HANDLER

stream of characters

stream of tokens

parse tree

abstract syntax tree

intermediate representation

optimized intermediate representation

target code
The Big Picture: 8

Scanner
PARSER
SEMANTIC ANALYZER
I.R. CODE GENERATOR
OPTIMIZER
CODE GENERATOR

stream of characters
stream of tokens
parse tree
abstract syntax tree
intermediate representation
optimized intermediate representation
target code
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.
Scanning: 2

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

- **Whitespace**: A sequence of space, tab, newline, carriage-return, form-feed characters etc.

- **Lexeme**: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators or punctuation symbols)

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

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

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

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

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

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

  1. An identifier consists of a sequence of 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

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) $\mathbb{A}$.

- Any set of strings built up from the symbols of $\mathbb{A}$ 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 $\mathcal{L}(\text{Menelik}|\text{Selassie}|\epsilon) = \{ \text{Menelik, Selassie, """"} \}$. 
String Repetitions

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

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

Then

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

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

$$x^0 = ""$$
$$x^1 = \text{Selassie}$$
$$x^2 = \text{SelassieSelassie}$$
$$\vdots$$
$$x^5 = \text{SelassieSelassieSelassieSelassieSelassieSelassie}$$
$$\vdots$$

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

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

Then

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

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

\[
X^0 = \{\"\"\}\n\]
\[
X^1 = \{\text{Mengesha, Gemechis}\}
\]
\[
X^2 = \{\text{MengeshaMengesha, GemechisMengesha, MengeshaGemechis, GemechisGemechis}\}
\]
\[
X^3 = \{\text{MengeshaMengeshaMengesha, GemechisMengeshaMengesha, MengeshaGemechisMengesha, GemechisGemechisGemechis, MengeshaGemechisGemechis, GemechisGemechisGemechis, GemechisGemechisGemechisGemechis}\}
\]

\[
\vdots
\]
\[
X^{n+1} = X.X^n
\]

\[
\vdots
\]
Kleene Closure

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

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

In general

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

denotes an infinite union of languages.

Further it is easy to show the following identities.

$$r^* = \epsilon | r.r^*$$  \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.

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

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

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

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

**Question 8.** Give regular expressions to specify

- real numbers in *fixed decimal point notation*
- real numbers in *floating point notation*
- real numbers in both *fixed decimal point notation* as well as *floating point notation*.
Equivalence of Regular Expressions

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

- $r \leq_A r$ 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.(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 $a^{b^c}$. For $a = 2$, $b = 3$, $c = 4$ is this expression to be interpreted as $a^{(b^c)}$ or as $(a^b)^c$?

**Exercise 3.1**

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)** \( N \) 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
  \[
  \rightarrow: Q \times (A \cup \{\varepsilon\}) \rightarrow 2^Q
  \]

  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 *different* target states,
  - one or more transitions on the *same input* symbol from one source state to two or more different target states,
  - choice between executing a transition on an input symbol and a transition on $\varepsilon$ (and going to different states).
Acceptance of NFA

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

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

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

  where $q_n \in F$ is an accepting state.

- Since the automaton is nondeterministic, it is also possible that there exists another sequence of transitions

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

  where $q'_n$ is not a final state.

- The automaton accepts $x$, if there is an accepting sequence for $x$. 
Language of a NFA

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

- Initial state
- Accepting state
- Typical transition
- 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

\[ N_\varepsilon \]

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon_0 )</td>
<td>( \emptyset ) ( \emptyset ) ( \emptyset ) ( { \varepsilon_f } )</td>
</tr>
<tr>
<td>( \varepsilon_f )</td>
<td>( \emptyset ) ( \emptyset ) ( \emptyset ) ( \emptyset )</td>
</tr>
</tbody>
</table>
Regular Expressions to NFAs:3

\[ N_{r|s} \]

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r \mid s_0 )</td>
<td>( \emptyset ) ( \cdots ) ( { r_0, s_0 } )</td>
</tr>
<tr>
<td>( r_0 )</td>
<td>( \cdots ) ( \cdots ) ( \cdots )</td>
</tr>
<tr>
<td>( r_f )</td>
<td>( \cdots ) ( \cdots ) ( { r \mid s_f } )</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>( \cdots ) ( \cdots ) ( \cdots )</td>
</tr>
<tr>
<td>( s_f )</td>
<td>( \cdots ) ( \cdots ) ( { r \mid s_f } )</td>
</tr>
<tr>
<td>( r \mid s_f )</td>
<td>( \emptyset ) ( \cdots ) ( \emptyset )</td>
</tr>
</tbody>
</table>
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

\[ N_{r^*} \begin{array}{|c|c|c|} \hline \text{Input Symbol} & a & \cdots & \varepsilon \\ \hline \text{State} & r_0^* & \emptyset & \cdots & \{r_0, r_f^*\} \\ \hline r_0 & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ rf & \cdots & \cdots & \{r_0, r_f^*\} \\ \hline r_f^* & \emptyset & \emptyset & \emptyset \\ \hline \end{array} \]
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

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

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

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

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

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.
Given a set $T$ of states, then $T_\varepsilon = \varepsilon$-\textit{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.

\textbf{Algorithm 1 $\varepsilon$-CLOSE}

\textbf{Require:} $T$ a set of states of the NFA

\textbf{Ensure:} $T_\varepsilon = \varepsilon$-\textit{CLOSE}(T).

1. $U := T$
2. \textbf{repeat}
3. $U_{old} := U$
4. $U := U_{old} \cup \{q' \mid q' \not\in U, \exists q \in U_{old} : q \xrightarrow{\varepsilon} q'\}$
5. \textbf{until} $U = U_{old}$
6. $T_\varepsilon = U$
7. \textbf{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

\begin{align*}
S := \varepsilon\text{-CLOSURE}\{q_0\}.
\end{align*}

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

\begin{algorithmic}
\While{$a \neq \text{end\_of\_string}$}
\State $S := \varepsilon\text{-CLOSURE}(S \xrightarrow{a})$
\State $a := \text{nextchar}(x)$
\EndWhile
\end{algorithmic}

\State 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 \( D = \langle Q_D, A, s_D, F_D, \delta_D \rangle \) 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 $N = (Q_N, A \cup \{\varepsilon\}, s_N, F_N, \rightarrow_N)$
Ensure: DFA $D = (Q_D, A, s_D, F_D, \delta_D)$ 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($q_D \xrightarrow{a} N$) \{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 $q_D' \not\in Q_D$ 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

\[ N(a|b)_{aabb} \quad D(a|b)_{aabb} \]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| State      | Input Symbol | \(D_{(a|b)^*abb}\) |
|------------|--------------|---------------------|
| \(EC_0\)  | a            | \(EC_{4,8}\)       |
| \(EC_{4,8}\)| a            | \(EC_{4,8}\)       |
| \(EC_5\)  | a            | \(EC_{4,8}\)       |
| \(EC_{5,9}\)| a            | \(EC_{4,8}\)       |
| \(EC_{5,10}\)| a           | \(EC_{4,8}\)       |
| \(EC_{5,9}\)| b            | \(EC_{5,10}\)      |
| \(EC_{5,10}\)| b            | \(EC_{5,9}\)       |
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 \texttt{cmp} and \texttt{cmpdir}, state how you will include such commands also in your DFA where one command is a prefix of another.
4. Parsing or Syntax Analysis

4.1. Grammars

Parsing Or Syntax Analysis
Formal languages: Definition, Recognition, Generation

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

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

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

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

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

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

• $R$ may be finitely represented by a regular expression (even though the actual expression is very long).

• However, $P$ cannot actually be represented by a regular expression

• A regular expression is not powerful enough to represent languages which require parenthesis matching to arbitrary depths.

• All high level programming languages require an underlying language of expressions which require parentheses to be nested and matched to arbitrary depth.
4.2. Context-Free Grammars

Grammars

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

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

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

Note. Every sentence is also a sentential form.

$^a$some authors call it a word. However we will reserve the term word to denote the tokens of a programming language.
Grammars: Notation

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

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

\[ G = (\{S\}, \{a, b\}, P, S) \], 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 $\varepsilon$ must be shown explicitly in the tree.
CFG: Derivation trees 2

Derivation tree of $abaabb$
CFG: Derivation trees 3

Another Derivation tree of

$abaabb$
Yet another Derivation tree of $abaabb$
4.3. 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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

Right-most derivation of \( y + 4 \times z \) corresponding to the second derivation tree.

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

$$
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow I \mid C \\
I & \rightarrow y \mid z \\
C' & \rightarrow 4
\end{align*}
$$

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

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

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

Compare it with the Left-most Derivation 1.

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

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

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.

$$
E \Rightarrow \\
E+T \Rightarrow \\
E+T*F \Rightarrow \\
E+T*I \Rightarrow \\
E+T*z \Rightarrow \\
E+C*z \Rightarrow \\
E+4*z \Rightarrow \\
F+4*z \Rightarrow \\
I+4*z \Rightarrow \\
+4*z \Rightarrow \\
y+4*z 
$$

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

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

$I \rightarrow y \mid z$

$C \rightarrow 4$

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 $+$. **Associativity.** $\ast$ and $+$ are both left associative operators.
Exercise 4.1

1. Two context-free grammars are considered equivalent if they generate the same language. Prove that $G_1$ and $G'_1$ are equivalent.

2. Palindromes. A palindrome is a string that is equal to its reverse i.e. it is the same when read backwards (e.g. aabbbaa and ababaabaab are both palindromes). Design a grammar for generating all palindromes over the terminal symbols $a$ and $b$.

3. Matching brackets.

   (a) Design a context-free grammar to generate sequences of matching brackets when the set of terminals consists of three pairs of brackets $\{ (, [ , \{, \\} \}$.

   (b) If your grammar is ambiguous give two rightmost derivations of the same string and draw the two derivation trees. Explain how you will modify the grammar to make it unambiguous.

   (c) If your grammar is not ambiguous prove that it is not ambiguous.

4. Design an unambiguous grammar for the expression language on integers consisting of expressions made up of operators $+, -, *, /, \%$ and the bracketing symbols $(, )$, assuming the usual rules of precedence among operators that you have learned in school.

5. Modify the above grammar to include the exponentiation operator $^\text{\(\)}$ 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 $\text{\(-\)}$ 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 \( X \).
Fully Bracketed Expression

Consider an example of a fully bracketed expression
Parsing: FB0

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

\[( a \rightarrow ( a \ / \ b ) ) \] Shift
Parsing: FB1

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

(a (a / b))
Parsing: FB2

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

r2. \( E \rightarrow T \)

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

r4. \( T \rightarrow D \)

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

\[ \rightarrow (a \lor b) \]

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Shift
Principle:
Reduce whenever possible.
Shift only when reduce is impossible.

Reduce
**Parsing: FB4**

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

**Principle:**
Reduce whenever possible. Shift only when reduce is impossible.

Reduce
Parsing: FB5

Principle:
Reduce whenever possible. Shift only when reduce is impossible

Reduce
Parsing: FB6

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

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

\[= (\text{a} / \text{b})()\]
Parsing: FB7

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

- E → E T
- E → T
- T → T D
- T → D
- D → a | b | ( E )

Shift

(( a / b ) )
Parsing: FB8

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

E → E T
E → T
T → T D
T → D
D → a | b | ( E )

Reduce
Parsing: FB9

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \cdot D$

r4. $T \rightarrow D$

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce
Parsing: FB10

r1. $E \rightarrow E \quad T$

r2  $E \rightarrow T$

r3  $T \rightarrow T / D$

r4  $T \rightarrow D$

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce?
Parsing: FB11

Principle:
Reduce whenever possible. Shift only when reduce is impossible.

Reduce? Shift

r1. $E \rightarrow E \ E$ $T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

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

/$ ) b ) $

( / $

$E$

$T$

$()$

$-\$

$E$

$(())$
Parsing: FB12

Principle:
Reduce whenever possible.
Shift only when reduce is impossible.
Parsing: FB13

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible
Parsing: FB14

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

\[
\begin{align*}
&\text{r1. } E \rightarrow E \rightarrow T \\
&\text{r2. } E \rightarrow T \\
&\text{r3. } T \rightarrow T \rightarrow D \\
&\text{r4. } T \rightarrow D \\
&\text{r5. } D \rightarrow a \ | \ b \ | \ ( \ E )
\end{align*}
\]
Parsing: FB15

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce?
Parsing: FB16

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

No, REDUCE!
Parsing: FB17

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce?
Parsing: FB18

Principle:
Reduce whenever possible.
Shift only when reduce is impossible.

E → E T
E → T
T → T D
T → D
D → a | b | ( E )

E
( ( )

Shift
Parsing: FB19

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce
Parsing: FB20

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce
Parsing: FB21

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce?
Parsing: FB22

Principle:
Reduce whenever possible.
Shift only when reduce is impossible.

No, REDUCE!
Parsing: FB23

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

E
( )

Shift
Principle:
Reduce whenever possible. Shift only when reduce is impossible.
Parsing: FB25

Principle:
Reduce whenever possible.
Shift only when reduce is impossible
Parsing: FB26

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

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

Reduce
Parsing: FB27

Principle:
Reduce whenever possible.
Shift only when reduce is impossible

E → E T
E → T
T → D
T → ( D
D → a | b | ( E )

Reduce
Unbracketed Expression

Consider an example of an unbracketed expression which relies on the precedence rules as defined in the grammar.
Parsing: UB0

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

a → a / b
**Parsing: UB1**

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

**Principle:**
Reduce whenever possible.
Shift only when reduce is impossible.

**Shift**
Parsing: UB2

r1. $E \rightarrow E\ T$

r2. $E \rightarrow T$

r3. $T \rightarrow T\ D$

r4. $T \rightarrow D$

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

Reduce by r5
Parsing: UB3

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

Reduce by r4
Parsing: UB4

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

Reduce by r2
Parsing: UB5

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

Shift
Parsing: UB6

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

r2. \( E \rightarrow T \)

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

r4. \( T \rightarrow D \)

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

Shift
Parsing: UB7

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

Reduce by r5
Parsing: UB8

r1. E → E T
r2. E → T
r3. T → T 0 D
r4. T → D
r5. D → a | b | ( E )

Reduce by r4
Parsing: UB8a

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: UB9a

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: UB10a

r1. \( E \rightarrow E \rightarrow T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \rightarrow D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid \( E \) \)}
Parsing: UB11a

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

r2. \( E \rightarrow T \)

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

r4. \( T \rightarrow D \)

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

Shift
Parsing: UB12a

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

Reduce by r5
Parsing: UB13a

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

Reduce by r4
Parsing: UB14a

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

Stuck!

Get back!

Reduce by r2
Parsing: UB14b

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

Get back!

Reduce by r2
Parsing: UB13b

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

Get back!
Reduce by r4
Parsing: UB12b

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

Get back!

Reduce by r5
Parsing: UB11b

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

Get back! Shift
Parsing: UB10b

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: UB9b

Get back to where you once belonged!

Reduce by r1
Parsing: UB8b

Principle:
Reduce whenever possible, but depending upon lookahead.

Shift instead of reduce here!

Shift-reduce conflict

Reduce by r4
Parsing: UB8

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: UB9

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

Shift

b

( // T
// E
a
Parsing: UB10

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

Shift
Parsing: UB11

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

Reduce by r5
Parsing: UB12

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

Reduce by r3
Parsing: UB13

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

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)

shift-reduce parsing: 0
Parse Trees: 1

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:1
Parse Trees: 2

r1. E → E - T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )
Parse Trees: 3

r1. $E \rightarrow E - T$

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. \( E \rightarrow E \rightarrow T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \rightarrow D \)

r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

shift-reduce parsing
Parse Trees: 3b

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

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

shift-reduce parsing
Parse Trees: 4

r1. \( E \rightarrow E \mathbin{-} T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \mathbin{(|)} D \)

r4. \( T \rightarrow D \)

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

shift-reduce parsing
Parse Trees: 5

shift-reduce parsing
Parse Trees: 5a

```
   E  →  E  −  T
  r1

   E  →  T
  r2

   T  →  T  /  D
  r3

   T  →  D
  r4

   T  →  a  |  b  |  (E)
  r5

shift-reduce parsing
```
Parse Trees: 5b

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

shift-reduce parsing
Parse Trees: 6

r1. $E \rightarrow E + T$

r2 $E \rightarrow T$

r3 $T \rightarrow T / D$

r4 $T \rightarrow D$

r5 $D \rightarrow a | b | (E)$
Parse Trees: 7

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: 8

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
Parsing: Summary: 1

• All high-level languages are designed so that they may be parsed in this fashion with only a single token look-ahead.

• Parsers for a language can be automatically constructed by parser-generators such as Yacc, Bison, ML-Yacc and CUP in the case of Java.

• Shift-reduce conflicts if any, are automatically detected and reported by the parser-generator.

• Shift-reduce conflicts may be avoided by suitably redesigning the context-free grammar.
Parsing: Summary: 2

- Very often shift-reduce conflicts may occur because of the prefix problem. In such cases many parser-generators resolve the conflict in favour of shifting.

- There is also a possibility of reduce-reduce conflicts. This usually happens when there is more than one nonterminal symbol to which the contents of the stack may reduce.

- A minor reworking of the grammar to avoid redundant non-terminal symbols will get rid of reduce-reduce conflicts.

The Big Picture
4.6. Simple LR Parsing

Parsing Problems 1

The main question in shift-reduce parsing is:

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

To answer this question we require

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

But the decision has to be automatic. So the parser requires some rules. Once given the rules we may construct the parser to follow the rules.
Parsing Problems 2

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

The next question:

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

Disregarding the very next input token as always available, the length of the extra amount of input required for a shift-reduce decision is called the lookahead.
Parsing Problems 3

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

in case of a valid sentence that it should only apply reduction rules and at-

tempt to reach the start symbol of the grammar only through reductions

and

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

parsing process

To solve this problem we augment every grammar with a new start symbol

$S$ and a new terminal token $\$ and augment the grammar with a new

special rule. For our previous grammar we have the new rule

\[ S \rightarrow E\$ \]
Augmented Grammar

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

1. $S \rightarrow E$
2. $E \rightarrow E − T$
3. $E \rightarrow T$
4. $T \rightarrow a$
5. $T \rightarrow (E)$
LR(0) Languages

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

- “L” refers to reading the input from left to right,
- “R” refers to the (reverse) of rightmost derivation
- “0” refers to no-lookahead..

- Many simple CFLs are LR(0). But the LR(0) parsing method is too weak for most high-level programming languages.
- But understanding the LR(0) parsing method is most crucial for understanding other more powerful LR-parsing methods which require lookaheads for deterministic shift-reduce decision-making
LR-Parsing Invariant

In any LR-parsing technique the following invariant holds.

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

Hence given at any stage of the parsing if \( \alpha \in (N \cup T)^* \) is the contents of the parsing stack and \( x \in T^* \) is the rest of the input that has not yet been read, then \( \alpha x \) is a sentential form of a right-most derivation.
LR(0) Item

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

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

  where $\alpha = \beta \gamma$.
- For each rule $X \rightarrow \alpha$, there are $|\alpha| + 1$ distinct LR(0) items – one for each position in $\alpha$. 
What does an LR(0) item signify?

The LR(0) item

\[ X \rightarrow \beta \uparrow \gamma \]

signifies that at some stage of parsing

- \( \beta \) is the string (of terminals and nonterminals) on the top of the stack and
- some prefix of the rest of the input can be generated by \( \gamma \) so that whenever \( \beta \gamma \) appears on the stack, \( \beta \gamma \) may be reduced immediately to \( X \).
LR0 Parsing Strategy

The LR0 parsing strategy is to

1. construct a DFA whose alphabet is $N \cup T \cup \{\$\}$

2. use the parsing stack to perform reductions at appropriate points

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

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

- **reduce** $j$ a reduction using the production rule $j$ is performed

- **goto** $k$ Based on the contents of the stack, the DFA moves to state $k$. 
Favourite Example

Consider our favourite augmented grammar

1. $S \rightarrow E$
2. $E \rightarrow E - T$
3. $E \rightarrow T$
4. $T \rightarrow a$
5. $T \rightarrow (E)$
Rule 1: Items

Rule 1

\[ R1. \ S \to E\$

has the following three items

\[ I1.1 \ S \to \^{\uparrow}E\$
\[ I1.2 \ S \to E\^{\uparrow}\$
\[ I1.3 \ S \to E\$^{\uparrow}\$

one for each position on the right hand side of the rule.
Rule 2: Items

Rule 2

\[ R2. \ E \rightarrow \ E - T \]

has the following items

\[ I2.1 \ E \rightarrow \ E\uparrow - T \]
\[ I2.2 \ E \rightarrow \ E \downarrow - T \]
\[ I2.3 \ E \rightarrow \ E - \uparrow T \]
\[ I2.4 \ E \rightarrow \ E - T \uparrow \]
Rule 3: Items

Rule 3

has just the items

\[ R3. \ E \rightarrow T \]

\[ I3.1 \ E \rightarrow \uparrow T \]

\[ I3.2 \ E \rightarrow T \uparrow \]
Rule 4: Items

Rule 4

has the items

\[ R4. \ T \rightarrow a \]

\[ I4.1 \ T \rightarrow \uparrow a \]

\[ I4.2 \ T \rightarrow a \uparrow \]
Rule 5: Items

Rule 5

\[ R5. \ T \rightarrow (E) \]

has the items

\[ I5.1 \ T \rightarrow \ ^{\wedge}(E) \]
\[ I5.2 \ T \rightarrow (E^{\wedge}) \]
\[ I5.3 \ T \rightarrow (E^{\wedge}) \]
\[ I5.4 \ T \rightarrow (E^{\wedge}) \]
Significance of I1.*

I1.1 $S \rightarrow \text{ } E \$. Hence

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

I1.2 $S \rightarrow E \$. Hence

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

I1.3 $S \rightarrow E \$. Hence

1. There is no input left to be read and
2. the stack contents may be reduced to the start symbol
DFA States: Initial and Final

- Clearly the *initial* state $S_1$ of the DFA will correspond to *item l1.1*.
- There should be a state corresponding to *item l1.2*.
- There should be a *goto* transition on the nonterminal symbol $E$ from the *initial state* (corresponding to *item l1.1*) to the state corresponding to *item l1.2*.
- The *accepting* state of the DFA will correspond to *item l1.3*.
- There would also be a *shift* transition on $\$ from the state corresponding to *item l1.2* to the accepting state corresponding to *item l1.3*.
- There should be a *reduce* action using *rule 1* when the DFA reaches the state corresponding to *item l1.3*.
Input Possibilities

Consider item I1.1.

1. How will a grammatically valid sentence input reduce to $E\$ $? From the grammar it is obvious that this can happen only if the input is of a form such that

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

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

(a) If the entire input consists of only $a$ then it could be reduced to $T$ or
(b) If the entire input could be reduced to the form $(E)$ then it could be reduced to $T$.

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

(a) If the entire input could be split into 3 parts $\alpha$, $\beta$ and $\gamma$ such that
   i. $\alpha$ is a prefix that can be reduced to $E$, and
   ii. $\beta = \epsilon$, and
   iii. $\gamma$ is a suffix that can be reduced to $T$
   then it could be reduced to $E - T$
Closures of Items

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

Algorithm 4 Closures of Items

Require: Set $\mathcal{I}$ of LR(0) items of a CFG with rule set $P$
Ensure: Closure of $I$ for a subset $I \subseteq \mathcal{I}$ of items
1. repeat
2. for all $A \rightarrow \alpha \uparrow X \beta \in I$ do
3. for all $X \rightarrow \gamma \in P$ do
4. $I := I \cup \{X \rightarrow \uparrow \gamma\}$
5. end for
6. end for
7. until no more changes occur in $I$
State Changes on Nonterminals

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

Algorithm 5 Goto for a set of $I$ of items

Require: $I \subseteq \mathcal{T}$ and $X \in N$

Ensure: States of the DFA

1. $J := \emptyset$
2. for all $A \rightarrow \alpha X \beta \in I$ do
3. \hspace{1em} $J := J \cup \{A \rightarrow \alpha X \beta\}$
4. end for
5. return $\text{CLOSURE}(J)$
State S1

\[ S1 = \text{Closure}\{S \rightarrow \uparrow E$\}\] 
\[ = \{S \rightarrow \uparrow E$, $E \rightarrow \uparrow E-T$, $E \rightarrow \uparrow T$, 
\[ T \rightarrow \uparrow a$, $T \rightarrow \uparrow (E)\}\] 

\[ S1 \xrightarrow{E} \text{Closure}(\{S \rightarrow E \uparrow $, $E \rightarrow E \uparrow -T\}) = S3 \]

\[ S1 \xrightarrow{T} \text{Closure}(\{E \rightarrow T \uparrow\}) = S7 \]

\[ S1 \xrightarrow{a} \text{Closure}(\{T \rightarrow a \uparrow\}) = S8 \]
State S2

\[ S^2 = \text{Closure}\left(\{T \rightarrow (\triangle E)\}\right) \]
\[ = \{T \rightarrow \triangle(E), E \rightarrow \triangle E - T, E \rightarrow \triangle T, \]
\[ T \rightarrow \triangle a, T \rightarrow \triangle(E)\}\]

\[ S^2 \xrightarrow{E} \text{Closure}\left(\{T \rightarrow (E \triangle)\}\right) = S^9 \]

\[ S^2 \xrightarrow{T} \text{Closure}\left(\{E \rightarrow T \triangle\}\right) = S^7 \]

\[ S^2 \xrightarrow{a} \text{Closure}\left(\{T \rightarrow a \triangle\}\right) = S^8 \]
Other States

\[ S^3 = \text{Closure}(\{S \to E \triangleright \$, E \to E \triangleright -T\}) \]
\[ = \{S \to E \triangleright $, E \to E \triangleright -T\} \]

However,

\[ S^3 \triangleright \text{Closure}(\{E \to E \triangleright -T\}) \]

and

\[ \text{Closure}(\{E \to E \triangleright -T\}) \]
\[ = \{E \to E \triangleright -T, T \to (\triangleright E), T \to \triangleright a\} \]
\[ = S^4 \]

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

- \[ S^5 = \{E \to E \triangleright -T \}$ \]
- \[ S^6 = \{S \to E $ \}$ \]
- \[ S^7 = \{E \to T \}$ \]
- \[ S^8 = \{T \to a \}$ \]
- \[ S^9 = \{T \to (E \)$, E \to E \triangleright -T\} \]
- \[ S^{10} = \{T \to (E) \}$ \]
Example: DFA

Parsing Table

Diagram showing the DFA with states S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, and transitions for E, T, $, and a.
DFA

S1 a$

Shift S8
DFA

S1 | a$  

S1 | a S8 | $  

Reduce Rule 4
<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>a</td>
<td>Reduce Rule 4</td>
</tr>
<tr>
<td>S1</td>
<td>S8</td>
<td>Shift S8</td>
</tr>
<tr>
<td>S1</td>
<td>T</td>
<td>Goto S7</td>
</tr>
</tbody>
</table>
DFA Parsing Table

S1 a S8 $  
Reduce Rule 4

S1 T $  
Goto S7

S1 T S7 $  
Reduce Rule 3

S1 E $  
Goto S3
<table>
<thead>
<tr>
<th>DFA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>a S8</td>
</tr>
<tr>
<td>S1</td>
<td>a S8</td>
</tr>
<tr>
<td>S1</td>
<td>T $</td>
</tr>
<tr>
<td>S1</td>
<td>T S7</td>
</tr>
<tr>
<td>S1</td>
<td>E $</td>
</tr>
<tr>
<td>S1</td>
<td>E S3</td>
</tr>
</tbody>
</table>

**DFA Parsing Table**

<table>
<thead>
<tr>
<th>S1</th>
<th>a $</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>a S8</td>
</tr>
<tr>
<td>S1</td>
<td>T $</td>
</tr>
<tr>
<td>S1</td>
<td>T S7</td>
</tr>
<tr>
<td>S1</td>
<td>E $</td>
</tr>
<tr>
<td>S1</td>
<td>E S3</td>
</tr>
</tbody>
</table>

**Reduce Rule 4**

**Goto S7**

**Reduce Rule 3**

**Goto S3**

**Accept**
DFA

<table>
<thead>
<tr>
<th>S1</th>
<th>a $</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 a</td>
<td>S8</td>
</tr>
<tr>
<td>S1 T</td>
<td>$</td>
</tr>
<tr>
<td>S1 T</td>
<td>S7</td>
</tr>
<tr>
<td>S1 E</td>
<td>$</td>
</tr>
<tr>
<td>S1 E</td>
<td>S3</td>
</tr>
</tbody>
</table>

DFA Parsing Table

<table>
<thead>
<tr>
<th>S1</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a $(a-a)$</td>
<td>S8</td>
</tr>
</tbody>
</table>

Reduce Rule 4
Goto S7
Reduce Rule 3
Goto S3
Accept
Shift S8
DFA

\[
\begin{array}{c}
S1 & a & S8 & \\
S1 & T & \\
S1 & T & S7 & \\
S1 & E & \\
S1 & E & S3 & \\
S1 & a - (a - a) & S8 & \\
S1 & a & S8 & -(a - a) & \\
\end{array}
\]

Reduce Rule 4
Goto S3
Goto S7
Reduce Rule 3
Accept

Shift S8
DFA

\[
\begin{array}{c|c}
S1 & aS8 \\
\end{array}
\]

Reduce Rule 4

\[
\begin{array}{c|c}
S1 & TS7 \\
\end{array}
\]

Reduce Rule 3

\[
\begin{array}{c|c}
S1 & ES3 \\
\end{array}
\]

Accept

\[
\begin{array}{c|c}
S1 & a \\
\end{array}
\]

Shift S8

\[
\begin{array}{c|c}
S1 & aS8 \\
\end{array}
\]

Reduce Rule 4

\[
\begin{array}{c|c}
S1 & T \\
\end{array}
\]

Goto S7

\[
\begin{array}{c|c}
S1 & E \\
\end{array}
\]

Goto S3

\[
\begin{array}{c|c}
S1 & E \\
\end{array}
\]

Goto S3
$S1 \quad -(a - a)$
\[ S1 \ T - (a - a) \]

\[ S1 \ T S7 - (a - a) \]
S1 T $ - (a - a) $

S1 T S7 $ - (a - a) $

S1 E $ - (a - a) $

Go to S3

Reduce Rule 3

Go to S3

DFA Parsing Table
S1 T \[-(a - a)\] Go to S7

S1 T S7 \[-(a - a)\] Reduce Rule 3

S1 E \[-(a - a)\] Go to S3

S1 E S3 \[-(a - a)\] DFA Parsing Table Shift S4

DFA Parsing Table Shift S4
\( (a - a) \)

Reduce Rule 3

Go to S3

Go to S7

DFA

Shift S4

Shift S2
\[
S1 \quad T \quad -(a - a)\\
S1 \quad T \quad S7 \quad -(a - a)\\
S1 \quad E \quad -(a - a)\\
\]

DFA

\[
S1 \quad E \quad S3 \quad -(a - a)\\
S1 \quad E \quad S3 \quad - S4 \quad (a - a)\\
S1 \quad E \quad S3 \quad - S4 \quad (S2 \quad a - a)\\
\]

Reduce Rule 3
Go to S7
Go to S3
 Parsing Table
Shift S4
Shift S2
Shift S8
$S1 \rightarrow (a - a)\$$

$S1 \rightarrow (a - a)\$ 

$S1 \rightarrow (a - a)\$$

DFA

$S1 \rightarrow (a - a)\$$

$S1 \rightarrow (a - a)\$$

$S1 \rightarrow (a - a)\$$

Reduce Rule 3

Go to S3

Go to S3

Go to S3

Reduce Rule 4

Go to S3

Shift S4

Reduce Rule 3

Shift S2

Shift S8

Reduce Rule 4
\( S1 \ T \ (a - a)S \)

\( S1 \ T \ S7 \ (a - a)S \)

\( S1 \ E \ (a - a)S \)

\( S1 \ E \ S3 \ (a - a)S \)

\( S1 \ E \ S3 \ S4 \ (a - a)S \)

\( S1 \ E \ S3 \ S4 \ (S2 \ a \ S8 \ a - a)S \)

\( S1 \ E \ S3 \ S4 \ (S2 \ T \ S7 \ a - a)S \)

\( S1 \ E \ S3 \ S4 \ (S2 \ T \ S7 \ a - a)S \)

Go to S3

Reduce Rule 3

Go to S7

Reduce Rule 3

Parsing Table

Shift S4

Shift S2

Shift S8

Reduce Rule 4

Parsing Table

Reduce Rule 3
\[
S1 \text{T } - (a - a) \\
S1 \text{T S7 } - (a - a) \\
S1 \text{E } - (a - a) \\
\text{DFA} \\
S1 \text{E S3 } - (a - a) \\
\text{S1 E S3 } - \text{ S4 } (a - a) \\
\text{S1 E S3 } - \text{ S4 } (\text{ S2 } a - a) \\
\text{S1 E S3 } - \text{ S4 } (\text{ S2 a S8 } - a) \\
\text{S1 E S3 } - \text{ S4 } (\text{ S2 T S7 } - a) \\
\text{DFA} \\
\text{Reduce Rule 3} \\
\text{Reduce Rule 3} \\
\text{Go to S3} \\
\text{Go to S7} \\
\text{Reduce Rule 4} \\
\text{Shift S4} \\
\text{Shift S2} \\
\text{Shift S8}
S1 E S3 - S4 ( S2 E -a )$
S1 E S3 - S4 (S2 E S9) -a)g

S1 E S3 - S4 (S2 E S9) -a)g
Go to S9
Shift S4
Shift S8
Reduce Rule 4
S1 E S3 − S4 ( S2 E S9 − S4 ) a)$

DFA

S1 E S3 − S4 ( S2 E S9 − S4 T )$
<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Next State(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>E</td>
<td>S3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4 (S2 $E$ S9 $a$ S8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S6</td>
</tr>
</tbody>
</table>

**DFA Parsing Table**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Next State(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>E</td>
<td>S3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4 (S2 $E$ S9 $a$ S8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S6</td>
</tr>
</tbody>
</table>

**Shift S4**

**Shift S8**

**Reduce Rule 4**

**Reduce Rule 2**
**DFA Parsing Table**

1. **S1 E S3 − S4**
   - **Go to S9**

2. **S1 E S3 − S4**
   - **Shift S4**

3. **S1 E S3 − S4**
   - **Shift S8**

4. **S1 E S3 − S4**
   - **Reduce Rule 4**

5. **S1 E S3 − S4**
   - **Go to S5**

6. **S1 E S3 − S4**
   - **Reduce Rule 2**

7. **S1 E S3 − S4**
   - **Go to S9**
<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>a S8</td>
<td>Reduce Rule 4</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 2</td>
</tr>
<tr>
<td>S1</td>
<td>T S5</td>
<td>Reduce Rule 2</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 5</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 5</td>
</tr>
</tbody>
</table>

DFA Parsing Table:

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>a S8</td>
<td>Reduce Rule 4</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 2</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 5</td>
</tr>
<tr>
<td>S1</td>
<td>S9</td>
<td>Reduce Rule 5</td>
</tr>
</tbody>
</table>

DFA:

- Go to S5
- Reduce Rule 4
- Go to S9
- Shift S10
- Reduce Rule 5

DFA Parsing Table:
Reduce Rule 2
S1 E S3 – S4 T $ 

S1 E S3 – S4 T S5 $ 

S1 E $ 

Go to S3

Reduce Rule 2

Go to S3
Exercise 4.2

1. Design a LR(0) parser for the grammar of palindromes. Identify whether there are any conflicts in the parsing table.

2. Design a LR(0) parser for the grammar of Matching brackets and identify any conflicts.

3. Design a context-free grammar for a language on the terminal symbols $a$ and $b$ such that every string has more $a$s than $b$s. Design a LR(0) parser for this grammar and find all the conflicts, if any.

4. Since every regular expression may also be represented by a context-free grammar design an LR(0) parser for comments in C.
CFG = RLG + Bracket Matching

We use the idea that a context-free grammar is essentially a regular grammar with parentheses matching to arbitrary depths. Hence a DFA with some reductions introduced may work. We modify the grammar to have a special terminal symbol called the end-marker (denoted by $\$). Now consider the following simple grammar with a single right-associative binary operator $^\wedge$ and bracket-matching. We create a DFA of “items” which also have a special marker called the “cursor” (▲).
LR(0) with Right-Association

Consider the following grammar

1. \( S \rightarrow E\$
2. \( E \rightarrow P \hat{E} \)
3. \( E \rightarrow P \)
4. \( P \rightarrow a \)
5. \( P \rightarrow (E) \)

The following items make up the initial state \( S_1 \) of the DFA

\[
I1.1 \quad S \rightarrow \uparrow E$
\]
\[
I2.1 \quad E \rightarrow \uparrow P \hat{E} 
\]
\[
I3.1 \quad E \rightarrow \uparrow P 
\]
\[
I4.1 \quad P \rightarrow \uparrow a 
\]
\[
I5.1 \quad P \rightarrow \uparrow (E) 
\]
Shift-Reduce Conflicts in LR(0)

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

$I2.2 \ E \rightarrow \ P \uparrow \hat{E}$
$I3.2 \ E \rightarrow \ P \uparrow$

Then clearly the LR(0) parser suffers a shift-reduce conflict because
- item $I2.2$ indicates a shift action,
- item $I3.2$ produces a reduce action.

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

The parsing table in this case would have a shift action if the input in state $S2$ is a $\hat{}$ and a reduce action for all other input symbols.
FOLLOW Sets

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

\[
\text{FOLLOW}(E) = \{\$, )\}
\]
\[
\text{FOLLOW}(P) = \{^\}
\]

Depending upon the input symbol and whether it appears in the FOLLOW set of the non-terminal under question we resolve the shift-reduce conflict.

This modification to LR(0) is called Simple LR (SLR) parsing method. However SLR is not powerful enough for many useful grammar constructions that are encountered in many programming languages.
Computing FIRST Sets

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

1. FIRST \((a) = \{a\}\) for every terminal symbol \(a\).

2. \(\varepsilon \in \text{FIRST}(X)\) if \(X \rightarrow \varepsilon \in P\).

3. If \(X \rightarrow Y_1Y_2 \cdots Y_k \in P\) then \(\text{FIRST}(Y_1) \subseteq \text{FIRST}(X)\)

4. If \(X \rightarrow Y_1Y_2 \cdots Y_k \in P\) then for each \(i : i < k\) such that \(Y_1Y_2 \cdots Y_i \Rightarrow \varepsilon\), \(\text{FIRST}(Y_{i+1}) \subseteq \text{FIRST}(X)\).
Computing FOLLOW Sets

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

1. $\in \text{FOLLOW}(S)$ where $S$ is the start symbol of the augmented grammar.

2. For each production rule of the form $A \rightarrow \alpha B \beta$, $\text{FIRST}(\beta) = \{\epsilon\} \subseteq \text{FOLLOW}(B)$.

3. For each production rule of the form $A \rightarrow \alpha B \beta$, if $\epsilon \in \text{FIRST}(\beta)$ then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$.

4. For each production of the form $A \rightarrow \alpha B$, $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$. 
if-then-else vs. if-then

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

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>i</td>
</tr>
<tr>
<td>then</td>
<td>t</td>
</tr>
<tr>
<td>else</td>
<td>e</td>
</tr>
<tr>
<td>booleans</td>
<td>b</td>
</tr>
<tr>
<td>other expressions</td>
<td>a</td>
</tr>
</tbody>
</table>

and construct the following two augmented grammars \( G_1 \) and \( G_2 \).

1. \( S \rightarrow I \) $ 
2. \( I \rightarrow U \) 
3. \( I \rightarrow M \) 
4. \( U \rightarrow ibt I \) 
5. \( U \rightarrow ibt M e U \) 
6. \( M \rightarrow ibt M e M \) 
7. \( M \rightarrow a \)
Problems in LR parsing

1. Prove that grammar $G_2$ is ambiguous.

2. Construct the LR(0) parsing tables for both $G_1$ and $G_2$ and find all shift-reduce conflicts in the parsing table.

3. Construct the FOLLOW sets in each case and try to resolve the conflicts.

4. Show that the following augmented grammar cannot be parsed (i.e. there are conflicts that cannot be resolved by FOLLOW sets) either by LR(0) or SLR parsers. (*Hint* First construct the LR(0) DFA).

   1. $S \rightarrow E$
   2. $E \rightarrow L = R$
   3. $E \rightarrow R$
   4. $L \rightarrow * R$
   5. $L \rightarrow a$
   6. $R \rightarrow L$
Nullable

A nonterminal symbol $X$ is **nullable** if it can derive the empty string, i.e. $X \Rightarrow^* \varepsilon$.

**Algorithm 6 Nullable**

Require: CFG $G = \langle N, T, P, S \rangle$

Ensure: $\text{NULLABLE}(N \cup T)$

1. for all $\alpha \in N \cup T$ do
   2. if $\alpha \rightarrow \varepsilon \in P$ then
      3. $\text{NULLABLE}(\alpha) := true$
   4. else
      5. $\text{NULLABLE}(\alpha) := false$
   6. end if
7. end for
8. repeat
9. for all $X \rightarrow \alpha_1 \ldots \alpha_k \in P$ do
10. if $\forall i : 1 \leq i \leq k : \text{NULLABLE}(\alpha_i)$ then
11. $\text{NULLABLE}(X) := true$
12. end if
13. end for
14. until $\text{NULLABLE}(N \cup T)$ is unchanged
**First**

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

**Algorithm 7 First**

Require: CFG $G = \langle N, T, P, S \rangle$

Ensure: $FIRST(N \cup T)$

1. for all $a \in T$ do
2. \quad $FIRST(a) := \{a\}$
3. end for
4. for all $X \in N$ do
5. \quad $FIRST(X) := \emptyset$
6. end for
7. repeat
8. \quad for all $X \rightarrow \alpha_1 \ldots \alpha_k \in P$ do
9. \quad \quad for $i := 1 \ldots k$ do
10. \quad \quad \quad if $\forall i': 1 \leq i' < i : NULLABLE(\alpha_{i'})$ then
11. \quad \quad \quad \quad $FIRST(X) := FIRST(X) \cup FIRST(\alpha_i)$
12. \quad \quad \quad end if
13. \quad \quad end for
14. \quad end for
15. until $FIRST(N \cup T)$ sets are all unchanged
First And Follow

Notice that if \( X \rightarrow \alpha Z \beta \) is a production then one cannot ignore the \( \text{FIRST}(Z) \) in computing \( \text{FIRST}(X) \) especially if \( \alpha \Rightarrow^* \varepsilon \). Further if \( Z \) is also nullable then \( \text{FIRST}(\beta) \subseteq \text{FIRST}(X) \).

\( \text{FOLLOW}(X) \) for any nonterminal symbol \( X \) is the set of terminal symbols \( a \) such that there exists a rightmost derivation of the form

\[
S \Rightarrow^* \cdots X a \cdots \Rightarrow^*
\]

i.e. \( \text{FOLLOW}(X) \) is the set of all terminal symbols that can occur to the right of \( X \) in a rightmost derivation.

Notice that if there exists a a rightmost derivation of the form

\[
S \Rightarrow^* \cdots X \alpha_1 \cdots \alpha_k a \cdots \Rightarrow^*
\]

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

\[
S \Rightarrow^* \cdots X \alpha_1 \cdots \alpha_k a \cdots \Rightarrow^* \cdots X a \cdots \Rightarrow^*
\]
Computing Follow

Algorithm 8 Follow

Require: CFG $G = (N, T, P, S)$
Ensure: $FOLLOW(N)$

1. for all $\alpha \in N \cup T$ do
2. $FOLLOW(\alpha) := \emptyset$
3. end for
4. repeat
5. for all $X \rightarrow \alpha_1 \ldots \alpha_k \in P$ do
6. for $i := 1 \ldots k$ do
7. if $\forall i' : i + 1 \leq i' \leq k : NULLABLE(\alpha_{i'})$ then
8. $FOLLOW(\alpha_i) := FOLLOW(\alpha_i) \cup FOLLOW(X)$
9. end if
10. for $j := i + 1 \ldots k$ do
11. if $\forall i' : i + 1 \leq i' \leq j - 1 : NULLABLE(\alpha_{i'})$ then
12. $FOLLOW(\alpha_i) := FOLLOW(\alpha_i) \cup FIRST(\alpha_j)$
13. end if
14. end for
15. end for
16. end for
17. until $FOLLOW(N \cup T)$ sets are all unchanged
Recursive Descent Parsing

- Suitable for grammars that are LL(1)
- A set of (mutually) recursive procedures
- Has a single procedure/function for each non-terminal symbol
- Allows for syntax errors to be pinpointed more accurately than most other parsing methods
Caveats with RDP: Left Recursion

Any direct or indirect left-recursion in the grammar can lead to infinite recursive calls during which no input token is consumed and there is no return from the recursion. In particular,

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

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

Example 4.11

$$A \rightarrow B\beta$$

$$B \rightarrow A\alpha$$

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

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

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

**Example 4.12** A set of productions of the form

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

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

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

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:

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

Now this grammar is no longer left-recursive and may then be parsed by a recursive-descent parser.
4.8. Specification of Syntax: Extended Backus-Naur Form

Specification of Syntax: EBNF
4.8.1. The Extended Backus-Naur Form (EBNF)

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

**Start symbol.** The rules are written usually in a “top-down fashion” and the very first rule gives the productions of the start symbol of the grammar.

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

**Metasymbols.**

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

**Terminals.** Terminal symbol strings are usually enclosed in double-quotes when written in monochrome (we shall additionally colour-code them).
Balanced Parentheses: CFG

Example 4.13 A context-free grammar for balanced parentheses (including the empty string) over the terminal alphabet \{(, ), [ , ] , { , } \} could be given as \( BP_3 = \langle \{ S \} , \{(, ), [ , ] , { , } \} , P , \{ S \} \rangle \), where \( P \) consists of the productions:

\[
\begin{align*}
S & \rightarrow \epsilon, \\
S & \rightarrow (S)S, \\
S & \rightarrow [S]S, \\
S & \rightarrow \{S\}S
\end{align*}
\]
Balanced Parentheses: EBNF

Example 4.14 $BP_3$ may be expressed in EBNF as follows:

$$
\begin{align*}
\text{BracketSeq} &::= \{ \text{Bracket} \}. \\
\text{Bracket} &::= \text{LeftParen} \text{BracketSeq} \text{RightParen} | \\
& \quad \text{LeftSqbracket} \text{BracketSeq} \text{RightSqbracket} | \\
& \quad \text{LeftBrace} \text{BracketSeq} \text{RightBrace}. \\
\text{LeftParen} &::= "(". \\
\text{RightParen} &::= ")". \\
\text{LeftSqbracket} &::= "[". \\
\text{RightSqbracket} &::= "]". \\
\text{LeftBrace} &::= "\{". \\
\text{RightBrace} &::= "\}".
\end{align*}
$$
EBNF in EBNF

EBNF has its own grammar which is again context-free. Hence EBNF (4.8.1) may be used to define EBNF in its own syntax as follows:

\[
\begin{align*}
\text{Syntax} & ::= \{ \text{Production} \} . \\
\text{Production} & ::= \text{NonTerminal} \ "::=" \ PossibleRewrites \ "." . \\
\text{PossibleRewrites} & ::= \text{Rewrite} \ \{ \ "|" \ \text{Rewrite} \} . \\
\text{Rewrite} & ::= \text{Symbol} \ \{ \text{Symbol} \} . \\
\text{Symbol} & ::= \text{NonTerminal} \mid \text{Terminal} \mid \text{GroupRewrites} . \\
\text{GroupRewrites} & ::= \"\}" \ PossibleRewrites \ "\}" \mid \\
& \quad \("[" \ PossibleRewrites \ "\]" \mid \\
& \quad \("(\" \ PossibleRewrites \ "\))" . \\
\text{NonTerminal} & ::= \text{Letter} \ \{ \text{Letter} \mid \text{Digit} \} . \\
\text{Terminal} & ::= \text{Character} \ \{ \text{Character} \} .
\end{align*}
\]
EBNF: Character Set

The character set used in EBNF is described below.

Character ::= Letter | Digit | SpecialChar
Letter ::= UpperCase | LowerCase
UpperCase ::= “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” |
            “I” | “J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” |
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

Expression

ObjectDeclaration

SignatureDeclaration

FunctorDeclaration

ObjectDeclaration

Declaration

structure Ident : Signature = Structure

local ObjectDeclaration in ObjectDeclaration end

:
**Signature Declaration**

```
signature Ident = Signature and ...
```

**Functor Declaration**

```
functor FunctorBinding and ...
```

**Functor Binding**

```
Ident ( FunctorArguments ) : Signature = Structure ...
```

**Functor Arguments**

```
Ident : Signature Specification ...
```

**Structure**

```
struct ObjectDeclaration end

CompoundIdent

Ident ( Structure )

let ObjectDeclaration in Structure end
```

**Signature**

```
sig Specification end

Ident Specification ...
```
val Ident : Type
and
type TypeVarList Ident
and
eqtype DatatypeBinding
and
datatype DatatypeBinding
and
exception Ident of Type
and
structure Ident : Signature
and
sharing CompoundIdent
type CompoundIdent =
and
local Specification in Specification end
and
open CompoundIdent
and
include Ident
and
DECLARATIONS

Declaration

\[
\text{val} \quad \text{Pattern} \quad \rightarrow \quad \text{Expression} \\
\text{rec} \\
\text{and} \\
\text{fun} \quad \text{FunHeading} \quad \rightarrow \quad \text{Expression} \\
\text{Type} \\
\text{1} \quad \text{and} \\
\text{type} \quad \text{TypeBinding} \\
\text{datatype} \quad \text{DatatypeBinding} \\
\text{withtype} \quad \text{TypeBinding} \\
\text{abstype} \quad \text{DatatypeBinding} \\
\text{withtype} \quad \text{TypeBinding} \\
\text{with} \quad \text{Declaration} \quad \text{end} \\
\text{exception} \quad \text{Name} \quad \text{of} \quad \text{Type} \\
\text{=} \quad \text{CompoundName} \\
\text{and} \\
\text{local} \quad \text{Declaration} \quad \text{in} \quad \text{Declaration} \quad \text{end} \\
\text{open} \quad \text{CompoundIdent} \\
\text{infix} \quad \text{Digit} \\
\text{infixr} \quad \text{Ident} \\
\text{nonfix} \\
\text{;}
\]
Syntax Diagrams of SML: 5

1. Function
   Name
   AtomicPattern
   /
   /
   /\Operator
   AtomicPattern
   /
   /
   /\Operator
   AtomicPattern

TypeBinding

   TypeVarList
   Ident
   \=
   Type
   and

DatatypeBinding

   TypeVarList
   Ident
   \=
   Ident
   of
   Type
   and

TypeVarList

   TypeVar
   ( TypeVar
   )
EXPRESSIONS

Expression

InfixExpression

InfixOperator

InfixExpression

AtomicExpression
Syntax Diagrams of SML: 7

**AtomicExpression**

```
AtomicExpression

  CompoundName
  Constant

  Expression

  Label

  let Declaration in Expression end

MATCHES AND PATTERNS

**Match**

```
Pattern => Expression
```

**Pattern**

```
AtomicPattern

  CompoundName

  Pattern InfixOperator Pattern

  Pattern Name Type as Pattern
```
FieldPattern

FieldPattern

FieldPattern
TYPES

Type

LEXICAL MATTERS: IDENTIFIERS, CONSTANTS, COMMENTS

CompoundIdent

CompoundName

Name

InfixOperator
Syntax Diagrams of SML: 10

1. Constant
   Numeral
   /.
   /
   /
   /
   Digit
   /
   /
   /
   /
   Numeral
   /
   /
   /
   /
   E
   /
   /
   /
   /
   any printable character except \ and "
   StringEscape

2. StringEscape
   n
   t
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   one of "@ABCDEFGHIJKLMNOPQRSTUVWXYZ\]^~_`
   Digit
   Digit
   Digit
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
   /
Syntax Diagrams of SML: 11

1. **Ident**
   - AlphanumericIdent
     - Letter
     - Digit
     - one of `/!/%/&/$/#/+/-///:/</=/>/?/@/\/~/`/^/|/*

2. **Label**
   - Ident
     - Digit

3. **AlphanumericIdent**
   - Letter
     - Letter
     - Digit
     - one of `/0123456789`

4. **Digit**
   - one of `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`

5. **Letter**
   - any text that does not include `(*) as a substring`

6. **Comment**
   - any text that does not include `(*) as a substring`
Exercise 4.3

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.
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 \(val\) associated with an expression

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

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

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

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

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

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

Note: The attribute \(n.val\) 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.
5.1.1. Synthesized Attributes

Synthesized Attributes: 0

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

Synthesized Attributes
Synthesized Attributes: 3

The diagram shows a tree structure with labeled nodes. The synthesized attributes are indicated with numbers (4 3 2 1) at the bottom of the tree.
Synthesized Attributes: 4

Synthesized Attributes

4 3 2 1
Synthesized Attributes: 5
Synthesized Attributes: 6
Synthesized Attributes: 7

Synthesized Attributes

4 3 2 1
Synthesized Attributes: 8
Synthesized Attributes: 9
Synthesized Attributes: 10
Synthesized Attributes: 11

Synthesized Attributes
Synthesized Attributes: 12
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.
Inherited Attributes: 0

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

\[
D \rightarrow T L \\
T \rightarrow \texttt{int} \mid \texttt{float} \\
L \rightarrow L,I \mid I \\
I \rightarrow x \mid y \mid z
\]
Inherited Attributes: 1

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: 2

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: 3

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

\[
\begin{align*}
D & \rightarrow T \ L \\
T & \rightarrow \texttt{int} \mid \texttt{float} \\
L & \rightarrow L, I \mid I \\
I & \rightarrow x \mid y \mid z
\end{align*}
\]
Inherited Attributes: 4

C-style declarations generating \texttt{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: 5

C-style declarations generating 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: 6

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

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

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

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

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

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

In the first semantic rule the symbol \(L.in\) is inherited from a symbol to its right viz. \(T.type\) and hence is not L-attributed.
Evaluating Non-L-attributed Definitions

In many languages like ML which allow higher order functions as values, a definition not being L-attributed may not be of serious concern. But in most other languages it is serious enough to warrant changing the grammar of the language so as to replace inherited attributes by corresponding synthesized ones. The language of the grammar of Pascal and ML declarations can be generated as follows:

\[ D \rightarrow \text{id}L \quad \triangleright \quad \text{addtype}(\text{id}, \text{L.type}) \]
\[ L \rightarrow :T \quad \triangleright \quad \text{L.in} := \text{T.type} \]
\[ L \rightarrow \text{,id} \text{ L} \quad \triangleright \quad \text{L}_0.\text{type} := \text{L}_1.\text{type}; \]
\[ \text{addtype}(\text{id}.\text{L}_1.\text{type}) \]
\[ T \rightarrow \text{int} \quad \triangleright \quad \text{T.type} := \text{int.int} \]
\[ T \rightarrow \text{real} \quad \triangleright \quad \text{T.type} := \text{real.real} \]
Dependency Graphs

In general, the attributes required to be computed during parsing could be synthesized or inherited and further it is possible that some synthesized attributes of some symbols may depend on the inherited attributes of some other symbols. In such a scenario it is necessary to construct a dependency graph of the attributes of each node of the parse tree.
Dependency Graph Construction

Algorithm 9 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

```
  /
 /--
|  
|--
|  
|  
\--/  
  |
  |
  |
  4 1
```

Evaluation: 1
Evaluation: 2
Evaluation: 3

Diagram: A tree with three nodes labeled 1, 2, and 3.
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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\(^a\)i.e.\textit{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 $\ bop $ is a binary operation
  - $x := uop \ y$, where $uop$ is a unary operation
  - $x := y$, load, store, copy or register transfer

- Jumps (conditional and unconditional)

- Procedures and parameters

- Arrays and array-indexing

- Pointer Referencing and Dereferencing
A typical instruction set: 2.2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
  - \texttt{goto L} – Unconditional jump,
  - \texttt{x relop y goto L} – Conditional jump, where \texttt{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

- \( x \leftarrow \hat{y} \) – referencing: set \( x \) to point to \( y \)
- \( x \leftarrow y \) – dereferencing: copy contents of location pointed to by \( y \) into \( x \)
- \( *x \leftarrow y \) – dereferencing: copy \( r\)-value of \( y \) into the location pointed to by \( x \)

Picture
Pointers

\[
x := \^y
\]

\[
x := \*y
\]

\[
x := \*z
\]

\[
\*x := y
\]
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

\[ E \rightarrow id \]

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

\[ E_0 \rightarrow E_1 - E_2 \]

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

\[ S \rightarrow \text{id} := E \]

\[ S\.code := E\.code \mid \text{emit}(\text{id}.\text{place}:=E\.\text{place}) \]

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

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

\[ S \rightarrow \text{id} := E \]

\[ S.code := E.code \]
\[ \mid \mid \text{emit(id.place} := E.place) \]

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

\[ S_0.begin := \text{newlabel;} \]
\[ S_0.after := \text{newlabel;} \]
\[ S_0.code := \text{emit(S}_0.begin:) \]
\[ \mid \mid E.code \]
\[ \mid \mid \text{emit(if } E.place= 0 \text{ goto } S_0.after) \]
\[ \mid \mid S_1.code \]
\[ \mid \mid \text{emit(goto } S_0.begin) \]
\[ \mid \mid \text{emit(S}_0.after:) \]
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

$$L, M, N ::= \begin{align*}
    x & \quad \text{Variable} \\
    \lambda x [L] & \quad \text{Abstraction} \\
    (L M) & \quad \text{Application}
\end{align*}$$

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.

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

- The set of bound variables $BV(N) = Var(N) - FV(N)$.
- The same variable name may be used with different bindings in a single term (e.g. $(\lambda x[x] \lambda x[(x y)])$).
- The brackets “[” and “]” delimit the scope of the bound variable $x$ in the term $\lambda x[L]$.
- $\Lambda_0 \subseteq \Lambda$ is the set of closed $\lambda$-terms (i.e. terms with no free variables).
Notational Conventions

To minimize use of brackets unambiguously

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

2. \( (L_1 L_2 \cdots L_m) \) 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:

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

- 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 $z \not\in FV(N) \cup Var(L)$.
- $z$ could be fresh even if $z \in BV(N)$.
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

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

**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.
$\alpha$-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 $\Lambda$

\[ \rightarrow_\beta \overset{\text{df}}{=} \{ ((\lambda x[L] M), \{M/x\}L') \mid L' \equiv_\alpha L, L', L, M \in \Lambda \} \]

• It is usually represented by the axiom

\[ (\lambda x[L] M) \rightarrow_\beta \{M/x\}L' \]

(4)

where $L' \equiv_\alpha L$. 
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

<table>
<thead>
<tr>
<th>$\beta_1$</th>
<th>$L \rightarrow^1_{\beta} M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow^1_{\beta} M$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\beta_1$Abs</th>
<th>$L \rightarrow^1_{\beta} M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x[L] \rightarrow^1_{\beta} \lambda x[M]$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\beta_1$AppL</th>
<th>$L \rightarrow^1_{\beta} M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(L N) \rightarrow^1_{\beta} (M N)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\beta_1$AppR</th>
<th>$L \rightarrow^1_{\beta} M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(N L) \rightarrow^1_{\beta} (N M)$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

$$\begin{align*} \beta^* & \quad \frac{L \rightarrow^n_\beta M}{L \rightarrow^*_\beta M} \quad (n \geq 0) \end{align*}$$
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

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

or at some point it could transform to

$$
(K \ (\omega \ \omega)) \rightarrow_{\beta}^{1} \lambda y[\(\omega \ \omega\)] \rightarrow_{\beta}^{1} \lambda y[\(\omega \ \omega\)] \rightarrow_{\beta}^{1} \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

$$
((K \ \omega) \ \Omega) \rightarrow_{\beta}^{1} ((K \ \omega) \ \Omega) \rightarrow_{\beta}^{1} \cdots
$$
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:

\[
((S \, K) \, K) \rightarrow^1_\beta \lambda z[((K \, z) \, (K \, z))] \rightarrow^1_\beta \lambda z[z] \equiv I
\]
10. Notions of Reduction

11. Logic Programming and Prolog

Prolog: Abstract Interpreter

Algorithm 10 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 ~ 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 $-$ \{A\}) $\cup$ \{B_1, ..., B_k\}
9: end while
10: if empty(resolvent) then
11:    return yes
12: else
13:    return no
14: end if