# COMP0012 Compilers Notes

Joe Xu
Henry Zhang
Felix Hu
Davies Xue
John Xu

May 22, 2019

# Contents

# 1 Introduction

Notes from COMP0012 Lecture Slides.

## 1.1 Compiler Importance

- Raise abstraction level.
- Bridge gap between human mind and machine execution.
- Increase maintainability of code.

## 1.2 Compiler Goals

- Translate source code into *equivalent* machine code.
- Translation efficiency.
- Useful error messages.
- Parallel compilation support.
- Program correctness verification.

## 1.3 High-level Compiler Structure

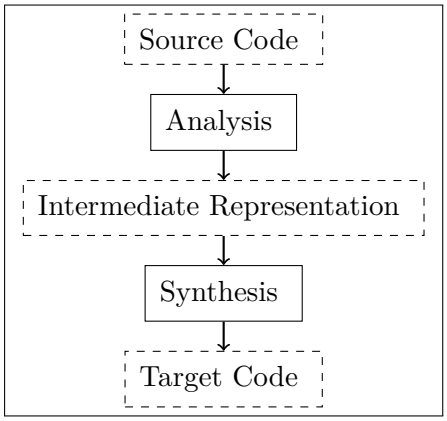



Figure 1: Compiler high-level overview

## 1.4 Compiler Front-end

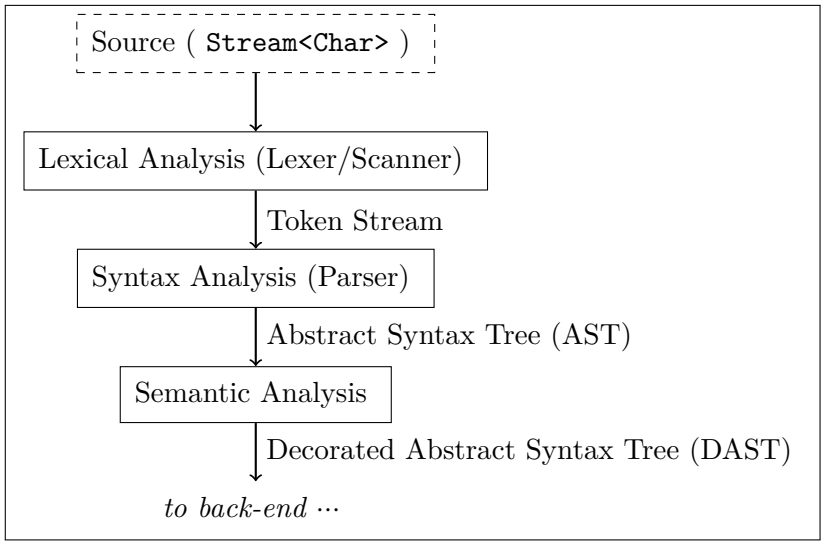



Figure 2: Compiler front-end overview

## 1.5 Compiler Back-end

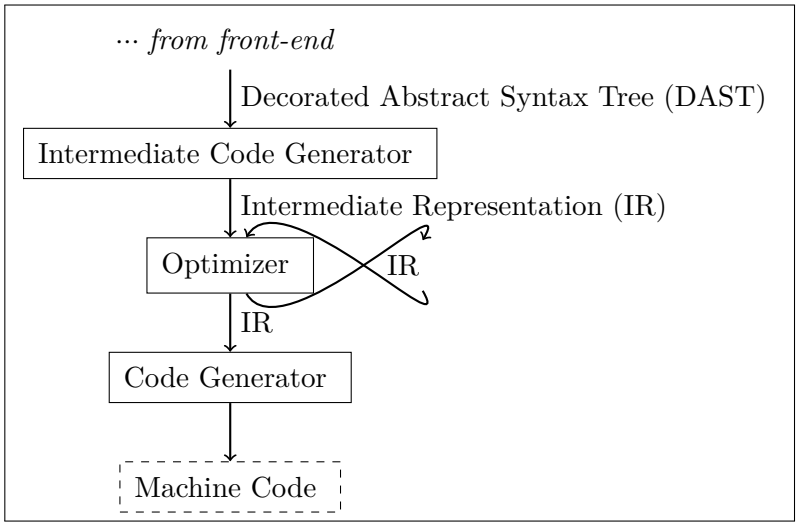



Figure 3: Compiler back-end overview

## 1.6 Lexical Analysis

To extract *lexemes*[1] from source code:

```
int main(void)
{
    return 0;
}
```

(a) Source Code

```
INT INDENTIFIER("MAIN") LPAREN
VOID RPAREN LBRACE RETURN
INT_LITERAL(0) SEMI RBRACE
```

(b) Lexemes

Figure 4: Example of converting source code to lexemes

## 1.7 Syntax Analysis

To recognize *structure* of a phrase in the language, which produces an *Abstract Syntax Tree* (AST).

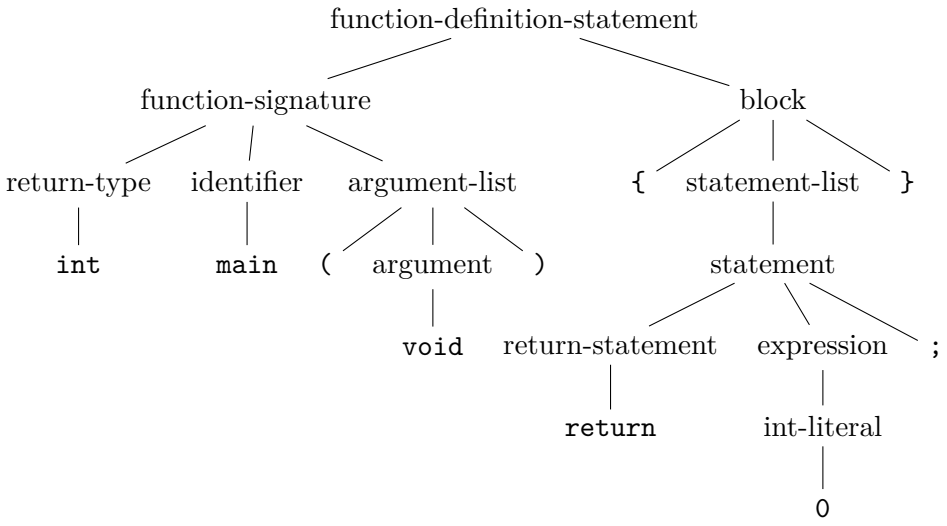



Figure 5: Simple parse tree of Figure 4a

[1]Note that *lexeme* refers to a word in the source code, while a *token* is a set of related lemexes such as *identifiers*.

## 1.8 Semantic Analysis

Validate the source code to have acceptable *meaning*. Note that while the *syntax* may be valid, but the *semantics* may not be.

```
String a;
if (a == 1)
    System.out.println(a);
```

Figure 6: Syntactically correct but semantically wrong. Note that trying to use the referential equality comparison operator `==` with two arguments of different types in this case yields a `TypeError`.

## 1.9 Error Handling and Symbol Table

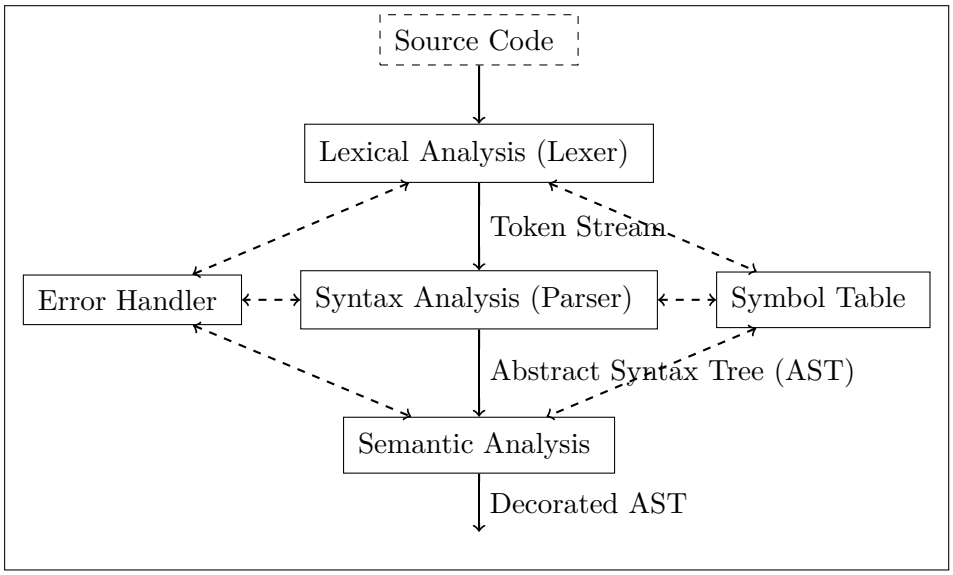



Figure 7: Error handling and symbol table

## 1.10 Intermediate Code Generator

To produce *intermediate code* which allows:

- *Portability* (machine and language independence).
- *Decoupling* between front-end and back-end.
- Allow *transpiling* and *optimization*.

### 1.10.1 Intermediate Representation (IR)

Three tiers of *intermediate representation* (IR) in varying degree of abstraction:

- High-level IR (HIR)
- Mid-level IR (MIR)
- Low-level IR (LIR)

Note that HIR is closest to AST and LIR is closest to machine code. Examples of LIR include:

- Three Address Code (TAC)
- LLVM IR
- GCC Gimple

## 1.11 Optimizer

Rewrite *intermediate code* to run faster and/or take up less space.

## 1.12 Code Generation

Output the program in the target language (usually assembly).

## 1.13 Combined Phases

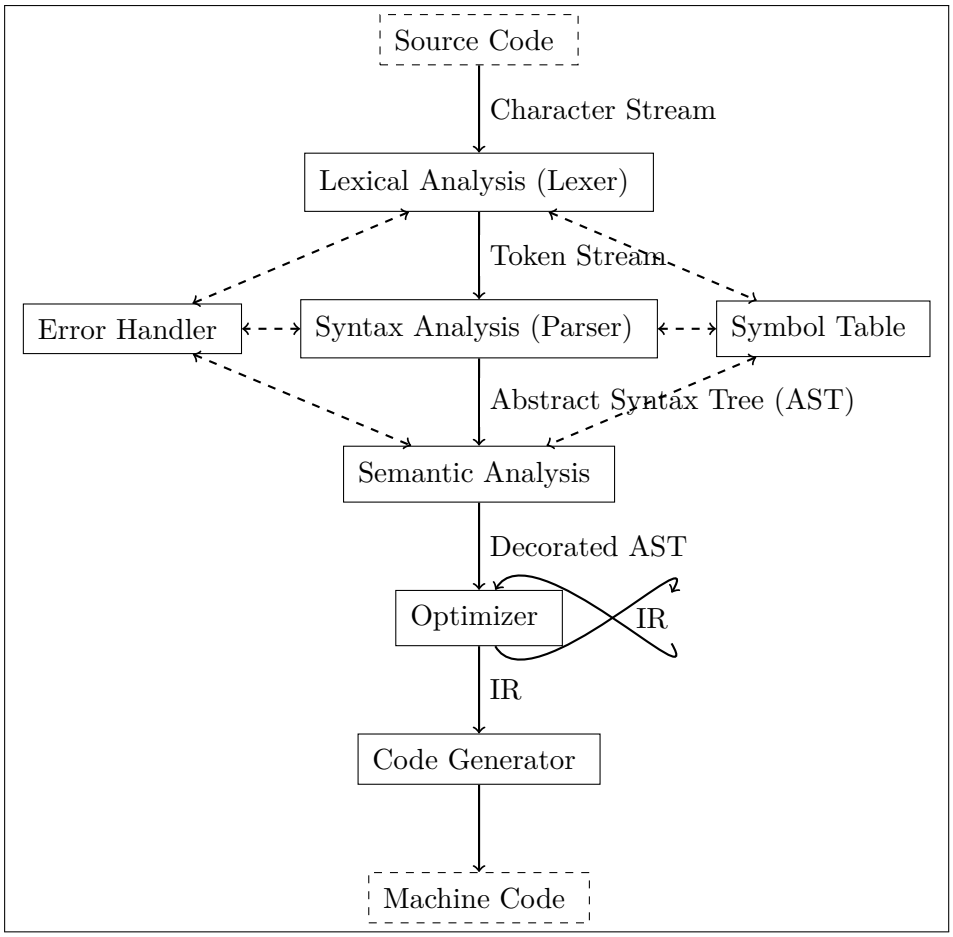
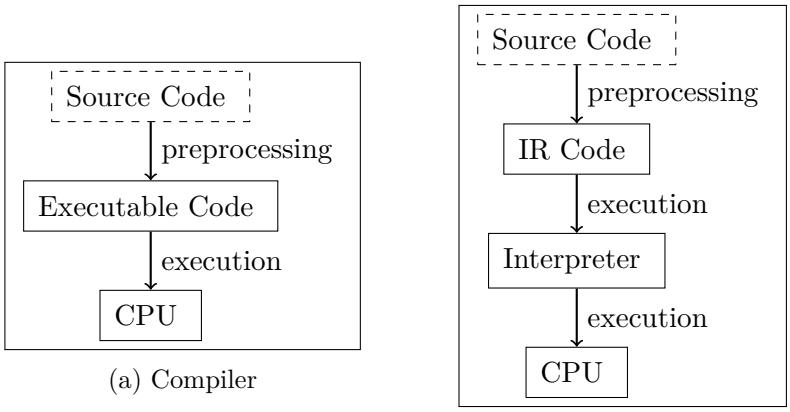



Figure 8: All phases of a compiler

### 1.14 Compiler versus Interpreter




(a) Compiler

(b) Interpreter

Figure 9: Difference between a compiler and an interpreter

## 2 Lexical Analysis

Extracts *lexemes* from source code taken in as a stream of charcaters.

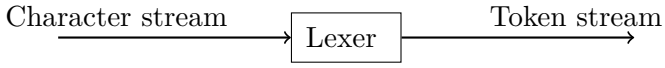



Figure 10: Lexical Analysis I/O

**Definition 2.1** (Lexeme)**.** A *lexeme* is a sequence of characters which form a *lexical unit* in the grammar of a given language.

**Definition 2.2** (Delimiter)**.** A *delimiter* is a character or a sequence of characters which separates lexemes.

*Remark.* A *lexeme* is analogous to a "word" within an English sentence separated by *delimiters*, such as a "space".

### 2.1 Regular Expressions

*Regular expressions* (often abbreviated as *regex*) can be used to describe the lexemes of a language.

To a recognize a *regex*, one may use a *Finite State Automata* (FSA), namely the *Deterministic Finite Automata* (DFA), which is sufficiently powerful in recognizing *regular* languages.

### 2.1.1 Definition of a Language

**Definition 2.3** (Alphabet)**.** An *alphabet* $\Sigma$ is a finite set of symbols.

**Definition 2.4** (String)**.** A *string* over $\Sigma$ is a consecutive sequence of symbols from alphabet $\Sigma$.

**Definition 2.5** (Language)**.** A *language* $L$ is a set of strings over an alphabet $\Sigma$.

### 2.1.2 Regular Operations

**Definition 2.6** (Regular Operations)**.** Given languages $L_0$ and $L_1$, there are three fundamental *regular operations* defined as

1. **Union** $(L_0 \cup L_1)$ (also denoted $L_0 \mid L_1$)

$$L_0 \cup L_1 \coloneqq \{x \mid x \in L_0 \vee x \in L_1\} \tag{1}$$

2. **Concatenation** $(L_0 \circ L_1)$ (also denoted $L_0 L_1$)

$$L_0 \circ L_1 \coloneqq \{xy \mid x \in L_0 \wedge y \in L_1\} \tag{2}$$

3. **Kleene Star** $(L_0^*)$

$$L_0^* \coloneqq \{x_1 x_2 \cdots x_k \mid k > 0 \wedge x_i \in L_0\} \cup \{\varepsilon\} \tag{3}$$

### 2.1.3 Regular Expression Formal Definition

**Definition 2.7** (Regular Expression)**.** For a given alphabet $\Sigma$, the *regular expression* $R$ is defined inductively as

1. $a \in \Sigma$
2. $\varepsilon$
3. $\emptyset$
4. $(R_1 \mid R_2)$
5. $(R_1 R_2)$
6. $(R_1^*)$

Where

- $R_1$, $R_2$ are valid *regular expressions*.
- $\varepsilon$ denotes the empty string.
- $\emptyset$ is the empty set.

*Remark.* Note that *parentheses* are need for the inductive cases to prevent ambiguity. This definition assumes that the concatenation, union and Kleene star operators have equal precedence.

*Example.* Given alphabet $\Sigma = \{a, b\}$, then

| Regular Expression $R$ | $L(R)$ |
|---|---|
| $a$ | $\{a\}$ |
| $a \mid b$ | $\{a, b\}$ |
| $(ab)^*$ | $\{\varepsilon, ab, abab, ababab, ...\}$ |
| $(a \mid \varepsilon)b$ | $\{b, ab, aab, ...\}$ |
| $(a \mid b^*)a$ | $\{a, aa, ba, bba, ...\}$ |

Note that $L(R)$ denotes "the *language* $L$ of the *regex* $R$".

### 2.1.4 Associativity

To reduce the number of *parantheses* required to unambiguously interpret a *regex*, it is possible to assign different *priorities* to the three operators $*$, $\circ$ and $|$.

**Definition 2.8** (Regular Operator Precedence)**.** For the regular operators,

| Operator Precedence | Operator |
|---|---|
| Highest $\uparrow$ | Kleene Star $*$ |
| | Concatenation $\circ$ |
| Lowest $\downarrow$ | Union $\cup$ |

Figure 11: Regular Operator Precedence

*Remark. Precedence* can be thought as "binding power", that is, higher precedence implies higher binding power, and vice versa.

*Example.* The regex

$$ab^* \mid cd$$

Is equivalently

$$((a \circ (b^*)) \mid (c \circ d))$$

### 2.1.5 Syntax Sugars

**Definition 2.9** (Syntax Sugar)**.** *Syntax sugar* for *regexes* are introduced as aliases to make the regular language easier to express. However, syntax sugars do not otherwise change the expressiveness of regular languages – they are simply *aliases*.

*Example.* Given the alphabet of the lowercase Roman alphabet $\Sigma$, one can introduce several syntax sugars operator and expressions

| Syntax Sugar | Equivalent Regular Expression |
|---|---|
| $a^+$ | $aa^*$ |
| $a?$ | $a \mid \varepsilon$ |
| $[abc]$ | $a \mid b \mid c$ |
| $[a\text{-}z]$ | $a \mid b \mid c \mid \cdots \mid z \equiv \Sigma$ |
| . | $a \mid b \mid c \mid \cdots \mid z \equiv \Sigma$ |
| $[\hat{}a\text{-}c]$ | $d \mid \cdots \mid z \equiv \Sigma - \{a, b, c\}$ |

Figure 12: Syntax sugar for the lowercase Roman alphabet

*Remark.* Operators introduced means there is a need to escape the *meta-characters*, the characters which themselves are used to represent the regular operators.

For example, to use the meta-character | as an input character for a regular expression, one could escape it as \|.

### 2.1.6 Ambiguity

**Definition 2.10** (Ambiguous)**.** A regex is *ambiguous* if it recognizes input string in multiple ways.

*Example.* The regex $(00)^*(000)^*$ is *ambiguous* since it can recognize the input string "000000" as either

- $000 \circ 000$, or
- $00 \circ 00 \circ 00$.

*Example.* A more significant example would be in the case of keywords vs identifiers in a programming language. For example,

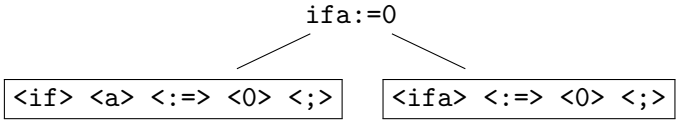



Figure 13: Ambiguity in keywords vs identifiers

It is then necessary to disambiguate between different possible matches and only retain a single matching pattern.

**Definition 2.11** (Disambiguation)**.** Two rules are used for *disambiguation*:

1. **Longest Match**
2. **Rule Priority** (order)

### 2.1.7 Greedy Matching

**Definition 2.12** (Greedy Matching)**.** *Greedy matching* refers to the default behaviour for regexes to recognize as much of the input as possible. For instance, to match HTML tags such as

```
<a href="/"> Test </a>
```

The pattern

$$\langle \Sigma^+ \rangle$$

Matches the entire input sequence

`<a href="/"> Test </a>`

To match only the tags themselves, one would need to exclude the opening and closing angle brackets

$$\langle (\Sigma - \{\langle, \rangle\})^+ \rangle$$

Which would yield the desired match

`<a href="/">` `Test` `</a>`

### 2.1.8 Searching versus Lexing

**Definition 2.13** (Searching)**.** *Searching* aims to find a substring which satisfies a regex and can ignore the rest of the input characters.

*Remark.* Note that while *searching* aims to find *a* substring which matches the supplied regex, *lexing* aims to find *all* substrings which matches the supplied regex.

## 2.2 Deterministic Finite Automata

To recognize a *regular expression*, one may use a *Deterministic Finite Automata* (DFA).

**Definition 2.14** (Deterministic Finite Automata (DFA))**.** A *Deterministic Finite Automata* (DFA) $M$ can be described via the 5-tuple

$$M = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle \tag{4}$$

Where

1. **Alphabet**: $\Sigma$ is the finite set of characters in the *alphabet*.
2. **States**: $\mathcal{Q}$ is the finite set of *states*.
3. **Transition Function**: $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. This function transitions from one *state* to another on a suitable input character.
4. **Start State**: $q_0 \in \mathcal{Q}$ is the *start state*.
5. **Final States**: $\mathcal{F} \subseteq \mathcal{Q}$ is the set of *final states*.

### 2.2.1 DFA Notation

A DFA can be represented graphically with the following notations.


(a) State (b) Start State (c) Final State (d) Transition

Figure 14: DFA Notations

**Definition 2.15** (DFA Notations)**.** Note that $q_1 \times a \to q_2$ denotes the transition from state $q_1$ to $q_2$ on reading input character $a$.

*Remark.* Note that DFA diagrams usually omit the trash state $q_{trash}$, which is the implicit final state that rejects the read input string.

## 2.3 Non-deterministic Finite Automata

A *Non-deterministic Finite Automata* (NFA) is a DFA with added convience features that do not otherwise change its expressive power.

**Definition 2.16** (Non-deterministic Finite Automata (NFA))**.** A *Non-deterministic Finite Automata* (NFA) $M$ can be described by the 5-tuple

$$M = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle \tag{5}$$

Where

1. **Alphabet**: $\Sigma$ is the finite set of characters making up the *alphabet*.

2. **States**: $\mathcal{Q}$ is the finite set of *states*.
3. **Transition Function**: $\delta \colon \mathcal{Q} \times \Sigma_\varepsilon \to \mathcal{P}(\mathcal{Q})$.
4. **Start State**: $q_0 \in \mathcal{Q}$.
5. **Final States**: $\mathcal{F} \subseteq \mathcal{Q}$.

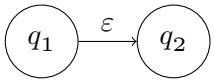
Note that $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ and $\mathcal{P}(\cdot)$ is the power set.

*Remark.* Special attention needs to be paid to the *transition function* of the NFA. Notice that for the transition function $\delta$

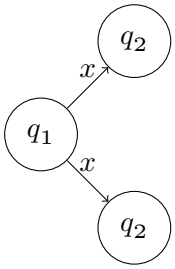
$$\delta \colon \mathcal{Q} \times \Sigma_\varepsilon \to \mathcal{P}(\mathcal{Q}) \tag{6}$$

- There may be $\varepsilon$-transitions (read as *epsilon*-transitions). The NFA may change states *without* reading input.
- On a single input character, the NFA can transition to *multiple* states.
- On a single input character, the NFA can also go to *no* state ($\emptyset$).

These convenience features combined gives rise to the *non-determinism* in the name of the NFA.




(a) Epsilon Transition




(b) Transition to Multiple States

Figure 15: NFA Convenience Features

**Definition 2.17** (NFA Acceptance)**.** An NFA *accepts* an input string if it is able to reach a final state upon consuming the input string.

## 2.4 DFA versus NFA

| Aspect | DFA | NFA |
|---|---|---|
| For each input character | Definitive next state | Can either take $\varepsilon$-transition or take which one of multiple possible transition. |
| Acceptance | Ends in final state | Has a path from start state to final state |
| Implementation | Table-driven | Set of possible states |
| Execution | Faster and less space needed | Slower and more space needed |

Figure 16: DFA versus NFA

*Remark.* Note that NFA often has fewer nodes and edges compared to a DFA due to its non-determinism.

## 2.5 Implementing Lexing Automata

### 2.5.1 Overview

Implementing a lexer specification involves the following process.

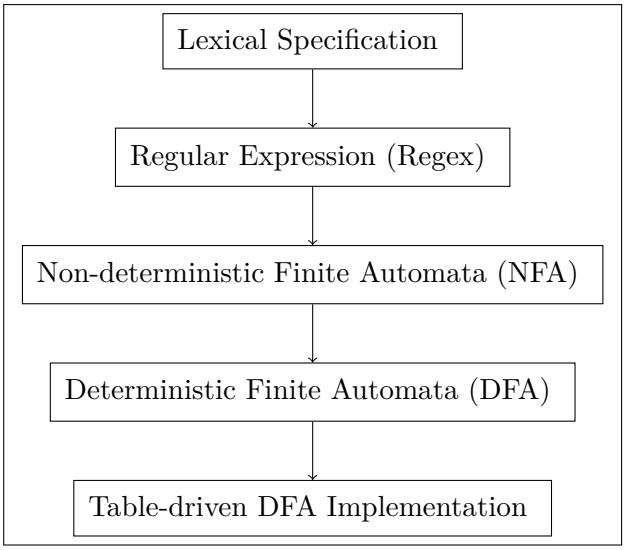



Figure 17: Lexing Automata Implementation

## 2.6 Regular Expression to NFA Conversion (Thompson's Construction)

The algorithm for converting regex to NFA is called *Thompson's Construction*.

**Definition 2.18** (Thompson's Construction)**.** Given regex $A$, let $N(A)$ denote the application of the rules of *Thompson's Construction*.

Let $A$ have a single start and final state.

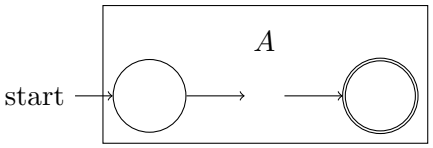



Figure 18: Regex $A$

**CASE 1**: Single Symbol.

Transition for a single input character $a \in \Sigma$ (note that $\varepsilon$ is simply a symbol).

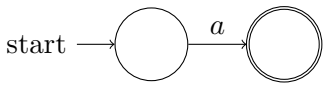



Figure 19: Single Symbol Case

**CASE 2**: Concatenation.

Given two regexes $A$ and $B$, then $AB$ is simply $A$'s final state *definalized*, and merged as $B$'s start state. That is, $A$'s final acceptance state is no longer accepting and becomes $B$'s start state.

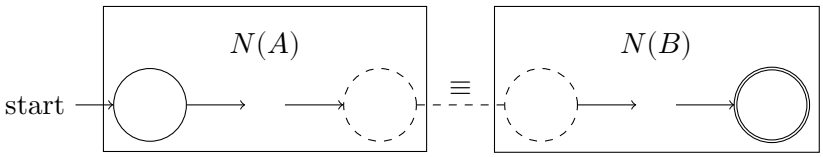



Figure 20: Concatenation Case

**CASE 3**: Union (Alternation).

Given two regexes $A$, $B$, then the union $A \mid B$ can be converted to NFA by:

1. Introduce new start state.
2. Definalize $A$ and $B$'s final state.
3. Introduce new final state.
4. Connect the new start state, new final state, and $A$ and $B$ with $\varepsilon$-transitions.

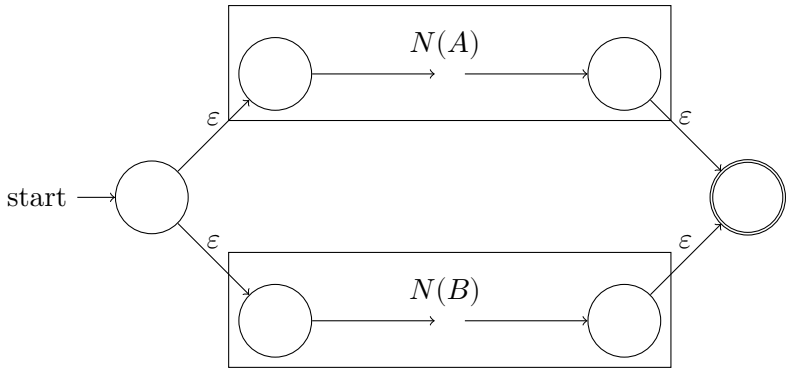



Figure 21: Union Case

**CASE 4**: Kleene Star.

Given regex $A$, then its Kleene star $A^*$ can be converted to NFA by connecting $N(A)$'s start state and final state.

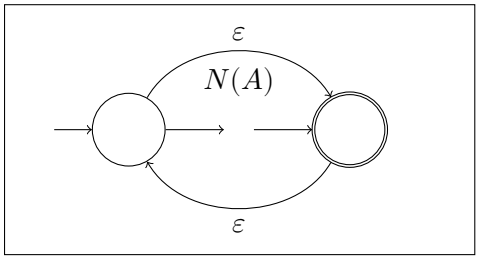



Figure 22: Kleene Star Case

## 2.7 NFA to DFA Conversion

It is also possible to convert from NFA to DFA as they have equivalent power.

**Definition 2.19** (NFA to DFA Conversion)**.** Given NFA $\langle \Sigma, \mathcal{Q}_N, \delta_N, q_N, \mathcal{F}_N \rangle$, it is possible to construct an equivalent DFA $\langle \Sigma, \mathcal{Q}_D, \delta_D, q_D, \mathcal{F}_D \rangle$.

Each state $q_D$ of the DFA is a subset of the states of the NFA.

$$\forall\, q_D \in \mathcal{Q}_D : q_D \subseteq \mathcal{Q}_N \tag{7}$$

**Definition 2.20** (Epsilon Closure)**.** The *$\varepsilon$-closure* of a state $q$, denoted $\varepsilon(q)$, is the set of NFA states which are reachable from $q$ with $\varepsilon$-transitions.

$$\varepsilon(q) = \left\{ r \in \mathcal{Q} \mid q \xrightarrow[\varepsilon]{*} r \right\} \tag{8}$$

Note that $\varepsilon(q) \neq \emptyset$ since the state $q$ is itself in its $\varepsilon$ closure, that is $q \in \varepsilon(q)$.

**Definition 2.21** (NFA to DFA High Level Algorithm)**.** The target is to convert from

$$NFA = \langle \Sigma, \mathcal{Q}_N, \delta_N, s_N, \mathcal{F}_N \rangle \implies \langle \Sigma, \mathcal{Q}_D, \delta_D, s_D, \mathcal{F}_D \rangle = DFA \tag{9}$$

Where

$$s_D \equiv \varepsilon(s_N) \tag{10}$$

For each DFA state $q_D \in \mathcal{Q}_D$, add the transition

$$q_D \times a \to T \subseteq \mathcal{Q}_N \tag{11}$$

To $\delta_D$ if there exists a state $s$ where $T$ is

$$T = \varepsilon(\delta_N(s, a)) \tag{12}$$

If some $q_D \in \mathcal{Q}_D$ contains one of the NFA's final states in $\mathcal{F}_N$, then mark $q_D$ as final as well, that is $q_D \in \mathcal{F}_D$.

**Definition 2.22** (NFA to DFA: addTransition)**.** To compute the *transitions* between the DFA states, the procedure `addTransition` can be used on

$$q_D \in \mathcal{Q}_D \subseteq \mathcal{P}(\mathcal{Q}_N) \tag{13}$$

Then `addTransition` has the signature

$$\texttt{addTransition}\colon \mathcal{Q}_D \times \Sigma \to \mathcal{Q}_D \times \delta_D \tag{14}$$

**Algorithm 1** Compute transitions $\delta_D$ for a DFA state $q_D$ given input $a$

```
1: procedure ADDTRANSITION(⟨q_D, a⟩)
2:     q'_D ← ∅
3:     for all q_N ∈ q_D do
4:         q'_D ← q'_D ∪ ε(δ_N(q_N, a))
5:     end for
6:     δ_D ← {q_D × a → q'_D}
7:     return ⟨q'_D, δ_D⟩
8: end procedure
```

**Definition 2.23** (NFA to DFA: Conversion Algorithm)**.** Then to convert from a NFA to DFA:

**Algorithm 2** Convert NFA to DFA.

**procedure** convertNfaToDfa($\langle \Sigma, \mathcal{Q}_N, \delta_N, s_N, \mathcal{F}_N \rangle$)
  $s_D \leftarrow \varepsilon(s_N)$
  $\delta_D \leftarrow \emptyset$
  $\mathcal{Q}_D \leftarrow \{s_D\}$
  WC.enqueue($s_D$) ▷ WC is the workset.
  **while** ¬ WC.isEmpty() **do**
    $q_D \leftarrow$ WC.dequeue()
    **for all** $a \in \Sigma$ **do**
      $\langle q'_D, \delta'_D \rangle \leftarrow$ addTransition($q_D, a$)
      $\delta_D \leftarrow \delta_D \cup \{\delta'_D\}$
      **if** $q'_d \notin \mathcal{Q}_D$ **then**
        WC.enqueue($q'_D$)
      **end if**
      $\mathcal{Q}_D \leftarrow \mathcal{Q}_D \cup \{q'_D\}$
    **end for**
  **end while**
  $\mathcal{F}_D \leftarrow \{x \in \mathcal{Q}_D \mid x \cap \mathcal{F}_N \neq \emptyset\}$
  **return** $\langle \Sigma, \mathcal{Q}_D, \delta_D, s_D, \mathcal{F}_D \rangle$
**end procedure**

### 2.7.1 DFA Implementation

**Definition 2.24** (Table-driven DFA)**.** A DFA can be implemented with a 2D table $T[\mathcal{Q}, \Sigma]$. The two dimensions are for

1. Set of states $\mathcal{Q}$.
2. Alphabet $\Sigma$.

Transitions of the form $s_i \xrightarrow{a} s_j$ corresponds to the table entry

$$T[i, a] = j \tag{15}$$

When the DFA is in state $s_i$, upon reading input character $a$, it is then trivial to find the next state $s_j = T[i, a]$.

## 2.8 Finite State Transducers

**Definition 2.25.** The *Finite State Transducer* (FST) $M$ can be described by the 6-tuple

$$M = \langle \Sigma, \Gamma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle \tag{16}$$

Where

1. **Input Alphabet**: $\Sigma$ is the set of characters making up the *input alphabet*.
2. **Output Alphabet**: $\Gamma$ is the set of characters making up the *output alphabet*.
3. **States**: $\mathcal{Q}$ is the finite set of *states*.
4. **Transition Function**: $\delta : \mathcal{Q} \times \Sigma_\varepsilon \to \Gamma_\varepsilon \times \mathcal{Q}$.
5. **Start State**: $q_0 \in \mathcal{Q}$.
6. **Final States**: $\mathcal{F} \subseteq \mathcal{Q}$.

Note that $\Gamma_\varepsilon := \Gamma \cup \{\varepsilon\}$ and $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$.

### 2.8.1 FST Output Function

There are two output functions defined implicitly in the previous definition of a FST.

**Definition 2.26** (Mealy Output Function)**.** The *Mealy output function* is implicit in the FSA's transition function, and is defined as

$$\mathcal{Q} \times \Sigma_\varepsilon \to \Gamma_\varepsilon \tag{17}$$

This output function outputs on the *edges*. Its edges are labeled as "$i/o$", where $i \in \Sigma_\varepsilon$ and $j \in \Gamma_\varepsilon$.

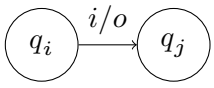



Figure 23: Mealy Output Function

**Definition 2.27** (Moore Output Function)**.** The *Moore output function* is also implicit in the FSA's transition function, and is defined as

$$\mathcal{Q} \to \Gamma_\varepsilon \tag{18}$$

This output function outputs upon entry to a state. Its nodes are labelled with output "$o$".


Figure 24: Moore Output Function

*Remark.* Note that FSTs allow us to label transitions into error states for error reporting.

# 3 Syntactic Analysis

The language used in *Syntactic Analysis* is the *Context-Free Grammar* (CFG). Sometimes *Parsing Expression Grammar* (PEG) is also used, but PEG is not covered in this course.

## 3.1 Context-Free Grammar

**Definition 3.1** (Context-Free Grammar (CFG))**.** A *Context-Free Grammar* (CFG) $G$ can be described by the 4-tuple

$$G = \langle \mathcal{N}, \mathcal{T}, S, \mathcal{R} \rangle \tag{19}$$

Where

1. **Non-terminals**: $\mathcal{N}$ is the finite set of *non-terminals* (uppercase by convention).
2. **Terminals**: $\mathcal{T}$ is the finite set of *terminals* (lowercase by convention).
3. **Start Symbol**: $S \in \mathcal{T}$ is the *start* symbol.
4. **Production Rules**: $\mathcal{R} \coloneqq \mathcal{N} \to (\mathcal{N} \cup \mathcal{T})^*$ is the set of finite relations, termed *productions* or *rules* of the grammar.

*Remark.* By convention,

- Non-terminals are written in uppercase.
- Terminals are either punctuation characters or written in lowercase.
- Start symbol is the left-hand side non-terminal of the first product.

*Example.* A *production*

$$X \to Y_1 \cdots Y_n$$

Means that the non-terminal $X$ can be *replaced* by $Y_1 \cdots Y_n$. Equivalently, the production right-hand side $Y_1 \cdots Y_n$ is *produced* by $X$.

### 3.1.1 Language of a Context-Free Grammar

**Definition 3.2** (Language of a Context-Free Grammar)**.** Given context-free grammar $G$ with the start symbol $S$, the *language* of $G$ is

$$L(G) \coloneqq \left\{ a_1 \cdots a_n \mid S \overset{*}{\to} a_1 \cdots a_n, a_i \in \mathcal{T} \right\} \tag{20}$$

That is, $L(G)$ is the set of all strings of *terminals* for which the grammar $G$ can generate in zero or more steps.

### 3.1.2 Terminals

**Definition 3.3** (Terminals)**.** Characters in the input alphabet $\mathcal{T}$ are called *terminals* because there does not exist any production rules which can replace them.

That is, *terminals* only appear in the *right-hand side* of any production rule.

*Remark.* If terminals are generated by any replacement steps, then they are *permanent*.

*Terminals* are often *lexemes* in the language. For example, the terminal `void` is a keyword lexeme in the C language.

## 3.2 Right Regular Grammar

**Definition 3.4** (Right-regular Grammar)**.** A *Right-regular Grammar* $G$ is the 4-tuple

$$G = \langle \mathcal{N}, \mathcal{T}, S, \mathcal{R} \rangle \tag{21}$$

And also satisfying three additional *constraints*; given $A, B \in \mathcal{N}$ and $a \in \mathcal{T}$, and given $B$ is *right-regular*,

1. $A \to a$
2. $A \to aB$
3. $A \to \varepsilon$

*Remark.* Note that a *Left-regular Grammar* has the above definition with only one difference, in the second constraint; given $A, B \in \mathcal{N}$ and $B$ is *left-regular*,

$$A \to Ba \tag{22}$$

If left and right rules are mixed together, a linear grammar is generated which is context-free.

### 3.2.1 Derivations

**Definition 3.5** (Derivation)**.** A *derivation* is a sequence of *sential forms* resulting from repeated applications of some *production rule* beginning from $S$, replacing *non-terminals* with their respective *productions*.

*Example.* For instance,

$$\begin{aligned} S &\to X_1 \cdots X_a \cdots X_b \cdots \\ &\to X_1 \cdots Y_c \cdots X_b \cdots \\ &\to \cdots \\ &\to Y_1 \cdots Y_m \\ &\to \cdots \\ &\to \alpha_1 \cdots \alpha_n \end{aligned}$$

The intermediate strings with mixed terminals and non-terminals are the *sentential forms*.

The last derivation with $S \overset{*}{\Rightarrow} \alpha_1 \cdots \alpha_n$ with each $\alpha_1, \ldots, \alpha_n \in \mathcal{T}$ being terminals is called a *sentence*.

**Definition 3.6** (Sentential Form)**.** A *sentential form* $\alpha$ refers to the *start symbol* $S$ or any strings of *terminals* and *non-terminals* $(\mathcal{N} \cup \mathcal{T})^*$ that can be *derived* from $S$.

$$\alpha = (\mathcal{N} \cup \mathcal{T})^* \tag{23}$$

**Definition 3.7** (Sentence)**.** A *sentence* $\beta$ refers to a string of only *terminals*, usually the last step in a sequence of *derivations*.

$$\beta = \mathcal{T}^* \tag{24}$$

## 3.3 Parse Trees

**Definition 3.8** (Parse Tree)**.** A *parse tree* illustrates a *derivation*.

1. Start symbol $S$ is the *root* of the tree.
2. For production $X \to Y_1 \cdots Y_n$, create a *node* with *key* $X$ and *children* $Y_1, \ldots, Y_n$.

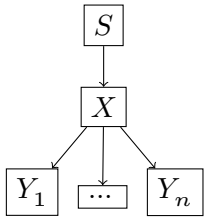



Figure 25: Generic Parse Tree. Note root $S$ is the start symbol and $X$ is an interior node of $S$ with children $Y_1, \ldots, Y_n$.

*Remark.* A *parse tree* corresponds with a *Context-Free Grammar* by having

- Terminals as *leaves*.
- Non-terminals as *interior nodes*.

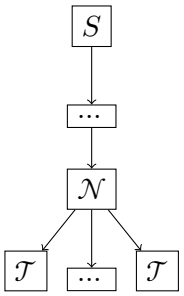



Figure 26: Parse tree correspondence with CFG

- An *in-order traversal* of the *parse tree* yields the original input string.
- The *parse tree* contains the *association* of *operations* for which the input string does not.

## 3.4 Left-most and Right-most Derivations

**Definition 3.9** (Left-most Derivation)**.** The *left-most derivation* occurs when the *left-most non-terminal* is always expanded.

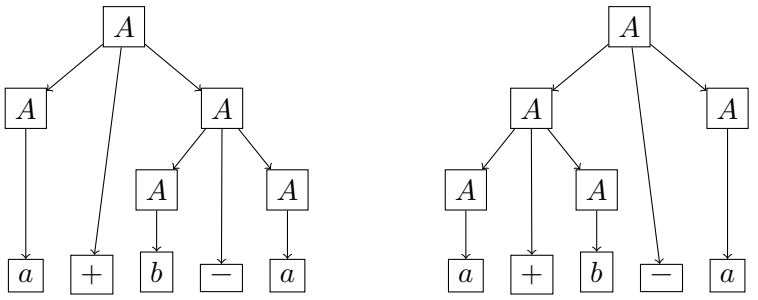
**Definition 3.10** (Right-most Derivation)**.** The *right-most derivation* occurs when the *right-most non-terminal* is always expanded.

*Example.* For a very simple grammar for addition and subtraction expressions

$$\begin{array}{rcl} S & \rightarrow & E + E \\ & | & E - E \\ & | & E \\ E & \rightarrow & a \\ & | & b \end{array}$$

This grammar is *ambiguous* because two parse trees can be produced for the input string

$$a + b - a$$




(a) Left-most derivation (b) Right-most derivation

Figure 27: Ambiguous grammar with multiple valid parse trees

When the *left-most* and *right-most* derivations both produce the *same* parse tree, then the grammar is *unambiguous*.

## 3.5 Chomsky Hierarchy

**Definition 3.11** (Chomsky Hierarchy)**.** The *Chomsky Hierarchy* classifies different languages in increasing expressive power.

| Expressiveness | Grammar | Production Constraint | Automata |
|---|---|---|---|
| ↑ | Universal | $\alpha \to \beta$ | Turing Machine |
| | Context-Sensitive | $\alpha A \beta \to \alpha \delta \beta$ | Linear Bounded Automata |
| | Context-Free | $A \to \alpha$ | Push-Down Automata |
| ↓ | Regular (right) | $A \to a \mid aB \mid \varepsilon$ | Deterministic Finite Automata |

Figure 28: Chomsky Hierarchy

## 3.6 Solutions to Resolve Ambiguity

1. **Hacking**: Rewrite the grammar to enforce the precedence.

*Example.* Simple grammar solution

$$\begin{array}{rcl} E & \rightarrow & E + E \\ & | & E * E \\ & | & (E) \\ & | & id \end{array}$$

converting to

$$\begin{array}{rcl} E & \rightarrow & E' + E \\ & | & E' \end{array}$$

$$\begin{array}{rcl} E' & \rightarrow & (E) * E' \\ & | & (E) \\ & | & id * E' \\ & | & id \end{array}$$

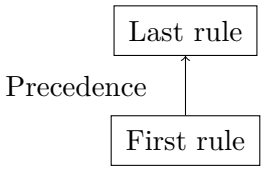
2. **Unambiguous annotations**:
   Using natural grammar but along with forced annotations (for example, precedence).

   *Example.* A simple example

   ```
   left  +
   right *
   ```

   Where

   ```
   %left  -->  left associative
   %right -->  right associative
   ```




## 3.7 Top-Down Parsing

**Definition 3.12** (Top-Down Parsing)**.** Given an input string consisting of terminals

$$t_1 \; t_2 \; \cdots \; t_n$$

Then its *parse tree* is constructed

- From *top* to *bottom*.
- From *left* to *right*.

### 3.7.1 Recursive Descent Parsing

**Definition 3.13** (Parsing)**.** *Parsing* aims to find the parse tree for a string in grammar $G$

$$t_1\ t_2\ \cdots\ t_n \tag{25}$$

**Definition 3.14** (Recursive Descent Parsing)**.** *Recursive Descent Parsing* tries to build the parse tree starting from the start symbol $S$ and trying all productions exhaustively.

- The *fringe* of the parse tree is

$$t_1\ t_2\ \cdots\ t_k\ A\ \cdots \tag{26}$$

- The parser needs to *backtrack* if the fringe does not match the $k$-length prefix of the string.
- Try all productions for $A$.
  - If there is a production $A \to BC$
  - Then the new fringe is

$$t_1\ t_2\ \cdots\ t_k\ B\ C\ t_{k+1}\ \ldots\ t_n \tag{27}$$

- Terminate when there are no more non-terminals remaining.

### 3.7.2 Limitations of Recursive Descent

*Recursive Descent* does *not* work when the grammar is *left-recursive*.

**Definition 3.15** (Left-recursive Grammar)**.** A *left-recursive grammar* has a non-terminal $A$ which, for zero or more derivations, derives

$$S \to^+ S\ \alpha \tag{28}$$

For some $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$. Recursive descent parsing *diverges* on *left-recursive* grammars.

*Example.* Given a production

$$S \to S\ a \tag{29}$$

Then it is possible to keep on replacing $S$ in the right-hand side with the production yielding no fringe to match (because the start can always be the non-terminal):

$$\begin{aligned} S &\to S\ a \\ &\to S\ a\ a \\ &\to S\ a\ a\ a\ \cdots \end{aligned}$$

### 3.7.3 Elimination of Left Recursion

**Definition 3.16** (Left Recursion Elimination)**.** Given a grammar $G$ which contains a *left-recursive* production of non-terminal $A$

$$\begin{aligned} A \rightarrow\ & A\ \alpha_1 \\ \mid\ & \cdots \\ \mid\ & A\ \alpha_n \\ \mid\ & \beta_1 \\ \mid\ & \cdots \\ \mid\ & \beta_m \end{aligned} \tag{30}$$

Where $\alpha_i \in (\mathcal{N} \cup \mathcal{T})^+$ and $\beta_i \in ((\mathcal{N} - \{A\}) \cup \mathcal{T})^+$.

Then this rule can be converted into a *right-recursive* rule by introducing a new non-terminal $A'$ and rewriting the original rule to give

$$\begin{aligned} A \rightarrow\ & \beta_1\ A' \\ \mid\ & \cdots \\ \mid\ & \beta_m\ A' \\ A' \rightarrow\ & \alpha_1\ A' \\ \mid\ & \cdots \\ \mid\ & \alpha_n\ A' \\ \mid\ & \varepsilon \end{aligned} \tag{31}$$

**Definition 3.17** (Indirect Left Recursion Elimination)**.** Given some production rules

$$\begin{aligned} S &\rightarrow A\ \alpha \mid \beta \\ A &\rightarrow S\beta \end{aligned} \tag{32}$$

Then $S$ is *left recursive* since

$$S \rightarrow^+ S\ \beta\ \alpha \tag{33}$$

This indirect left recursion can be eliminated by back-substituting $A$ into $S$ to give

$$S \rightarrow S\ \beta\ \alpha \mid \delta \tag{34}$$

Then perform the previous rewrite technique.

**Definition 3.18** (Nullable Non-terminal)**.** A non-terminal $N$ is *nullable* iff $N \rightarrow^+ \varepsilon$.

**Definition 3.19** ($\varepsilon$-masked Left Recursion Elimination)**.** Given some production with the form

$$\begin{aligned} S &\to Y\ S \mid q \\ Y &\to A\ x\ B \mid C\ c \mid c\ d \mid \varepsilon \end{aligned} \tag{35}$$

Since $Y \to^{+} \varepsilon$ is *nullabe*, then $S$ is in fact *left-recursive*. This left recursion can be eliminated by performing back-substitution, similar to the indirect recursion case, then perform the previous rewrite technique.

Note that should the back-substitution step generate any rules of the form

$$Y \to Y \tag{36}$$

Then that production can be simply discarded (it has no effect).

Also note that such back-substitution may cause an exponential growth in the number of production rules.

**Definition 3.20** (Generic Left Recursion Elimination)**.** With each left recursive type considered, it is possible to give a generic left recursion elimination procedure.

**Algorithm 3** Generic Left Recursion Elimination

```
1: procedure EliminateLeftRecusion(G)
2:     repeat
3:         EliminateUnreachableRules(G)
4:         EliminateUselessProductions(G)
5:         EliminateDirectLeftRecursionViaRewrite(G)
6:         EliminateIndirectLeftRecursionViaBackSubstitution(G)
7:         EliminateEpsilonMaskedLeftRecursionViaBackSubstitution(G)
8:     until NoEliminationApplies()
9: end procedure
```

### 3.7.4 Predictive Parsers

**Definition 3.21** (Predictive Parser)**.** Similar to *recursive descent* parsers but with *lookahead(s)* to choose the correct production. I.e. for some grammar $G$, when looking at

$$t_1\ t_2\ \cdots\ t_k\ B\ C\ t_{k+1}\ \cdots\ t_n \tag{37}$$

The terminals $t_{k+1}\ \cdots\ t_n$ may be considered to choose a production.

**Definition 3.22** ($LL(k)$ Grammars)**.** *Predictive* parsers accept $LL(k)$ grammars, where $LL(k)$ denotes *left-to-right* input scan, *leftmost* derivation with $k$ tokens of *lookahead.* Usually $LL(1)$ grammars are used in practice.

### 3.7.5 Parse Tables

**Definition 3.23** (Parse Tables)**.** Given a context-free grammar $G$, then its *Parse Table* is a 2D table with the dimensions:

1. Current non-terminal production.
2. Next $k$ lookahead tokens.

Then each table entry $T[N, a]$ where $N \in \mathcal{N}, a \in \mathcal{T}$ contains a subset of productions which has the non-terminal head in the first dimension, i.e.

$$T[N, a] = \{(N \to \alpha), (N \to \beta), \cdots\} \tag{38}$$

**Definition 3.24** ($LL(1)$ Parse Table)**.** $LL(1)$ grammars imply that given a pair of non-terminal and lookahead token $(N, a)$ where $N \in \mathcal{N}, a \in \mathcal{T}$, there exists only *one* production which can lead to a successful parse.

That is, a $LL(1)$ parse table has two dimensions:

1. Current non-terminal.
2. Next lookahead token $k = 1$.

The restriction by $LL(1)$ grammars means that each table entry is restricted to contain only a single production, i.e.

$$T[N, a] = \{(N \to \alpha)\} \tag{39}$$

Where

$$|T[N, a]| \equiv 1 \tag{40}$$

*Remark.* Note that trash states may be implicitly denoted by empty cells, which do not violate the single-production-per-cell constraint for $LL(1)$ grammars.

### 3.7.6 Left Factoring

**Definition 3.25** (Converting to $LL(1)$ Grammar)**.** A grammar may have to be *left-factored* into a $LL(1)$ grammar before it is able to be parsed via predictive parsing.

*Example.* Two productions with the same lookahead token for the same non-terminal cause the grammar to be beyond $LL(1)$, e.g.

$$\begin{aligned} T &\to int \\ &\mid int * T \end{aligned} \tag{41}$$

Since both productions begin with *int*, it is impossible for the $LL(1)$ predictive parse to choose which production is the correct one based on a single lookahead token.

**Definition 3.26** (Left Factoring)**.** *Left Factoring* is the process to rewrite a grammar so that each production of each non-terminal has a unique prefix.

This is analogous to factoring out a common prefix, i.e.

$$ax + ay = a * (x + y) \tag{42}$$

Given a non-terminal $N$ that needs to be left factored, and some $A, B \in (\mathcal{N} \cup \mathcal{T})^+$, supposing that $N$ has the form

$$\begin{aligned} N \to\ & A\ B_1 \\ |\ & A\ B_2 \\ |\ & \cdots \\ |\ & A\ B_n \\ |\ & \langle \text{rules not prefixed with } A \rangle \end{aligned} \tag{43}$$

Then it can be left-factored by introducting a new non-terminal $N'$ and rewriting $N$ to the form

$$\begin{aligned} N \to\ & A\ A' \\ |\ & \langle \text{rules not prefixed with } A \rangle \\ A' \to\ & B_1 \\ |\ & B_2 \\ |\ & \cdots \\ |\ & B_n \end{aligned} \tag{44}$$

Note that any $B_i$ can be $\varepsilon$.

### 3.7.7 LL(1) Parsing

**Definition 3.27** ($LL(1)$ Parse Actions)**.** Given a $LL(1)$ parse table $T$

1. If $T[N, a] = N \to \beta$:
   - $N$ is to be replaced by or expanded to $\beta$.
2. If $T[N, a] = \varepsilon$:
   - $N$ is to be discarded.

**Definition 3.28** (Building $LL(1)$ Parse Table)**.** Given the parse is in some state considering

$$S \to^* \beta\ A\ \gamma \tag{45}$$

Where $A$ is the non-terminal being considered and $b$ is the lookahead token.

Then there exists two possibilities for the position of $b$:

1. $b \in \text{First}(A)$: $b$ belongs to some expansion of $A$, that is, if $b$ can start a string derived from $\alpha$ from the production

$$A \to \alpha \tag{46}$$

2. $b \in \text{Follow}(A)$: $b$ does not belong to any expansion of $A$.
   - $A \to^* \varepsilon$ and $b \in \text{First}(\gamma)$, in $S \to^* \beta\ A\ b\ \omega$ supposing $\gamma \to^* b\ \omega$.
   - Note that any of $A \to \alpha$ can be used if $\alpha \to^* \varepsilon$, i.e. $\varepsilon \in \text{First}(A)$.

**Definition 3.29** (First Set)**.** Given a grammar $G$ and a non-terminal $X \in \mathcal{N}$, then the *First* set of $X$ is defined as

$$\text{First}(X) := \{t \in \mathcal{T} \mid X \to^+ t\ \alpha\} \cup \{\varepsilon \mid X \to^+ \varepsilon\} \tag{47}$$

Where $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$.

**Definition 3.30** (Computing First Sets)**.** Given a grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle$, with $\mathcal{T}_\varepsilon := \mathcal{T} \cup \{\varepsilon\}$, then the algorithm to compute the first set of its non-terminal $X \in \mathcal{N}$ is given as

**Algorithm 4** Computing First Set

```
1:  procedure ComputeFirstSet(⟨N, T, R, S⟩, X ∈ N)
2:      ∀ t ∈ T_ε : First(t) ← {t}
3:      First(X) ← ∅
4:      for all productions P = X → A_1 ··· A_n ∈ R do
5:          for each A_i do
6:              First(X) ← First(X) ∪ (First(A_i) − {ε})
7:              if ε ∉ First(A_i) then
8:                  NextProduction()
9:              end if
10:         end for
11:         First(X) ← First(X) ∪ {ε}        ▷ Only if all A_i ∈ P are nullable
12:     end for
13:     return First(X)
14: end procedure
```

**Definition 3.31** (Follow Sets)**.** Given some grammar $G$ and non-terminals $\mathcal{N}$, then the *Follow* set of non-terminal $X$ is given by

$$\begin{aligned}\text{Follow}(X) \coloneqq \{t \mid Y \to \alpha\ X\ \beta \wedge t \in (\text{First}(\beta) - \{\varepsilon\})\} \cup \\ \{t \mid \beta \to^* \varepsilon \wedge t \in \text{Follow}(Y)\}\end{aligned} \tag{48}$$

Note that the non-terminal $X$ appearing on the right-hand side of the production rule of non-terminal $Y$. It is also required that Follow($X$) is the minimum set, if in the process of calculating Follow($X$) that Follow($X$) appears on the right-hand side, i.e. if there is an intermediate step

$$\text{Follow}(X) = \cdots \cup \text{Follow}(X) \cup \cdots \tag{49}$$

Then the Follow($X$) on the right-hand side is simply absorbed and has no effect.

**Definition 3.32** (Computing Follow Sets)**.** Given a grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle$, with end-of-input symbol \$, then the algorithm to compute the follow set of its non-terminal $X \in \mathcal{N}$ is given as

**Algorithm 5** Computing Follow Set

```
1:  procedure ComputeFollowSet(⟨N, T, R, S⟩, X ∈ N)
2:      for all non-terminals N ∈ N do
3:          First(N) ←ComputeFirstSet(⟨N, T, R, S⟩, N)
4:      end for
5:      Follow(X) ← ∅
6:      Follow(S) ← {$}                          ▷ Starting non-terminal S
7:      for all productions P = Y → ··· X A_1 ··· A_n ∈ R do
8:          for each A_i do
9:              Follow(X) ← Follow(X) ∪ (First(A_i) − {ε})
10:             if ε ∉ First(A_i) then
11:                 NextProduction()
12:             end if
13:         end for
14:         Follow(X) ← Follow(X) ∪ Follow(Y) ▷ Iff all A_i ∈ P is nullable
15:     end for
16: end procedure
```

It should be noted that

- $\varepsilon \notin \text{Follow}(X)$

**Definition 3.33** (Building $LL(1)$ Parse Table Algorithm)**.** To construct a $LL(1)$ parse table $T$ for a given grammar $G$

**Algorithm 6** Building $LL(1)$ parse table

```
1: procedure BuildLL1ParseTable(G)
2:     for each production A → α do
3:         for each terminal b ∈ First(α) − {ε} do
4:             T[A, b] ← α
5:         end for
6:         if α →* ε then
7:             for each b ∈ Follow(A) do
8:                 T[A, b] ← ε
9:             end for
10:        end if
11:    end for
12: end procedure
```

*Remark.* A $LL(1)$ parse table has exactly one production per table cell. If there are multiple productions per table cell, then the grammar is $LL(1)$ because of any of the following reasons:

- $G$ is *ambiguous*
- $G$ is *left recursive*
- $G$ is not *left factored*

**Definition 3.34** (Using $LL(1)$ Parse Table)**.** A $LL(1)$ parse utilizes the $LL(1)$ parse table similar to that in recursive descent, but with differences in that

- For each *non-terminal* $N$
- Consider the lookahead token $a$
- Choose production at

$$T[N, a] \tag{50}$$

The $LL(1)$ parser uses a *stack* to record pending non-terminals

- *Reject* if an error state is encountered.
- *Accept* if *end-of-input* is reached.

**Definition 3.35** ($LL(1)$ Parsing Algorithm)**.** Given input stream of characters, with $\langle H\ rest \rangle$ denoting the top of the stack and the rest of the stack respectively, then the $LL(1)$ parsing algorithm is given by

**Algorithm 7** $LL(1)$ Parsing Algorithm

```
1: procedure ParseWithLL1(InputTokenStream)
2:     I ← InputTokenStream
3:     stack ← ⟨S $⟩
4:     head ← 0
5:     repeat
6:         switch ⟨τ rest⟩ do                          ▷ τ is the top of stack
7:             case τ ∈ 𝒩                               ▷ τ is a non-terminal
8:                 if T[τ, I[head]] ≡ Y_1 ··· Y_n then
9:                     stack ← ⟨Y_1 ··· Y_n, rest⟩
10:                else
11:                    ReportError(τ)
12:                end if
13:                break
14:            case τ ∈ 𝒯                               ▷ τ is a terminal
15:                if τ ≡ I[head] then
16:                    stack ← ⟨rest⟩
17:                    head++
18:                else
19:                    ReportError(τ)
20:                end if
21:                break
22:    until stack ≡ ⟨$⟩
23: end procedure
```

Note that the right-hand side of $\tau$, $Y_1 \ \cdots \ Y_n$, is the production for which the *head* selects.

## 3.8 Bottom-Up Parsing

**Definition 3.36** (LR Parsing)**.** *Bottom-Up* parsers, or $LR$ parsers, construct the parse tree from the *bottom up*, where

- Tokens are read *left-to-right*.
- *Right-most* derivations are selected.

$LR$ parses have the additional benefit that grammars do not need to be left-factored, and left-recursive grammars can also be handled. As such, $LR$ parsers handle more grammars than $LL$ parsers.

**Definition 3.37** (High-level $LR$ Parsing Algorithm)**.** $LR$ parsing produces the parse tree by *reducing* the input string to the start symbol $S$ by *inverting*

*productions*.

Conceptually, it works by

**Algorithm 8** High Level $LR(1)$ Parsing Algorithm

**procedure** ParseWithLR1($InputString$)
  $str \leftarrow InputString$
  **repeat**
    Find rightmost $\beta \in (\mathcal{N} \cup \mathcal{T})^+$ which has been examined in $str = \alpha\ \beta\ \gamma$ with $A \to \beta$ being a production.
    Replace $\beta$ by $A$, meaning $str \Rightarrow \alpha\ A\ \gamma$.
  **until** $str \equiv S$ ▷ Until string becomes the start symbol
**end procedure**

It is actually a right-most derivation in reverse, by replacing the right-hand sides of production rules with their respective head non-terminals.

**Definition 3.38** ($LR(1)$ Parsing Notation)**.** $LR(1)$ parsing splits the input string into two substrings

1. **Left substring**: consisting of terminals and non-terminals, i.e. $(\mathcal{N} \cup \mathcal{T})^*$.
2. **Right substring**: unexamined terminals, can be considered source of additional context should the $LR(1)$ parser need more information in its current state.

The two partitions are separated by the symbol $\|$. Upon initial input, no input is examined, giving

$$\| \ x_1\ x_2\ \cdots\ x_n \tag{51}$$

**Definition 3.39** ($LR$ Actions)**.** Bottom-up $LR$ parsers use two *actions* to find a suitable parse tree:

1. Shift
2. Reduce

**Definition 3.40** (Shift)**.** The *Shift* action moves the $\|$ partition symbol one terminal to the right, adding a terminal from the right unexamined substring to the left substring.

Given some already examined left substring $\alpha$, then terminal $\tau_1$ from the right unexamined substring is shifted to the left substring

$$\alpha \ \| \ \tau_1\ \tau_2\ \cdots\ \tau_n \xrightarrow{\text{SHIFT}} \alpha\ \tau_1 \ \| \ \tau_2\ \cdots\ \tau_n \tag{52}$$

**Definition 3.41** (Reduce)**.** The *Reduce* action applies an inverse production at the right end of the left substring.

Given some left substring $\alpha\ \beta\ \gamma$ and a production rule $A \to \beta\ \gamma$, with a right substring $\zeta$, then

$$\alpha\ \underbrace{\beta\ \gamma}_{A \to \beta\ \gamma}\ \|\ \zeta \xRightarrow{\text{REDUCE}} \alpha\ A\ \|\ \zeta \tag{53}$$

**Definition 3.42** (Left Substring as a Stack)**.** The *left substring* can be implemented via a *stack*, with the top of the stack being the rightmost symbol, closest to $\|$.

- The *Shift* action pushes a terminal from the right substring on to the stack.
- The *Reduce* action pops 0 or more symbols off the stack (production right-hand side) and pushes a non-terminal on to the stack (production head left-hand side).

**Definition 3.43** (Deciding When to Shift or Reduce)**.** An *Finite State Automaton* (FSA) can be used to determine when to *shift* or *reduce* based on current stack content and the lookahead terminal.

- FSA takes the stack's content as input, which is the content to the left of $\|$.
- FSA's alphabet is $\Sigma \equiv \mathcal{N} \cup \mathcal{T}$.

When the FSA is run on the stack's content, examination of

1. *Resulting state* $X$, and
2. *Lookahead terminal* $s$ after $\|$ from the unexamined right substring

Determines whether to shift or reduce:

- **Shift**: iff $X$ has a transition out labelled with $s$, or
- **Reduce**: iff $X$ is labelled with

$$A \to \beta \text{ on } s \tag{54}$$

Note that this FSA is in fact a *Finite State Transducer*, as detailed in subsection 2.8.


(a) Shift condition

(b) Shift condition

Figure 29: Finite State Transducer deciding when to shift or reduce

### 3.8.1 Representing LR(1) Parsing FSA

**Definition 3.44** ($LR(1)$ Parsing FSA as 2D Table)**.** The $LR(1)$ parsing FSA can be represented with a 2D table:

1. FSA state as rows.
2. Terminals and non-terminals as columns.

The columns can be partitioned into

1. **Action table**: for lookaheads (terminals).
2. **Goto table**: for non-terminals.

*Remark.* After each shift or reduce action, the FSA is rerun on the entire stack which is inefficient. To improve, remember the resulting FSA state each stack element brings.

### 3.8.2 LR(1) Parsing Algorithm

**Definition 3.45** ($LR(1)$ Parsing Algorithm)**.** Let $\alpha_i \in \Sigma \equiv \mathcal{N} \cup \mathcal{T}$ and $\sigma_i$ be the set of FSA states. An $LR(1)$ parser maintains a stack with (state, sentential form) pairs.

$$\begin{gathered} \langle \sigma_1, \alpha_i \rangle \\ \langle \cdots \rangle \\ \langle \sigma_n, \alpha_n \rangle \end{gathered} \tag{55}$$

Each $\sigma_k$ is the FSA state reached on the sentential form

$$\alpha_1 \ \cdots \ \alpha_k \tag{56}$$

The FSA can then jump to $\sigma_n$ on $\alpha_n$.
More formally,

**Algorithm 9** $LR(1)$ Parsing Algorithm

```
1: procedure ParseWithLR1(InputString)
2:     I ← InputString $                                  ▷ is the input string
3:     j ← 0          ▷ j is the index into I and let σ_0 be FSA's start state
4:     stack.push(⟨σ_0, ε⟩)                          ▷ ε takes FSA to start state
5:     while true do
6:         ⟨σ, α⟩ ← stack.top()
7:         switch Action[σ, I[j]] do
8:             case Shift σ_t
9:                 stack.push(⟨σ_t, I[j]⟩)
10:                j++                                    ▷ Advance lookahead
11:                break
12:            case Reduce X → β                      ▷ β is RHS of production
13:                for i ← |β|, i ≥ 0, i−− do      ▷ |β| is the number of
   symbols in the RHS of the production
14:                    stack.pop()
15:                end for
16:                stack.push(⟨X, Goto[σ, α]⟩)
17:                break
18:            case Accept
19:                Halt()
20:            case Error
21:                ReportError(stack)
22:    end while
23: end procedure
```

**Definition 3.46** ($LR(1)$ Item)**.** An $LR(1)$ *item* is given by the pair

$$X \to \alpha \bullet \beta, a \tag{57}$$

Where

- $X \to \alpha \bullet \beta$ is a production, with $\alpha$ already on the stack and $\beta$ remaining to be handled.
- $a$ is the lookahead terminal.

Here $LR(1)$ refers to having one lookahead.

**Definition 3.47** (Parser Context)**.** The *context* of the parser can be described with

$$[X \to \alpha \bullet \beta, a] \tag{58}$$

- Looking for $X$ followed by the lookahead $a$.

- Top of stack is $\alpha$.
- Need to find prefix derived from $\beta\ a$.

Where $X \in \mathcal{N}$, $\alpha, \beta \in (\mathcal{N} \cup \mathcal{T})$.

**Definition 3.48** ($LR(1)$ Augmented Grammar)**.** A new start symbol $S'$ is added to the target grammar $G$ with the production rule

$$S' \to S \tag{59}$$

With $S$ being the old start symbol.
The initial parsing context contains

$$S' \to \bullet S, \$ \tag{60}$$

Indicating that
- Trying to find string derived from $S\$$.
- Stack is empty.

**Definition 3.49** (Closure Operation)**.** The *closure* operation extends the context with additional items. Given a grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle$,

**Algorithm 10** Closure operation

```
1: procedure CLOSURE(Items)
2:     repeat
3:         for each [X → A • Y β, a] ∈ Items do
4:             for each Y → γ ∈ R do        ▷ All productions with head Y
5:                 for each b ∈ First(β a) do
6:                     Items ← Items ∪ [Y → •γ, b]
7:                 end for
8:             end for
9:         end for
10:    until Items unchanged
11: end procedure
```

*Remark.* If two or more $LR(1)$ items in the same FSA state have the same production with the dot in the same position, with the only difference being the lookahead terminal, then it can be written in short-hand as

$$\boxed{\begin{array}{l} X \to A \bullet Y\ \beta, \tau_1 \\ X \to A \bullet Y\ \beta, \cdots \\ X \to A \bullet Y\ \beta, \tau_n \end{array}} \Rightarrow \boxed{X \to A \bullet Y\ \beta, \tau_1 / \cdots / \tau_n} \tag{61}$$

**Definition 3.50** (Transition between $LR(1)$ FSA States)**.** The *Transition* procedure computes the next states reachable from a given state in a $LR(1)$ parsing FSA, given by

**Algorithm 11** Transition

```
1: procedure TRANSITION(state, y)
2:     Items ← ∅
3:     for each X → A • y β, b ∈ state do
4:         Items ← Items ∪ [X → A y • β, b]
5:     end for
6:     return CLOSURE(Items)
7: end procedure
```

Where $X \in \mathcal{N}$, $y \in (\mathcal{N} \cup \mathcal{T})$, $A, \beta \in (\mathcal{N} \cup \mathcal{T})^*$ and $b \in \mathcal{T}$.

**Definition 3.51** (Constructing $LR(1)$ Parsing FSA)**.** The *start context* needs to be constructed first by computing the closure of the start symbol

$$\text{Closure}(\{S' \to \bullet S, \$\}) \tag{62}$$

With each FSA state being a *closed* set of $LR(1)$ items, meaning the *closure* operation needs to be applied to each FSA state upon creation, then

1. The start state has
$$[S' \to \bullet S, \$] \tag{63}$$
2. A state is labelled with *reduce on* $X \to \alpha$ *on* $b$ iff it contains
$$[X \to \beta \bullet, b] \tag{64}$$
3. A state has an outwards *transition labelled* $y$ to a state which contains the items of Transition$(state, y)$ iff it contains
$$[X \to A \bullet \gamma\ \beta, b] \tag{65}$$

The overall algorithm is then given by

**Algorithm 12** $LR(1)$ Parsing FSA Construction

```
1: procedure BuildLR1FSA()
2:     workSet ← S_0 = Closure(S' → •S, $)
3:     Q ← ∅
4:     while workSet ≠ ∅ do
5:         S_s ← removeFirst(workSet)
6:         Q ← Q ∪ {S_s}
7:         for each [X → α • y β, t] ∈ Items(S_s) do        ▷ y ∈ (N ∪ T)
8:             S_t ← Transition(S_s, y)
9:             δ ← δ ∪ {S_s × y → S_t}
10:            if S_t ∉ Q then
11:                workSet ← workSet ∪ {S_t}
12:            end if
13:        end for
14:    end while
15:    F ← {q ∈ Q | ∃ i ∈ Items(q) : i = [Y → α•, $]}
16:    return ⟨Q, Σ = (N ∪ T), δ, F, S_0⟩
17: end procedure
```

**Definition 3.52** (Constructing $LR(1)$ Parse Table)**.** Given some grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle$ then to build the $LR(1)$ parse table,

1. Compute First$(n)$ for each non-terminal $n \in \mathcal{N}$.
2. Add new start state $S'$ giving the augmented grammar $G'$

$$G' = \langle \mathcal{N} \cup \{S'\}, \mathcal{T}, \mathcal{R} \cup \{S' \to S\}, S' \rangle \tag{66}$$

3. Run FSA construction algorithm on $G'$ and
   (a) Create $LR(1)$ item sets (FSA states).
   (b) Add transitions.
4. Convert FSA to parsing table.

The conversion between the FSA and the parsing table is given as follows. Given some grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle$ and given its parsing FSA $M = \langle \Sigma, \mathcal{Q}, \delta, F, q_0 \rangle$, then the parsing table has

- FSA states $\mathcal{Q}$ as rows.
- Terminals and non-terminals $(\mathcal{N} \cup \mathcal{T})$ as columns.
- *Shift*, *reduce*, *goto* or *accept* actions as table entries, where
  1. *Shift*: shift$(q)$
  2. *Goto*: goto$(q)$
  3. *Reduce*: reduce$(n)$ with $n$ numbering the production rules such

that

$$n \coloneqq I \xrightarrow{X} J \in \delta \tag{67}$$

More specifically, for the parse table $T$, with each table entry being a pair $\langle q \in \mathcal{Q}, \beta \in (\mathcal{N} \cup \mathcal{T}) \rangle$

$$T[q, \beta] \tag{68}$$

| Action | Corresponding Cell Entry |
|---|---|
| Shift | $\forall\, I \xrightarrow{\alpha} J \colon \alpha \in \mathcal{T} \implies T[I, \alpha] = \text{shift}(J)$ |
| Goto | $\forall\, I \xrightarrow{X} J \colon X \in \mathcal{N} \implies T[I, X] = \text{goto}(J)$ |
| Accept | $\forall\, I \in \mathcal{Q} \colon (S' \to S\bullet, \$) \in I \implies T[I, \$] = \text{accept}$ |
| Reduce | $\forall\, I \in \mathcal{Q} \colon (A \to g\bullet, y) \wedge n \coloneqq (A \to g) \implies T[I, y] = \text{reduce}(n)$ |

Figure 30: $LR(1)$ actions and corresponding parse table entry

*Remark.* The algorithm builds an NFA as there does not exist a transition $\delta$ for each symbol $\beta \in (\mathcal{N} \cup \mathcal{T})$. Although, the NFA does not contain any $\varepsilon$-transitions, meaning that it can be easily converted to a DFA by explicitly labelling unused transitions to a trash state, with any states that have accept annotations marked as final.

### 3.8.3 Shift/Reduce Conflicts

**Definition 3.53** (Shift/Reduce Conflict)**.** A *Shift/Reduce conflict* (S/R conflict) occurs when there exists a DFA state which contains both

$$[X \to A \bullet a\ \beta, b] \qquad\qquad [Y \to \Gamma\bullet, a] \tag{69}$$

Since on reading the input character $a$ it is equally valid to either

1. Shift into the state
$$[X \to A\ a \bullet \beta, b] \tag{70}$$
2. Reduce with
$$Y \to \Gamma \tag{71}$$

Such S/R conflicts tend to be due to ambiguities in the grammar. The default behaviour of many tools is to shift.

**Definition 3.54** (Resolve S/R Conflicts)**.** S/R conflicts can be resolved by either fixing the grammar, or introducing precedence on terminals which tend to cause the S/R conflicts. S/R conflicts are to resolved with a shift in tools like JFlex if any of the conditions below are satisfied

1. No precedence declared for rule or terminal.
2. Input terminal has higher precedence than rule.
3. Both have same precedence and are right associative.

**Definition 3.55** (Reduce/Reduce Conflicts)**.** A *Reduce/Reduce conflict* (R/R conflict) occurs if there exists a DFA state which contains both

$$[X \to A\bullet, a] \qquad\qquad [Y \to B\bullet, a] \tag{72}$$

Then on reading input character $a$ it is equally valid to reduce with either rules, and the parse cannot choose which.

**Definition 3.56** ($LR(1)$ Item Set Core)**.** The *core* of a set of $LR(1)$ item sets is the set of the first components without the lookahead tokens, e.g. given the set of $LR(1)$ items

$$\{[X \to \alpha \bullet \beta, b], [Y \to \gamma \bullet \delta, d]\} \tag{73}$$

Then the *core* of the item set is

$$\{X \to \alpha \bullet \beta, Y \to \gamma \bullet \delta\} \tag{74}$$

*Remark.* Parser generates are available, but many do not construct the DFA since the $LR(1)$ parsing FSA can yield many states even for simple languages. Note that many FSA states are similar, e.g. there may be two states $q_i$ and $q_j$ where

$$\boxed{q_i \colon X \to a\bullet, \$/+} \qquad\qquad \boxed{q_j \colon X \to a\bullet, )/+}$$

Figure 31: Similar $LR(1)$ states

Then they can be merged together since they have the same *core* and differing only in the lookahead token, giving

$$\boxed{q_k \colon X \to a\bullet, \$/+/)}$$

Figure 32: Merged $LR(1)$ state

**Definition 3.57** ($LALR(1)$ States)**.** Given some $LR(1)$ states

$$\begin{gathered}\{[X \to \alpha\bullet, a], [Y \to \beta\bullet, c]\} \\ \{[X \to \alpha\bullet, b], [Y \to \beta\bullet, d]\}\end{gathered} \tag{75}$$

Since they have the same *core*, they can be merged to give the $LALR(1)$ state

$$\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, c/d]\} \tag{76}$$

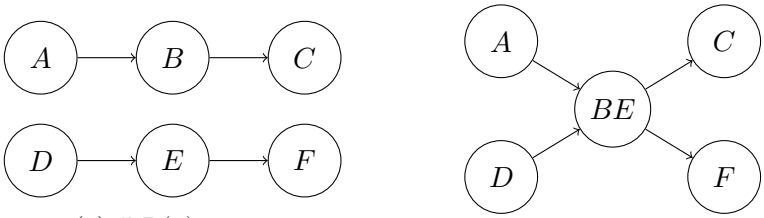
With $LALR(1)$ denoting *Look-Ahead* $LR(1)$.

**Definition 3.58** (Constructing $LALR(1)$ FSA)**.** Given a $LR(1)$ FSA and its states, it is then possible construct its corresponding $LALR(1)$ FSA with

**Algorithm 13** $LALR(1)$ FSA construction

```
1: procedure BuildLALR1States(LR1States)
2:     repeat
3:         ⟨q_i, q_j⟩ ← GetTwoStatesWithSameCore(LR1States)
4:         I_merged ← Items(q_i) ∪ Items(q_j)
5:         q_merged ← CreateState(I_merged)
6:         pred ← GetPrecedessors(q_i) ∪ GetPrecedessors(q_j)
7:         for each q_p ∈ pred do
8:             RedirectEdges(from : q_p, to : q_merged)
9:         end for
10:        succ ← GetSuccessors(q_i) ∪ GetSuccessors(q_j)
11:        for each q_s ∈ succ do
12:            RedirectEdges(from : q_merged, to : q_s)
13:        end for
14:    until all states have distinct core
15: end procedure
```

Graphically, given $LR(1)$ states $A, B, C, D, E, F$ with $B, E$ having the same cores needing to be merged, then a new $LALR(1)$ state $BE$ is created:




(a) $LR(1)$ states

(b) Merged $LALR(1)$ states

Figure 33: Merging $LR(1)$ states to produce $LALR(1)$ states

*Remark.* Merging $LR(1)$ states could introduce new Reduce/Reduce con-

flicts. For instance, given the $LR(1)$ states

$$\begin{array}{l} \{[X \to \alpha\bullet, a], [Y \to \beta\bullet, b]\} \\ \{[X \to \alpha\bullet, b], [Y \to \beta\bullet, a]\} \end{array} \tag{77}$$

That merge to give the $LALR(1)$ state

$$\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, a/b]\} \tag{78}$$

Which contains a new Reduce/Reduce conflict.

# 4 Semantic Analysis

## 4.1 Overview

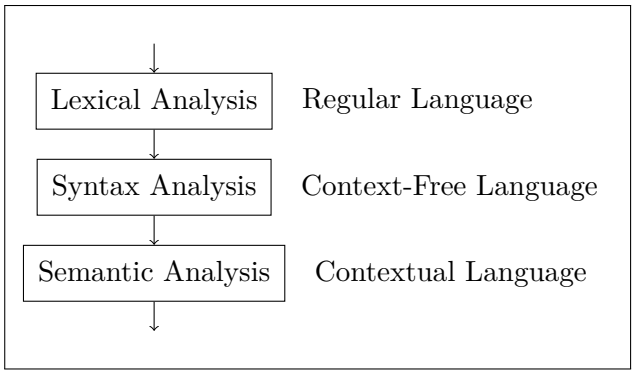



Figure 34: Languages used for different phases

*Remark.* Up to this point, recall that

- *Lexers* accept *regular* grammar (*Regex*).
- *Parsers* accept *context-free* grammar (*CFG*).
- But programs are *not context-free*.

That is,

- Programs which are *syntactically correct* are indeed *context-free*.
- But being *syntactically correct* does not guarantee that a program can correctly compile – the program must also be *semantically correct*.

Hence, there exists an important distinction between *Context-Free Grammars* (CFG) and *Context-Sensitive Grammar* (CSG):

- The *context* is *not* retained in *Context-Free Grammar*, allowing programs that are *syntactically correct* but make no *semantic* sense. Additionally, the *scope* of variables are not checked.

*Example.* For example, given the short method

```
public void methodOne()
{
    int x = 2;
    System.out.println(x.getDate());
    y++;
    int y;
    return x;
}
```

Figure 35: A program that is synatically correct but semantically wrong

The program is syntactically valid. However, there are multiple semantic errors:

- `x` is an integer, but an attempt was made to call `getDate()` on `x`.
- `y` is used before being declared.
- `methodOne` is declared to have no return value. However, `x` is returned in the last statement.

## 4.2 Context-Sensitive Grammar

**Definition 4.1** (Context-Sensitive Grammar (CSG))**.** The *Context-Sensitive Grammar* $G$ can be described by the 4-tuple

$$G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, S \rangle \tag{79}$$

Where

1. **Non-terminals**: $\mathcal{N}$.
2. **Terminals**: $\mathcal{T}$.
3. **Production Rules**: $\mathcal{R}$.
4. **Start Symbol**: $S$.

The *context-sensitive* property requires that

$$\forall\, r \in \mathcal{R} \colon r = \alpha A \beta \to \alpha \gamma \beta \tag{80}$$

That each production rule are of such form, where

- $A \in \mathcal{N}$ is a *non-terminal*.
- $\alpha, \beta \in (\mathcal{N} \cup \mathcal{T})^*$ are strings of *non-terminals* and *terminals*.

- $\gamma \in (\mathcal{N} \cup \mathcal{T})^+$ is a *non-empty* string of *non-terminals* and *terminals*.

$$\gamma \neq \emptyset \tag{81}$$

## 4.3 Semantic Analysis

Check outside the grammar for contextual correctness instead of trying to implement constraints within the language, for aspects such as

- Undefined variables.
- Type errors.
- Scope errors.
- Duplicate definitions.
- Inheritance relationships.

### 4.3.1 Contextual Constraints

Important *contextual constraints* include

1. **Scoping** rules: Checking area of effect of declarations.
2. **Typing** rules: Checking expressions have valid types.

### 4.3.2 Semantic Analysis Sub-phases

Note that this list is *non-exhaustive*.

- **Name Resolution**: Apply scoping rules to correspond identifier to declaration.
- **Type Checking**: Apply typing rules to infer type of expressions, and verify that the inferred type is compatible with the expected type.

## 4.4 Scoping Rules

**Definition 4.2** (Declaration)**.** A *declaration* associates information with an *identifier*.

There are two types of declarations:

- **Explicit Declaration**. (In Java)

```
int i;
```

- **Implicit Declaration**. (In Fortran)

```
i;
```

  In Fortran, some identifiers `i`, `j`, ..., `n` are implicitly assumed to be integers, unless this feature is turned off.

It is possible for the same identifier to have independent declarations in different parts of a program, differentiating via *scoping rules*.

**Definition 4.3** (Scoping Rules)**.** *Scoping rules* define which declaration an identifier resolves to in each part of the program.

**Definition 4.4** (Scope)**.** The *scope* of a declaration is the part of the program for which the declaration applies to.

### 4.4.1 Binding

**Definition 4.5** (Binding)**.** A *binding* is the association between an identifier and the thing it represents.

*Example.* Given the Java method

```
public boolean foo(int n, int m) {
    boolean tmp;
    if (n<m) tmp = true;
    else tmp = false;
    return tmp;
}
```

Figure 36: A Java method

Then its binding $\sigma_{\texttt{foo}}$ is

$$\sigma_{\texttt{foo}} = \{\texttt{foo} \mapsto \texttt{int} \times \texttt{int} \to \texttt{boolean}, \texttt{n} \mapsto \texttt{int}, \texttt{m} \mapsto \texttt{int}, \texttt{tmp} \mapsto \texttt{bool}\} \tag{82}$$

### 4.4.2 Static Scope

**Definition 4.6** (Static Scope (Lexical Scope))**.** *Static scope* is when scoping information is available at *compile time* – the scoping information is available within the source code.

Usually, the *most recent encounter* binds.

**Definition 4.7** (Static Scope Design)**.** Two *static scope* designs exist:

1. Global.
2. Local to function.

Allowing nested functions introduce further complications as each nested function can begin a new scope.

**Definition 4.8** (Closest Nested Scope Rule)**.** The *Closest Nested Scope* rule is defined as follows

- An identifier introduced in a declaration is known
  - in the scope where the identifier is declared in, or
  - in each internal scope that is nested after the declaration,
  - *unless* it is hidden by another declaration of the same identifier in one or more of the nested scope (known as "*name shadowing*").

*Example.* In the code fragment

```
let x = 4;                  // Outermost scope L1
if (x > 0)
{                           // Inner scope L2
    let x = 3;              // L2's x shadows /
                            //     hides x from L1
    if (x > 1)
    {                       // Innermost scope L3
        x = x + 1;          // L3's x modifies x from L2
        print(x)            // => 4
    }

    print(x);               // => 4 because L2's
                            //     x modified in L3
}
print(x);                   // => 4 L1's x
```

Figure 37: Closest nested scope in effect

**Definition 4.9** (Hole)**.** A *hole* in the scope of a binding occurs when the declaration of an identifier is *hidden* or *shadowed* by a nested declaration of the same identifier.

In figure 37, binding of `x` in $L_1$ is hidden by the declaration of the same name in the nested scope $L_2$.

**Definition 4.10** (Qualifier)**.** Some programming languages allow references to declarations in outer scopes that are shadowed by providing *qualifiers* or *scope resolution operators*.

*Example.* The *scope resolution operator* :: is provided in C++

```
#include <iostream>
int main(void)
{
    int cout = 42;
    cout << cout;
}
```

```
#include <iostream>
int main(void)
{
    int cout = 42;
    std::cout << cout;
}
```

Figure 38: Scope resolution operator in C++

### 4.4.3 Dynamic Scope

**Definition 4.11** (Dynamic Scope)**.** *Dynamic scope* refers to binding that is determined at run-time, depending on the flow of execution.

- Most recent declaration takes precedence.
- Binding expires when execution flow leaves lexical scope.

Generally execution flow cannot be predicted at compile time and so dynamic scope is usually implemented within *interpreted* languages.

*Remark.* *Static* scoping versus *dynamic* scoping can be analogous to *spatial* scoping versus *temporal* scoping:

- *Static* scoping looks for the lexically-closest declaration.
- *Dynamic* scoping looks for the most recent declaration with regard to the run time flow of execution.

## 4.5 Referencing Environment

**Definition 4.12** (Referencing Environment)**.** The sequence of scopes which needs to be examined in order for the current binding of a given identifier to be found is the *referencing environment*.

**Definition 4.13** (Shallow Binding (Late Binding, Dynamic Binding))**.** *Shallow binding* refers to delaying the creation of the reference environment until a routine is called.

**Definition 4.14** (Deep Binding (Static Binding, Early Binding))**.** *Deep binding* refers to creating the reference environment upon making the first reference.

## 4.6 Symbol Table

**Definition 4.15** (Symbol Table)**.** A *Symbol Table* stores binding information, helping to enforce scoping rules.

*Example.* Given the Java method

```
public boolean foo(int n, int m)
{
    boolean tmp;
    if (n < m) tmp = true;
    else tmp = false;
    return tmp;
}
```

(a) Java method

| Identifier | Kind | Type | Attributes |
|---|---|---|---|
| `foo` | method | `int` × `int` → `boolean` | ... |
| `n` | parameter | `int` | ... |
| `m` | parameter | `int` | ... |
| `tmp` | local variable | `boolean` | ... |

(b) Symbol table

Figure 39: Java code and its corresponding symbol table

### 4.6.1 Checking for Undefined Variables

**Definition 4.16** (Abstract Syntax Tree and Symbol Table)**.** Adding entries and querying entries between the Abtract Syntax Tree (AST) ad the Symbol Table involves two operations

1. **Insert**. A new binding is added to the currently active Symbol Table when a new identifier is declared.
2. **Lookup**. Traverse AST hierarchy from the current scope upwards.

*Remark.* A single Symbol Table, however, is insufficient for

- Nested scoping, when inner bindings hide outer bindings.
- Forward references, when identifiers can be referenced before they are declared.

**Definition 4.17** (Symbol Table Hierarchy)**.** A *hierarchy* of *Symbol Tables* can be used to implement nested scoping.

1. When entering a new nested scope, a new Symbol Table is created.
2. When exiting a nested scope, the Symbol Table for the nested scope is destroyed.

*Example.* For the Java code

```
public class Circle
{
    double PI = 3.141;

    public double circumference(double radius)
    {
        double diameter = 2 * radius;
        double circ = PI * diameter;
        return circ;
    }

    public double area(double radius)
    {
        double a = PI * radius * radius;
        return a;
    }
}
```

Figure 40: Nested scoping and hierarchy of Symbol Tables

There are three Symbol Tables

$$\sigma_0 = \{\texttt{PI} \mapsto \texttt{double}, \texttt{circumference} \mapsto \texttt{double} \to \texttt{double}, \texttt{area} \mapsto \texttt{double} \to \texttt{double}\}$$
$$\sigma_1 = \sigma_0 \cup \{\texttt{diameter} \mapsto \texttt{double}, \texttt{circ} \mapsto \texttt{double}\}$$
$$\sigma_2 = \sigma_0 \cup \{\texttt{a} \mapsto \texttt{double}\}$$

Note that

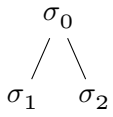



Figure 41: Hierarchical Symbol Table

### 4.6.2 Finding Active Declaration of Identifier in Symbol Table Hierarchy

Finding the active declaration of identifier in Symbol Table involves

- Being from current scope.
- Keep going up the hierarchy until the identifier is found.
- Never traverse downwards.
- If the identifier is not found even at the root of the hierarchy, the identifier is being used without being defined, which is an error.

*Remark.* Hierarchical Symbol Tables also have the benefit of resolving name collisions – the deepest nested scope with the identifier takes precedence.

### 4.6.3 Symbol Table Implementation

A symbol table could be implemented via a *Hash Table*, given that the *hashing function* has good spread and minimal primary or secondary clustering.

To visit the hierarchical Abstract Syntax Tree, one may utilize the *Visitor* design pattern, which decouples the tree's structural representation from any algorithms traversing it.

## 4.7 Type Checking

**Definition 4.18** (Type)**.** A *type* refers to a set of *values* with some *operations* defined on it.

**Definition 4.19** (Type System)**.** A *type system* consists of *type expressions* and corresponding *type-checking rules*.

**Definition 4.20** (Type Expression)**.** A *type expression* is one of the possible types in a program.

*Example.* For instance, a *type expression* in Java could be

- `int`,
- `boolean`,
- or a user-defined type such as `Bag<Integer>`.

**Definition 4.21** (Type Rules)**.** *Type rules* are semantic constraints for which program constructs must satisfy in order to successfully compile.

### 4.7.1 Type Expressions

*Type Expressions* consists of *primitive types* and *compound types*.

- **Primitive Types**
  1. `int`
  2. `double`
  3. `boolean`
  4. `long`
  5. `float`
  6. `double`
  7. `char`
  8. `short`
- **Compound Types**
  *Compound types* are constructed from combining primitive types. For example,
  1. Pointer types. `char *`
  2. Struct types. `struct point  int x; int y; ;`
  3. Array types. `char[]`
  4. Generic types. `List<Integer>`

### 4.7.2 Type Information in Symbol Table

The Symbol Table usually also includes type information

- Bindings between identifier and type.
- Bindings between formal parameter and type.
- Bindings between method name, parameters and return type.
- Bindings between class name, class variables and method declarations.

### 4.7.3 Type-Checking Rules

*Example.* A type-checking rule could require that *the type of the left-hand side of an assignment must have the same type as the right-hand side of the assignment.*

For a very simple expression `int a = 2 * 2;`, it fulfills the type-checking rule because

$$\dfrac{\dfrac{\Gamma \vdash \texttt{int a}}{\Gamma \vdash \texttt{a}\colon \mathit{int}} \quad \dfrac{\Gamma \vdash \texttt{2}\colon \mathit{int}}{\Gamma \vdash \texttt{2} * \texttt{2}\colon \mathit{int}}}{\Gamma \vdash \texttt{int a} = \texttt{2} * \texttt{2};\colon \mathit{int}}$$

Figure 42: Assignment type-checking rule

### 4.7.4 Type-Checking Implementation

There are two steps to *type-checking*

1. **Symbol Table Construction**.
   - Visit AST.
   - Store type information in Symbol Table(s).
2. **Scope and Type Checks**.
   - Visit AST and access Symbol Table(s).
   - Verify program conforms with scope and type rules.

**Definition 4.22** (Types and Symbol Table Construction)**.** Each *type expression* can be represented as a class, with the Symbol Table binding each identifier to a specific *type object*. Semantic analysis can then convert *type expressions* to corresponding *type objects*.

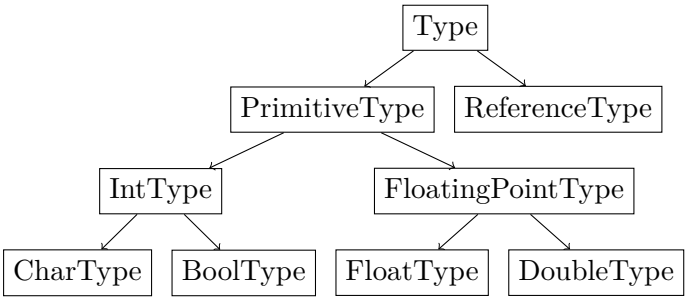



Figure 43: Type Objects

Alternatively, each type can also be represented by a string with a predifined format.

$$\texttt{int} \mapsto \texttt{"I;"}$$
$$\texttt{int[ ]} \mapsto \texttt{"[I;"}$$

Figure 44: Type String

**Definition 4.23** (Type Comparison Implementation)**.** With the two possible type implementations, *type comparison* can also be implemented with

- Implement `T1.equals(T2)` object comparison method if types are implemented with objects.
- String comparison if types are implemented with strings.

**Definition 4.24** (Type Checking Implementation)**.** *Type Checking* can be implemented via a Visitor to the AST hierarchy; type rules are checked during the recursive traversal.

### 4.7.5 Type Error Reporting

**Definition 4.25** (Type Error Handling)**.** When a *type-checking error* is discovered during analysis, the error should be reported and the semantic analysis should continue.

Correct types may have to be inferred to avoid *cascading failures*.

### 4.7.6 Simple Semantic Analyzer

**Definition 4.26** (Simple Semantic Analyzer)**.** A simple semantic analyzer can be implemented via two phases

**Algorithm 14** Simple Semantic Analyzer

```
 1: procedure DoSemanticAnalysis(Program)
 2:     for all DeclarativeRegion ∈ Program do
 3:         ProcessDeclarations(DeclarativeRegion)
 4:         ProcessStatements(DeclarativeRegion)
 5:     end for
 6:     for all DeclarativeRegion ∈ Program do
 7:         InferTypes(DeclarativeRegion)
 8:         FindTypeErrors(DeclarativeRegion)
 9:     end for
10: end procedure
11: procedure ProcessDeclarations(DeclarativeRegion)
12:     for all Declaration ∈ DeclarativeRegion do
13:         SymbolTable.AddEntry(Declaration)
14:         if IsDuplicateDeclaration(Declaration) then
15:             ReportTypeError(Declaration)
16:         end if
17:     end for
18: end procedure
19: procedure ProcessStatements(DeclarativeRegion)
20:     for all Statement ∈ DeclarativeRegion do
21:         FindUndeclaredVariables(Statement)
22:         UpdateAstIdNodes(SymbolTable)
23:     end for
24: end procedure
```

# 5 Intermediate Representation

**Definition 5.1** (Intermediate Representation (IR))**.** *Intermediate Representation* (IR) acts as an adapter between the *source language* and the *target language*. This helps to decouple the compiler's front-end and back-end, and allows optimization to be source language-agnostic.

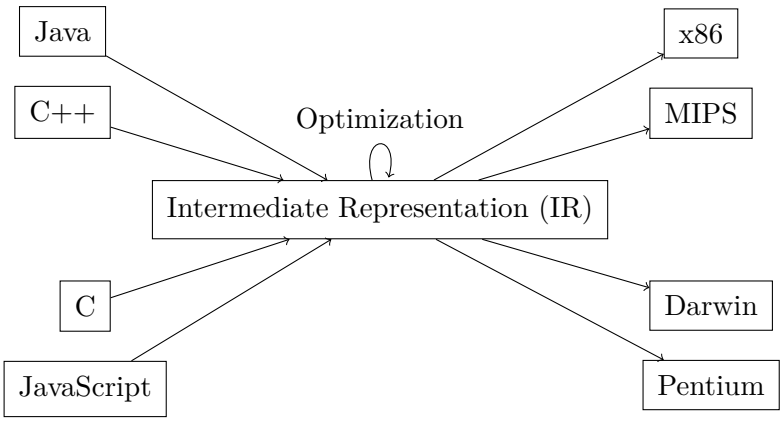



Figure 45: Intermediate representation between source language and destination language.

**Definition 5.2** (High-level IR vs Low-level IR)**.** *High-level* IR and *low-level* IR differs in the amount of source information they retain.

| High-level IR | Low-level IR |
|---|---|
| Retains more source code information. Allows compiler to extract source code intention more easily. | Retains less source code information. Allows compiler to generate code for hardware platform more easily. |

Figure 46: High-level IR vs Low-level IR

**Definition 5.3** (Properties of Good IR)**.** A *good IR* has the properties

- Accuracy.
- Decoupled from source and target language.
- Easy to produce.
- Easy to translate.
- Clear, unambiguous language constructs.
- No convenience features.

**Definition 5.4** (IR Categories)**.** There exists three major categories of IR

| Structural IR | Hybrid IR | Linear IR |
|---|---|---|
| Graphically structured | Both graphical and linear code | Pseudo-machine code |
| Used heavily in transpilers and verification tools | | Different abstraction levels |
| Usually large | | Easier to rearrange |
| e.g.: Trees, DAGs | e.g. Control-Flow Graph | e.g.: Three-Address Code, Stack Machine Code |

Figure 47: Categories of Intermediate Representation

## 5.1 Linear Intermediate Representation

**Definition 5.5** (Linear IR)**.** A *Linear Intermediate Representation* (LIR) is an instruction set for an abstract machine.

- Intermediate between Decorated AST and Machine Code.
- Allows language-agnostic optimizations.

It is ideal for optimization since

- Linear AST has a *narrow interface* of limited instructions.
- This allows optimizations, re-targeting and translating to become easier.

Examples include

- GCC Gimple
- LLVM IR
- Java Bytecode

**Definition 5.6** (Abstract Machines)**.** LIRs are instructions for *abstract machines*; two types of abstract machines are

1. **Register Machines**.
2. **Stack Machines**.

**Definition 5.7** (Register Machines)**.** A *register-based abstract machine* uses *identifier-based temporary variables* (which later gets mapped to real registers during register-selection phase).

*Example.* A very well known example of register machine is MIPS, a corresponding register-based IR is *Three-Address Code* (TAC).

**Definition 5.8** (Stack Machines)**.** A *stack machine* performs all operations based on *stack*. It has two memory regions

1. **Instruction Memory**: which contains IR code.
2. **Data Memory**: which stores values, referenced via id or address.

*Example. Java Bytecode* is a stack-based IR which runs on the *Java Virtual Machine*, a stack-based abstract machine.

## 5.2 Register Machines and Three-Address Code

**Definition 5.9** (Three-Address Code)**.** *Three-Address Code* (IR) is a register-based IR which has at most three addresses per instruction. Each instruction has the form

$$\text{OP } a_1 \ a_2 \ a_3 \tag{83}$$

Where $a_1, a_2, a_3$ are arguments to the operation. The number of operand registers may vary based on the arity of the operation.

Note that compound expressions may have to be partitioned into small consecutive steps (unary, binary expressions) using temporary registers to hold intermediate values due to the instruction size constraint.

**Definition 5.10** (Three-Address Code Statements)**.** Let us define a custom-flavoured Three-Address Code (TAC).

| Binary | | | | | | | Control Flow | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Arithmetic | Logical | Comparisons | Unary | Copy | Array Access | Field Access | Labelling | Unconditional Jump | Conditional Jump | Function Calls |
| ADD | AND | EQ | MINUS | a = b | a = b[i] | a = b.f | label L: | jump L | tjump a L | param p |
| SUB | OR | NEQ | NEG | | a[i] = b | a.f = b | | | fjump a L | call f |
| MUL | XOR | LE | | | | | | | | a = call f |
| DIV | | LEQ | | | | | | | | return |
| MOD | | GE | | | | | | | | return y |
| | | GEQ | | | | | | | | |

Table 1: Three-Address Code Instructions

**Definition 5.11** (TAC Operands)**.** Each *operand* in a TAC instruction can be

1. Program variables
2. Constants
3. Temporary Variables

**Definition 5.12** (Temporary Variables)**.** *Temporary variables* are new locations which are used to store intermediate values.

## 5.3 Translation

**Definition 5.13** (Translation)**.** Translation between languages requires dealing with *nested constructs*. This derives the need for *inductive*, *template-based* translation

- Start from AST root node.
- Define translation rules for each node type.
- Recursively apply templates down AST.

**Definition 5.14** (Translation Notation)**.** If $e$ is a high-level construct, then

- $T[e]$ is the sequence of low-level translated instructions.
- If $e$ is an *expression*, then $e$ represents a *value*;
  - $t = T[e]$ means the result of $T[e]$ is stored at $t$.
- If $e$ is a *variable*, then $t = T[e]$ is the *copy instruction* $t = e$.

**Definition 5.15** (Translating Expressions)**.** There are generic templates:

1. **Unary operation**.

$$t = T[\texttt{OP}\ e]$$

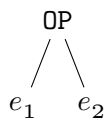
It is translated to

$$\begin{array}{c} \texttt{OP} \\ | \\ e \end{array} \qquad\qquad \begin{aligned} t_1 &= T[e] \\ t &= \texttt{OP}\ t_1 \end{aligned}$$

2. **Binary operation**.

$$t = T[e_1 \texttt{OP}\ e_2]$$

It is translated to

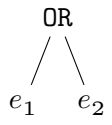



$$t_1 = T[e_1]$$
$$t_2 = T[e_2]$$
$$t = t_1 \text{OP } t_2$$

3. **Short-circuiting** `OR`.

$$t = T[e_1 \texttt{OR } e_2]$$

It is translated to



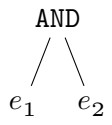

$t = T[e_1]$
`tjump` $t$ `END`
$t = T[e_2]$
`label END`

Doing so allows us to avoid evaluating $e_2$ supposing $e_1$ is true since $\top \vee q \equiv t$. This is espeially beneficial if $e_2$ involves some expensive calculations or I/O operations.

4. **Short-circuiting** `AND`.

$$t = T[e_1 \texttt{AND } e_2]$$

It is translated to



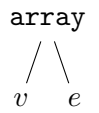

$t = T[e_1]$
`fjump` $t$ END
$t = T[e_2]$
`label END`

This can be done because $\bot \wedge q \equiv \bot$.

5. **Array access**.

$$t = T[v[e]]$$

It is translated to

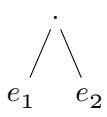



$$t_1 = T[e]$$
$$t = v[t_1]$$

6. **Field access**.

$$t = T[e_1.e_2]$$

It is translated to

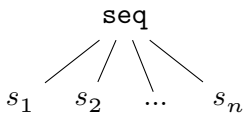



$$t_1 = T[e_1]$$
$$t_2 = T[e_2]$$
$$t = t_1.t_2$$

7. **Statements list**.

$$T[s_1; s_2; ...; s_n]$$

It is translated to

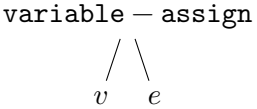



$$T[s_1]$$
$$T[s_2]$$
$$...$$
$$T[s_n]$$

8. **Variable assignment**.

$$T[v = e]$$

It is translated to

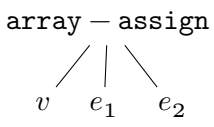



$$v = T[e]$$

9. **Array assignment**.

$$T[v[e_1] = e_2]$$

It is translated to




$$\begin{aligned} t_1 &= T[e_1] \\ t_2 &= T[e2_2] \\ v[t_1] &= t_2 \end{aligned}$$

10. **If-then**.

$$T[\texttt{if } (e) \texttt{ then } s]$$

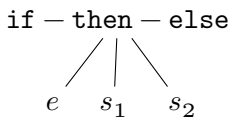
It is translated to


$$\begin{aligned} &t_1 = T[e] \\ &t_2 = \texttt{NOT } t_1 \\ &\texttt{tjump } t_2 \texttt{ END} \\ &T[s] \\ &\texttt{label END} \end{aligned}$$

11. **If-then-else**.

$$T[\texttt{if } (e) \texttt{ then } s_1 \texttt{ else } s_2]$$

It is translated to




$t_1 = T[e]$
`tjump` $t_1$ `LEFT`
$T[s_2]$
`jump END`
`label LEFT`
$T[s_1]$
`label END`

12. **While**.

$$T[\texttt{while}\ (e)\ \{s\}]$$

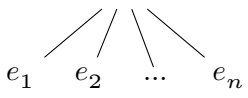
It is translated to


`label TEST`
$t_1 = T[e]$
`fjump` $t_1$ `END`
$T[s]$
`jump TEST`
`label END`

13. **Function call**.

$$T[f(e_1, e_2, \ldots, e_n)]$$

It is translated to

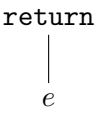



```
param T[e_1]
param T[e_2]
...
param T[e_n]
call f
```

14. **Return statement**.

$$T[\texttt{return}\ e]$$

It is translated to



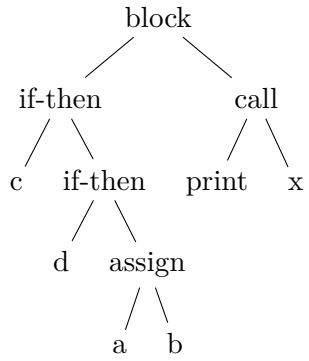

```
t = T[e]
return t;
```

**Definition 5.16** (TAC Recursive Translation)**.** With the previously defined templates in mind, it is then possible to recursively apply the templates in order to translate nested constructs.

*Example.* Given the code

```
if (c)
{
    if (d)
    {
        a = b
    }
}
print(x)
```

Which generates the AST




$t_1 = T[c]$
`tjump` $t_1$ `OUTER_TRUE`
`param` $x$
`call` *print*
`jump OUTER_END`
`label OUTER_TRUE`
$t_2 = T[d]$
`tjump` $t_2$ `INNER_TRUE`
`jump INNER_END`
`label INNER_TRUE`
$t_3 = T[b]$
$t_4 = T[a]$
$t_4 = t_3$
`label INNER_END`
`label OUTER_END`

## 5.4 Implementing Three-Address Code

**Definition 5.17** (TAC Implementation)**.** *Three-Address Code* (TAC) can be implemented by

- **Quadruples**
- **Triples**
- **Indirect Triples**

**Definition 5.18** (Quadruples)**.** Each instruction in *Quadruples* consist of an operator, two arguments field, as well as an result field.

- *Operator*: type of instruction.
- *Arguments*: at most two, possibly empty.
- *Result*: address to store computed result.

Note that `jump` instructions place label address in the result field.

For example, the code fragment `a = -c + b` could be represented as

| Operator | Argument 1 | Argument 2 | Result |
|---|---|---|---|
| `UMINUS` | $c$ | | $t_1$ |
| `PLUS` | $b$ | $t_1$ | $t_2$ |
| `ASSIGN` | $t_2$ | | $a$ |

Figure 63: Example Quadruples

**Definition 5.19** (Triples)**.** In *Triples*, results are used as temporaries.

For example

| Address | Operator | Argument 1 | Argument 2 |
|---|---|---|---|
| 20 | `UMINUS` | $c$ | |
| 21 | `PLUS` | $b$ | [20] |
| 22 | `ASSIGN` | $a$ | [21] |

Figure 64: Example Triples

*Remark.* Note that Triples suffer from the problem that since optimization tends to need to move code around, operation addresses are not preserved and will need to be updated.

**Definition 5.20** (Indirect Triples)**.** *Indirect Triples* improve upon Triples by adding an additional address-resolution (pointer) table which allows instruction to refer to "virtual" addresses instead of fixed absolute addresses.

For example,

| Pointer | Address |
|---|---|
| [20] | 45 |
| [21] | 46 |
| [22] | 47 |

| Address | Operator | Argument 1 | Argument 2 |
|---|---|---|---|
| 20 | `UMINUS` | $c$ | |
| 21 | `PLUS` | $b$ | [20] |
| 22 | `ASSIGN` | $a$ | [21] |

Figure 65: Example Indirect Triples

*Remark.* Each type of TAC implementation has its own benefits and caveats.

| Quadruples | Triples | Indirect Triples |
|---|---|---|
| Easier to optimize | Need to update references upon moving instructions | Need to reorder statements but no changes to references |
| Could generate excessive temporaries | | Saves space if temporaries are reused |

Figure 66: Benefits and issues of the different TAC implementations

### 5.5 Stack Machines

**Definition 5.21** (Stack Machine)**.** A *Stack Machine* utilizes a *stack* to perform operations which stores operands and results.

**Definition 5.22** (Abstract Stack Machine)**.** An *Abstract Stack Machine* (ASM) has the properties

- Operands and results stored in stack.
- No need for register allocation.
- Detailed run-time organization can be delayed.

**Definition 5.23** (Benefits and Drawbacks of Stack-based IRs)**.** Stack-Based IRs have the following benefits and drawbacks:

- Benefits
  - Generate more compact IR.
  - Simpler compiler because independent instructions.
  - Simpler interpreter due to centralized memory access and less variation in instruction types.
- Drawbacks
  - More memory references are needed since variables need to be pushed back on to the stack.
  - Difficult to factor out common sub-expressions.
  - Instruction order tied to storage of temporary values making moving instructions around difficult, making optimization more difficult.

## 6 Optimization

### 6.1 Basics

**Definition 6.1** (Ideal Characteristics of Target Code)**.** The *ideal target code* has the characteristics

- Semantics-preserving
- Efficient
- Small memory footprint
- Small space usage

**Definition 6.2** (Semantic-Preserving)**.** *Optimization* as an *improvement* requires that the semantics of the source code is strictly *preserved*; only space and/or time requirements are improved in the target code.

**Definition 6.3** (Side-Effects)**.** A function $f$ has *side-effects* if it

- Modifies external state, or
- Interacts with calling functions or the outside world, or
- Produces different output given the same input.

If a function $f$ does *not* have *side-effects*, then $f$ is considered a *pure* function.

*Remark.* Side-effects need to be carefully considered when trying to optimize function calls.

For instance, it is only safe to rewrite

```
f(x) + f(x)
```

With

```
2 * f(x)
```

If and only if $f$ is a *pure* function.

**Definition 6.4** (Optimization at Different Levels)**.** Different information is available for optimization at different levels

- High level
  - Type information
- Intermediate level
  - Control Flow Analysis
  - Data Flow Analysis
  - Index calculation
  - Bounds checking
- Low level
  - Target code selection
  - Register allocation

*Remark.* There is a classic conflict in optimization: the trade-off between *time* and *space*. But it is generally the case that the fewer instructions there are, the faster the execution.

**Definition 6.5** (Common Peephole Optimizations)**.** There are numerous optimizations which are common at the source, AST or IR level:

- Common Subexpression Elimination
- Algebraic Identities
- Dead Code Elimination
- Constant Folding
- Strength Reduction
- Inlining
- Copy Propagation
- Loop optimizations
  - Unrolling
  - Fusion / Fission
  - Code Motion
  - Tiling
  - Inversion
  - Interchange
  - Unswitching

## 6.2 Common Subexpression Elimination

**Definition 6.6** (Common Subexpression)**.** An expression $E$ is termed a *common subexpression* if it has been previously computed, and the variables involved in the subexpression has not changed since the computation.

**Definition 6.7** (Common Subexpression Elimination)**.** Introduce a new temporary, storing the value computed by the common subexpression.

*Example.* For instance, given the code fragment

```
int a = x + 42 * d;
int b = y + 42 * d;
```

The common subexpression `42 * d` can be factored out to give

```
int t1 = 42 * d;
int a = x + t1;
int b = y + t1;
```

## 6.3 Algebraic Identities

**Definition 6.8** (Algebraic Identities Optimization)**.** Simple algebraic identities such as

$$x \times 0 = 0 \tag{84}$$
$$x \times 1 = 1 \tag{85}$$

Can be used to simplify trivial expressions such as

- `x * 0` to `0`
- `x * 1` to `x`

Additionally, properties like commutatity of integer addition can help to factor out common subexpressions.

However, it must also be noted that computations such as floating point arithmetic do not necessarily follow properties satisfied by real numbers, especially involving floating point precision.

*Example.* Given the code fragment

```
int x = (2 + z) * j;
int y = (z + 2) * k;
```

Given that `z` is an integer, meaning `2 + z` and `z + 2` are both integer expressions, the two subexpressions are commutative and yield identical results, meaning that it is safe to factor them out to become

```
int t1 = 2 + z;
int x = t1 * j;
int y = t1 * k;
```

## 6.4 Dead Code Elimination

**Definition 6.9** (Dead Code Elimination)**.** With *control flow analysis* and *data flow analysis*, it can be determined that some parts of the code is never reached and never execute and can safely be removed.

*Example.* The body of the `if` statement in the code fragment below is never reached since the condition is always false, then the entire `if` block can be safety discarded.

```
int a = 2;
if (a > 3)
{
    System.out.println("Hello World!");
}
```

## 6.5 Constant Folding

**Definition 6.10** (Constant Folding Optimization)**.** Some subexpressions can be eagerly evaluated at compile time to reduce the number of instructions required, especially expressions involving numeric constants.

It is important that semantics must be respectived, especially with respect to precision; that is,

- Compile-time precision must match run-time precision.

*Example.* Expressions such as

```
int PI = 3.14159;
int circumference = 2 * PI * r;
```

Can be folded by precomputing the expression `2 * PI`, giving

```
int PI = 3.14159;
int circumference = 6.28318 * r;
```

Note that if `PI` has no later references, it can also be safety discarded by running a Dead Code Elimination pass after the Constant Folding pass.

## 6.6 Strength Reduction

**Definition 6.11** (Strength Reduction)**.** *Strength Reduction* involves substituting expression operations with semantically-equivalent but faster ones.

In the order of slowest to fastest,

- Division, Modulo
- Multiplication
- Addition, Subtraction, Shifts
- Comparisons

That is, divisions and modulo operations are more expensive (slower) than multiplication, and so on.

*Example.* Expressions such as

```
5 * x
```

May be substituted with

```
x << 2 + x
```

Since multiplication is more expensive than bitwise shift and addition.

**Definition 6.12** (Induction Variables)**.** A variable which is modified with each iteration by some common pattern may be subject to strength reduction via *induction*.

*Example.* In the loop

```
int i = 10;
int t = 0;
while (i > 0)
{
    i--;
    t = 4 * i;
    a[i] = b[t];
}
```

The expression assigned to $t$ can be substituted with

```
int i = 10;
int t = 4 * i;
while (i > 0)
{
    i--;
    t = t - 4;
    a[i] = b[t];
}
```

## 6.7 Inlining

**Definition 6.13** (Inlining)**.** *Inlining* intends to avoid function calls which are expensive by replacing the function invocation via copying the function body to the call site and initializing with parameters.

Note that

- Inlined function needs to be sufficiently small, or target code may suffer from code bloat.
- Recursive functions need to be handled carefully.

*Example.* Given the code fragment

```c
#include <stdio.h>

int foo(int a, int b)
{
    return (a + b) * b;
}

int main(void)
{
    int x = 0;
    for (int a = 0; a < 10; a++)
        for (int b = 0; b < 20; b++)
            x += foo(a, b);

    printf("%d\n", x);

    return 0;
}
```

Then the call to `foo(a, b)` can be replaced with its body to give

```c
#include <stdio.h>

int main(void)
{
    int x = 0;
    for (int a = 0; a < 10; a++)
        for (int b = 0; b < 20; b++)
            x += (a + b) * b;

    printf("%d\n", x);

    return 0;
```

```
}
```

Note that `foo(int, int)` could be eliminated after an additional Dead Code Elimination pass since it is no longer needed.

## 6.8 Loop Optimizations

### 6.8.1 Loop Unrolling

**Definition 6.14** (Loop Unrolling)**.** *Loop Unrolling* refers to repeat the loop body given a known number of iterations.

- Benefits
  - Less jumps.
  - Less condition checking and no need for index variables.
- Drawbacks
  - Increase in code size.
  - May be too large for L1 instruction cache.

*Example.* The loop

```
for (int i = 0; i < 3; i++)
    printf("Hello World!\n");
```

Could be unrolled to

```
printf("Hello World!\n");
printf("Hello World!\n");
printf("Hello World!\n");
```

### 6.8.2 Fusion / Fission

**Definition 6.15** (Loop Fusion / Fission)**.** *Loop Fusion* and *Loop Fission* refers to splitting or combining loops with the same index range.

- Loop fusion gives less jumps.
- Loop fission allows multiple processors to execute each loop.

*Example.* The following loops demonstrate fusion and fission:

```c
int a[100];
int b[100];
for (int i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```

(a) Fusion

```c
int a[100];
int b[100];
for (int i = 0; i < 100; i++)
{
    a[i] = 1;
}
for (int i = 0; i < 100; i++)
{
    b[i] = 2;
}
```

(b) Fission

Figure 67: Loop Fusion and Loop Fission. Going from left version to right version is *fission*, while going from right version to left version is *fusion*.

### 6.8.3 Code Motion

**Definition 6.16** (Code Motion)**.** *Code Motion* moves loop-invariant (loop-independent) code outside the loop.

- Benefits
  - Less instructions.
  - Constants may be stored in registers.
- Drawbacks
  - Can cause register spill.

*Example.* For instance, in

```c
#define N 100
while (i <= N - 1)
{
    // ...
}
```

The predicate check involves computing `N - 1` each iteration, which is unnecessary and can be moved out to become

```c
#define N 100
int max = N - 1;
```

```
while (i <= max)
{
    // ...
}
```

### 6.8.4 Tiling

**Definition 6.17** (Tiling)**.** *Tiling* creates smaller loops around blocks of data to make sure that the data fits within CPU caches.

*Example.* For the loop

```
for (int i = 0; i < N; i++)
{
    // ...
}
```

A nested loop can be introduced which allows finer partitions

```
for (int i = 0; i < N; i += B)
{
    for (int j = i; j < min(N, j+b); i++) {
        // ...
    }
}
```

### 6.8.5 Loop Inversion

**Definition 6.18** (Loop Inversion)**.** *Inversion* converts a `while` loop into an equivalent `do` loop in order to

- Reduce number of jumps, implying fewer pipeline stalls.

*Example.* For instance, given the `while` loop

```
int i = 0;
while (i < 100)
{
    // ...
```

```
    i++;
}
```

Is equivalently

```
int i = 0;
if (i < 100)
{
    do {
        // ...
        i++:
    } while (i < 100);
}
```

### 6.8.6 Principles of Locality

**Definition 6.19** (Temporal Locality)**.** *Temporal Locality* refers to reusing specific data or resource within a short period of time.

**Definition 6.20** (Spatial Locality)**.** *Spatial Locality* refers to using data between close storage locations, e.g. array elements.

### 6.8.7 Interchange

**Definition 6.21** (Interchange)**.** Outer and inner loops may be interchanged for better *locality of reference*.

*Example.* The outer loop performing less iterations could be swapped with the inner loop in

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 20; j++)
        x[i][j] = i + j;
```

To become

```
for (int j = 0; j < 20; j++)
    for (int i = 0; i < 10; i++)
        x[i][j] = i + j;
```

### 6.8.8 Unswitching

**Definition 6.22** (Unswitching)**.** *Unswitching* extracts conditional inside a loop by copying the loop for true and false branches.

- Helps with parallelism.
- Allows optimizing loops independently.

*Example.* Given the loop

```
for (int i = 0; i < 100; i++)
{
    doSomething();
    if (predicate)
    {
        doSomethingElse();
    }
}
```

Then it becomes

```
if (predicate)
{
    for (int i = 0; i < 100; i++)
    {
        doSomething();
        doSomethingElse();
    }
}
else
{
    for (int i = 0; i < 100; i++)
    {
        doSomething();
    }
}
```

### 6.8.9 Peephole Optimization vs Super Optimization

**Definition 6.23** (Peephole Optimization)**.** Aforementioned optimizations are all *peephole* optimizations in that they

- Operate at limited scope.
- Use pattern matching against predefined rules.
- Keeps trying to apply optimizations until nothing changes.

Note that there may not necessarily be a *correct* order to apply different peephole optimizations, and that one peephole optimization may allow another to function.

**Definition 6.24** (Super Optimization)**.** Given a program written in machine language, exhaustive search is performed to find the smallest equivalent program.

- Can be used to optimize critical inner loops.
- Can be used to find peephole optimizer schemes.

# 7 Run Time Organization

## 7.1 Basics

**Definition 7.1** (Machine Code)**.** *Machine Code* is a set of *instructions* executed directly by CPUs.

**Definition 7.2** (Assembly Code)**.** *Assembly* is a thin layer of abstraction on top of machine code, making it human readable.

- Can translate into object code or machine code via an *Assembler*.

**Definition 7.3** (Run Time Organization)**.** The execution of a user program is initially controlled by the Operating System (OS). When the program is launched,

- OS allocates memory for the program.
- Code is loaded into allocated memory.
- OS jumps to entry point (i.e. `main`).

**Definition 7.4** (C Program Memory Layout)**.** A C program has the memory layout illustrated as follows:

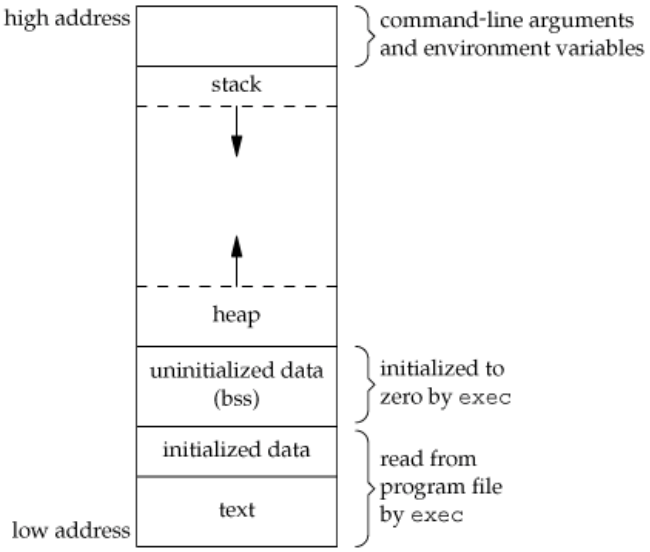



Figure 68: C program memory layout [1]

## 7.2 Procedure Activation

**Definition 7.5** (Procedure Invocation / Activation)**.** A procedure $P$ is *invoked* or *activated* when it is called, assuming

1. Sequential execution.
2. Return address is immediately after call site.

**Definition 7.6** (Lifetime of Activation of Procedure $P$)**.** A procedure $P$ has the *lifetime* of

1. All steps to execute $P$
2. All steps within procedures called by $P$

**Definition 7.7** (Lifetime of Variable $x$)**.** A variable $x$ has the lifetime of the part of the execution for which $x$ is well defined dynamically. Note that this is not equivalent to the scope of variable $x$.

**Definition 7.8** (Activation Tree)**.** If procedure $P$ calls procedure $Q$, then $Q$ must return before $P$ is returned.

- The lifetimes of procedure activations are hence nested and recursive, allowing them to be represented by trees.

*Example.* Given the procedures

```
class C
{
    public void foo()
    {
        // ...
        bar();
    }

    public void bar()
    {
        // ...
    }

    public static void main(String args[])
    {
        foo();
        bar();
    }
}
```

Then the activation tree is given by

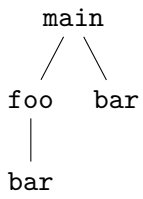



Figure 69: Corresponding activation tree

*Remark.* Since the activations are nested, a *stack* could be utilized to track the currently active procedures.

**Definition 7.9** (Activation Record)**.** The information required to handle a procedure's activation is called an *Activation Record* (AR) or a *Stack Frame*.

A generic activation record contains:

1. **Return values**.
2. **Parameters**.
3. **Control link** to previous activation record.
4. **Return address** to caller.

5. **Temporaries**.
6. **Local variables**.
7. **Access link** to additional data.

*Remark.* If a procedure $A$ calls procedure $B$, then the called procedure $B$'s stack frame consists of information from both $A$ and $B$. For instance, $B$'s stack frame need to keep track of the return address within $A$ in order to return control from $B$ to $A$ after $B$ completes its execution.

**Definition 7.10** (Global Variable)**.** A *global variable* is a variable such that all references to it point to the same object, meaning that it cannot be stored on an activation record. This requires a fixed address to be assigned by the compiler.

**Definition 7.11** (Dynamically Allocated Data)**.** Any data which needs to have a lifetime longer than the procedure which created it cannot be kept within the creation procedure's activation record. This is resolved by allocating memory for storing the data on the *Heap*.

*Remark.* Then, with reference to figure 68,

- *Text* area contains the object code, fixed size and readonly.
- *Initialized data* area contains data with fixed addresses, possibly writable and readable.
- *Stack* contains activation records for each active procedure. Each stack frame usually has a fixed size with local variables.
- *Heap* contains dynamically allocated data.

Notice that the *stack* and *heap* both grow in size dynamically as the program executes. It is thus important that they do not grow within each other causing memory access violations. This is resolved by having the *stack* and *heap* located at opposite ends of the memory and have them grow towards each other. A pointer of the lowest / highest address of the *stack* and *heap* is maintained respectively. When the *stack* pointer meets the *heap* pointer, there exists no more memory available for allocation.

## 7.3 Memory Management

**Definition 7.12** (Garbage)**.** A *garbage* is any value that is no longer needed for execution by later parts of the program. Specifically, it refers to allocated memory which contains data that is no longer needed.

**Definition 7.13** (Reclaiming Memory)**.** There are two solutions for managing garbage:

1. Allow programmer to manage memory explicitly.
2. Employ automatic garbage collection.

Each of these solutions have their respective benefits and drawbacks:

- **Manual management**
  - Benefits
    - Low run time overhead.
    - Allows manual control of resources.
    - May not need to free memory if there are plenty left.
  - Drawbacks
    - Memory leaks
      - Object allocated but no longer have any pointers referencing it making it inaccessible.
    - Dangling references
      - Multiple references to same object, one of them used to free the object but other references still point to same location.
    - Double free bugs
      - Trying to free a memory location which has already been freed.
    - More cognitive burden when writing code
- **Automatic Garbage Collection (GC)**
  - Benefits
    - Lower cognitive burden, no need to explicitly manage memory, reducing memory-deallocation related bugs
  - Drawbacks
    - Need additional run time resources
    - Likely not suitable for embedded development because rich run time environment and resources required
    - Could delay or stall program execution
    - Cannot coexist with manual management

# 8 Code Generation

## 8.1 Basics

**Definition 8.1** (Code Generation)**.** *Code Generation* refers to the conversion between IR into machine instructions. Note that the output may be

- Absolute machine language.
  - Can directly execute.
- Relocatable machine language.

  - Subprograms can compile separately.
- Assembly.
  - Human readable but needs an Assembler to generate machine code.

The process of *code generation* has several components

1. **Instruction selection**.
2. **Instruction scheduling**.
3. **Register allocation**.
4. (*Optional*) **debug information generation**.

**Definition 8.2** (Major Processor Architectures)**.** There are three major processor architectures:

1. **Register**
   - Operation between any two arbitrary registers.
2. **Stack**
   - Operation between top of stack and items near top of stack.
3. **Accumulator**
   - Processor has single calculating register (*accumulator*) where calculations occur, other registers serve as data store.

**Definition 8.3** (Instruction Set)**.** An *instruction set* (IS), with its corresponding *instruction set architecture* (ISA), is the interface between a computer's software and hardware. The IS defines permitted operations for which the processor is able to handle, including

- Native data types
- Instructions
- Registers
- Address modes
- Memory architecture
- Interrupt and exception handling
- External I/O

There are two major instruction set flavours:

1. **RISC**: Reduced Instruction Set Computer
2. **CISC**: Complex Instruction Set Computer

*Remark.* Code generation needs information on target ISA such that generated code fits as many frequently used variables into registers (fastest access from processor).

**Definition 8.4** (Registers)**.** *Registers* can be categorized into three categories:

1. **General registers**
   (a) **Data registers**
   (b) **Pointer registers**
   (c) **Index registers**
2. **Segment registers**
3. **Control registers**

## 8.2 Simple Code Generator

**Definition 8.5** (Basic Blocks)**.** IR instructions can be grouped into *basic blocks*. *Basic blocks* are sequences of consecutive instructions where control flow enters at beginning of block and exits at end of block with no branching in between.

A *name* is considered *live* at some given point of execution if its value is referenced afterwards in the program.

**Definition 8.6** (Basic Block Leader)**.** A *leader* of a basic block is an IR instruction statement which satisfies any of the three requirements:

1. First statement is a *leader*.
2. Statement which is the target of jump is a *leader*.
3. Statement immediately following a jump statement is a *leader*.

**Definition 8.7** (Identifying Basic Blocks)**.** The algorithm to partition a sequence of Three-Address Code (TAC) statements into basic blocks is given by

**Algorithm 15** Partitioning TAC statements into basic blocks

```
1: Input
2:     Sequence of TAC statements stmts
3: Output
4:     List of Basic Blocks
5: procedure PartitionTACIntoBasicBlocks(stmts)
6:     leadersSet ← FindLeaders(stmts)
7:     basicBlocks ← []
8:     for each leader ∈ leadersSet do
9:         basicBlock ← [leader]
10:        next ← NextLeader()
11:        s ← StatementsBetween(leader, next)
12:        basicBlock.add(s)
13:        basicBlocks.add(basicBlock)
14:    end for
15:    return basicBlocks
16: end procedure
```

**Definition 8.8** (Flow Graph)**.** A *flow graph* is a representation of basic blocks.

- The *initial node* is the basic block for which its leader is the first statement.
- A *direct edge* between two basic blocks $B_1 \rightarrow B_2$ indicates that either
  - There exists a jump between last statement of $B_1$ and first statement of $B_2$.
  - $B_2$ immediate follows $B_1$ and that $B_1$ does not end with unconditional jump.

**Definition 8.9** (Next-Use Analysis)**.** In order to generate code for a basic block, it is important to identify whether each variable within the basic block has references from later parts within the block.

- If variable is used again, variable is *live* and should keep in register.
- if variable is not used again, variable is *dead* and should deallocate register.

It is also important to identify whether a variable used within a basic block is *live-on-exit*, that is if the variable is used in succeeding basic blocks. Since code is generated on a block-by-block basis, it cannot be determined how the generator arrived at the current block, and so the variables cannot be assumed to preside within registers. Information for next blocks are also not available as variables in registers should be restored back into memory

before block is exited. It is important that variables are kept in registers as long as possible to avoid reloading if the variables are needed again in the same block.

**Definition 8.10** (Next-Use Algorithm)**.** An algorithm for computing *next-use* and *liveness* information for a given basic block is given below.

The algorithm consists of two passes:

1. **Forward pass**.
   - Scan basic block to find its end.
2. **Backward pass**.
   - Scan basic block from end to beginning.

Initially, it is assumed that

1. All non-temporaries are *live*.
2. All temporaries are *dead*.

If some TAC statement such as

```
x = y OP z
```

Is encountered during the backward pass, then

1. Information from symbol table about *liveness* and *next-use* `x`, `y` and `z` is attached to statement as metadata.
2. Set `x` to *dead* and *no next use* in the symbol table.
3. Set `y` and `z` to *live* and their next uses to the statement in the symbol table.

**Definition 8.11** (Simple Code Generation Algorithm)**.** Given some basic blocks, then for each basic block

- Translate one statement at a time based on location of operands.
  - **Register Descriptor**
    - What is contained within a register, used to determine whether a new register is required.
  - **Address Descriptor**
    - Where a name is located, to determine which access method is suitable.

It is assumed that

- There exists a corresponding target-language operator for each TAC operator.
- Computed results stored in a register for as long as possible, until the register is needed for another computation or before a procedure call, jump or label.

Then the simple code generate algorithm is given by

**Algorithm 16** Simple code generation algorithm [2]

```
1: Input
2:     x = y OP z
3: Output
4:     Generated target code
5: procedure GenerateTargetCode(x = y OP z)
6:     L ←GetRegister(x = y OP z)
7:     L_y ←GetLocation(y)
8:     if L_y ≠ L then
9:         Generate(MOV L_y, L)
10:    end if
11:    Generate(OP L_z, L)
12:    UpdateLocation(x, L)
13:    if L ∈ R then
14:        UpdateRegister(L, x)
15:    end if
16:    if y ∈ R ∧ ¬HasNextUse(y) ∧ ¬LiveOnExit(y) then
17:        ClearRegister(y)
18:    end if
19:    if z ∈ R ∧ ¬HasNextUse(z) ∧ ¬LiveOnExit(z) then
20:        ClearRegister(z)
21:    end if
22: end procedure

23: procedure GetRegister(x = y OP z)
24:    if IsAt(y, L_y ∈ R) ∧ ¬ContainsOtherNames(y, L_y) ∧
    ¬IsLive(y) ∧ ¬HasNextUse(y) then
25:        return L_y
26:    else if HasEmptyRegister(R, r_∅) then
27:        return r_∅
28:    else if HasNextUse(x) then
29:        if ∃ SuitableRegisterOccupiedByVariable(r_o, a) then
30:            addr ← GetAddress(a)
31:            Generate(MOV r_o, addr)
32:            UpdateRegister(r_o, x)
33:            return r_o
34:        end if
35:        return GetAddress(x)
36:    end if
37: end procedure
```

## References

[1] W. R. Stevens, *Advanced Programming in the UNIX Environment*, 3rd ed. Ann Arbor, Michigan, USA: Pearson Education, 2013, ISBN: 978-0321637734.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, USA: Pearson Education, Inc., 2007, ISBN: 978-0321486813.