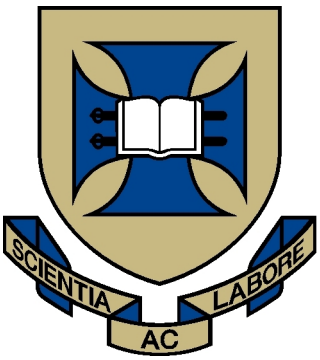
Daniel **Fitz**
43961229



University of Queensland

CSSE3002 – The Software Process

Lecture 1

Table of Contents

Phases of a Compiler

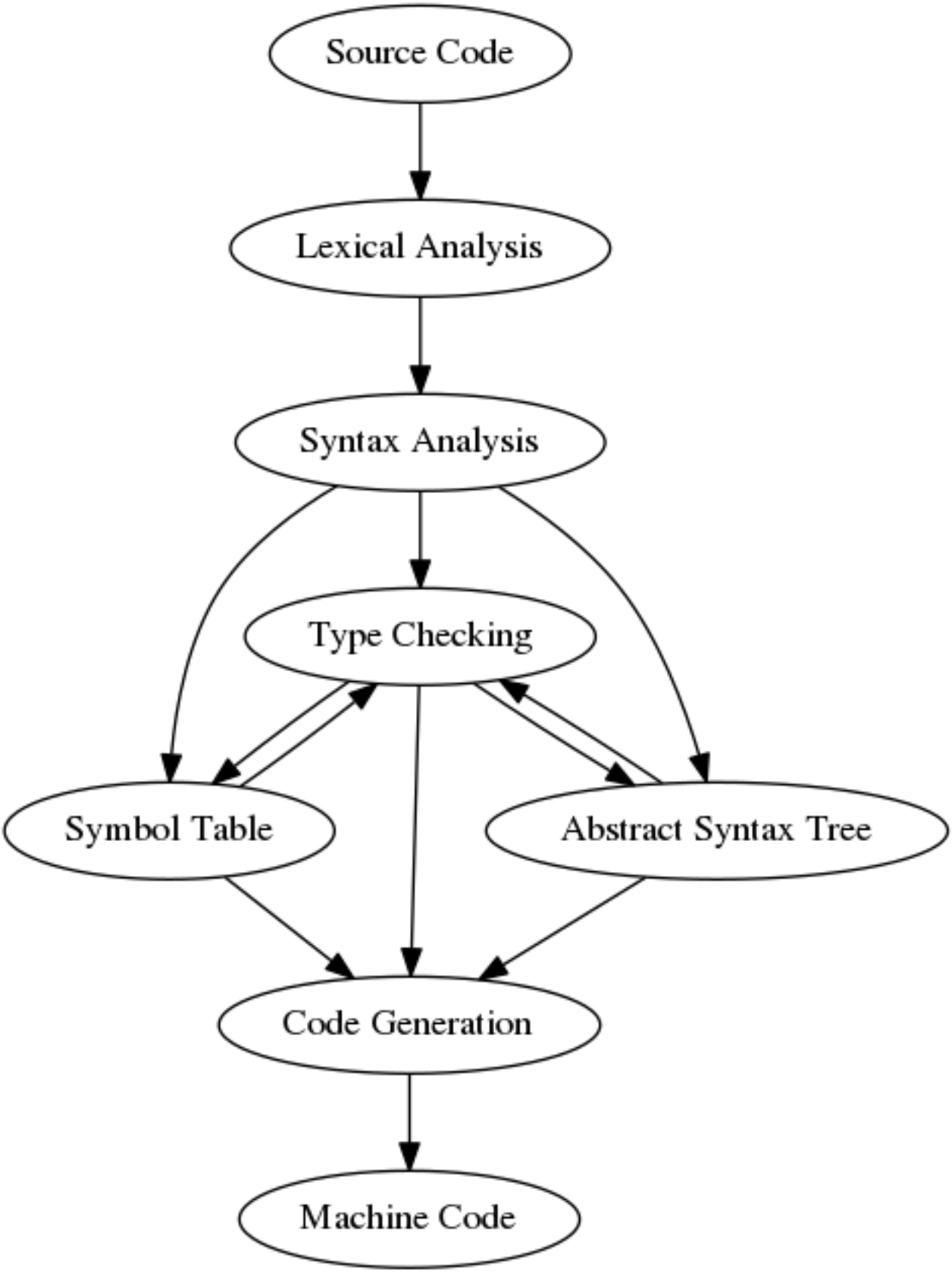


Figure 1: Phases of a Compiler

See Figure 1

Lexical Analysis

- **Input:** a sequence of characters representing a program
- **Output:** a sequence of lexical tokens
- **Lexical tokens:** identifiers, numbers, keywords (e.g., “if”, “while”), symbols (e.g., “+”, “<=”)
- **Ignores white space:** blanks, tabs, newlines, carriage returns, form feeds
- **Comments:** treated as white space

Syntax Analysis

- **Input:** a sequence of lexical tokens
- **Output:** an abstract syntax tree and symbol table
- Symbol table
  - contains information about all identifiers that are defined within the program (plus a few predefined ones)
  - may be organised into scopes, e.g, identifiers defined within a procedure

Type Checking a.k.a Static Semantic Analysis

- **Input:** Symbol table and abstract syntax tree
- **Output:** Updated symbol table and abstract syntax tree
- Resolves all references to identifiers
  - updates symbol table entries with type information
  - checks abstract syntax tree for type correctness
  - updates abstract syntax tree with type coercions

Code Generation

- **Input:** Symbol table and (updated) abstract syntax tree
- **Output:** code for the target machine
- May include
  - machine-independent optimisations
  - machine-dependent optimisations
  - instruction selection
  - register allocation

Context-free Grammars

The context-free grammar has start symbol E, nonterminals {E, Op}, and terminals {“(”, “)”, *number*, “+”, “-”, “*”}

- E $\rightarrow$ E Op E
- E $\rightarrow$ “(” E “)”
- E $\rightarrow$ *number*
- Op $\rightarrow$ “+”
- Op $\rightarrow$ “-”
- Op $\rightarrow$ “*”

Definition (Context-free Grammar)

A context-free grammar consists of

- A finite set, $\sum$, of terminal symbols
- A finite nonempty set of nonterminal symbols (disjoint from the terminals)
- A finite nonemtpy set of productions of the form $A \rightarrow \alpha$, where A is a nonterminal symbol, and α is a possibly empty sequence of symbols, each of which is either terminal or nonterminal
- A start symbol, which must be a nonterminal

Definition (Direct derivation step)

Let both α and β be possibly empty sequences of terminal and nonterminal symbols, and N a nonterminal symbol. If there is a production of the form $N \rightarrow \gamma$, then a derivation step can be applied to a sequence of symbols of the form $\alpha N \beta$ to

replace the N by the sequence γ to give the sequence $\alpha\gamma\beta$. We say $\alpha N\beta$ directly derives $\alpha\gamma\beta$, written

$$\alpha N\beta \Rightarrow \alpha\gamma\beta$$

Definition (Derives)

We say a sequence of terminal and nonterminal symbols α derives a sequence β, written

$$\alpha \Rightarrow^* \beta$$

if there is a finite sequence of zero of more direct derivation steps starting from α and finishing with β. That is, there must exist one or more sequences, $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n$, such that,

$$\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = \beta$$

Definition (Nullable)

A possibly empty sequence of symbols, α, is nullable if it can derive the empty sequence of symbols, i.e., written either

$$\alpha \Rightarrow^* \epsilon$$

or

$$\alpha \Rightarrow^*$$

Some rules for nullable

- ϵ is nullable
- any terminal symbol is not nullable
- a sequence of the form $S_1S_2 \ldots S_n$ is nullable if all of the constructs $S_1, \ldots, S_n$ are nullable
- a set of alternatives $S_1 \mid \ldots \mid S_n$ is nullable if at least one of the alternatives is nullable
- EBNF constructs for optionals and repetitions are nullable
- a nonterminal N is nullable if there is a production for N with a nullable right side

Definition (Language)

The language $\mathcal{L}(G)$ corresponding to a grammar, G, is the set of all finite sequences of terminal symbols that can be derived from the start symbol of the grammar using its productions.

$$\mathcal{L}(G) = \{s \in seq \sum \mid S \Rightarrow^* s\}$$

where S is the start symbol of G and $\sum$ is its set of terminal symbols

Definition (sentence)

A sequence of terminal symbols s such that $S \Rightarrow^* s$ is called a sentence in the language

Definition (sentential form)

A sequence of terminals and nonterminals α such that $S \Rightarrow^* \alpha$ is called a *sentential form* of the language

Hence all sentences are also sentential forms.

Parse trees

A derivation sequence for a string in the language (a sentence) defines a corresponding parse tree.

- A direct derivation step corresponding to the production $N \rightarrow \alpha$ generates a parse tree with N as its root node and the symbols in α as children.
- A derivation sequence generates a parse tree by applying each derivation step to progressively to build the parse tree.

Note that two different derivation sequences may give rise to the same parse tree due to nonterminals being expanded in different orders

Derivation Sequence (leftmost)

3+4

$$
\begin{aligned}
E &\Rightarrow E\ Op\ E \\
&\Rightarrow number(3)\ Op\ E \\
&\Rightarrow number(3) + E \\
&\Rightarrow number(3) + number(4) \\
E &\Rightarrow^{*} number(3) + number(4)
\end{aligned}
$$

Concrete Syntax Tree

```
1    if x<0 then z:=-x else z:=x
```

KW_IF, IDENTIFIER(“x”), LESS, NUMBER(0), KW_THEN, IDENTIFIER(“z”), ASSIGN, MINUS, IDENTIFIER(“x”), KW_ELSE, IDENTIFIER(“z”), ASSIGN, IDENTIFIER(“x”)

See Figure 2

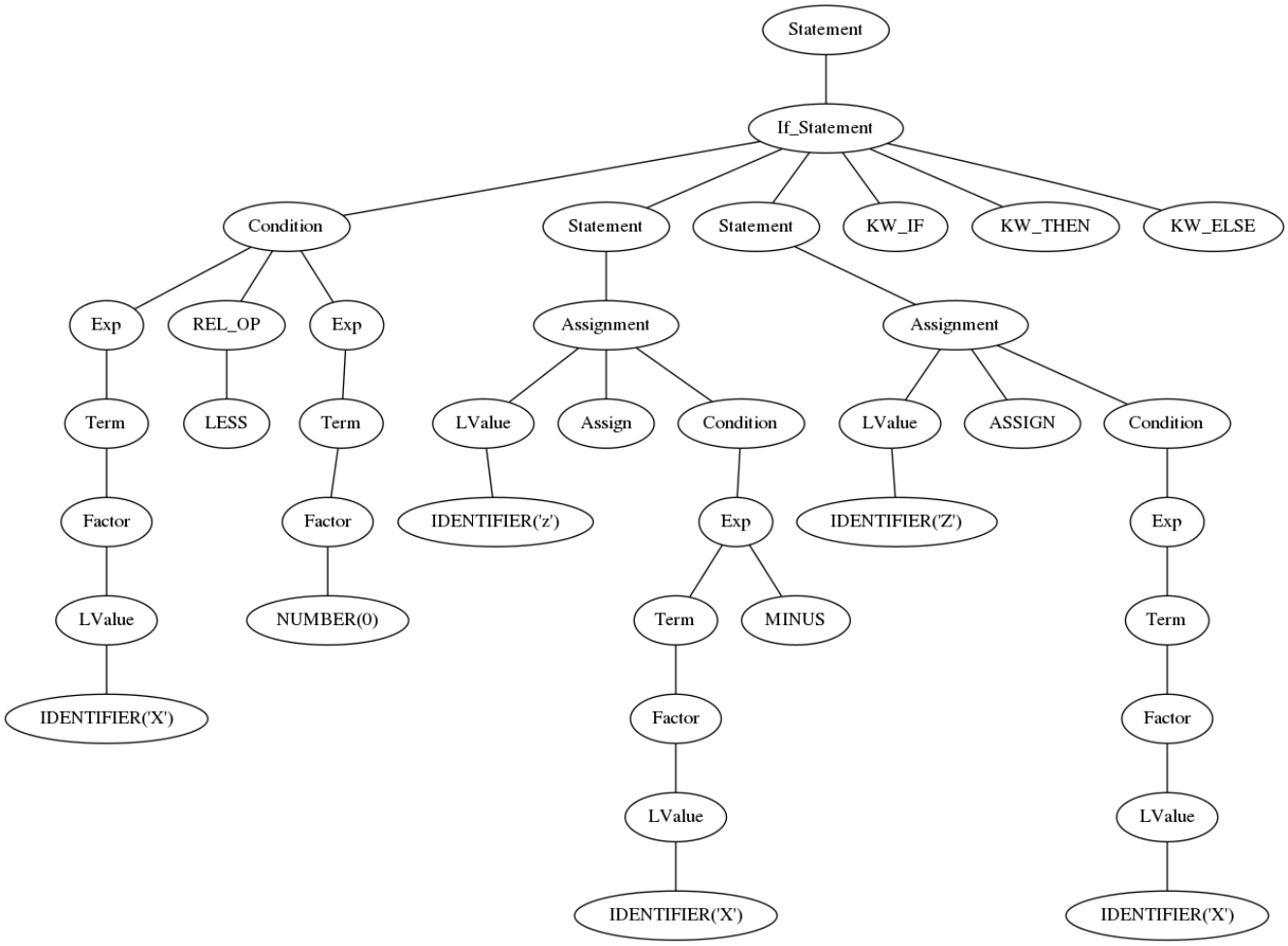


Figure 2: Concrete Syntax Tree

Abstract Syntax Tree

See Figure 3

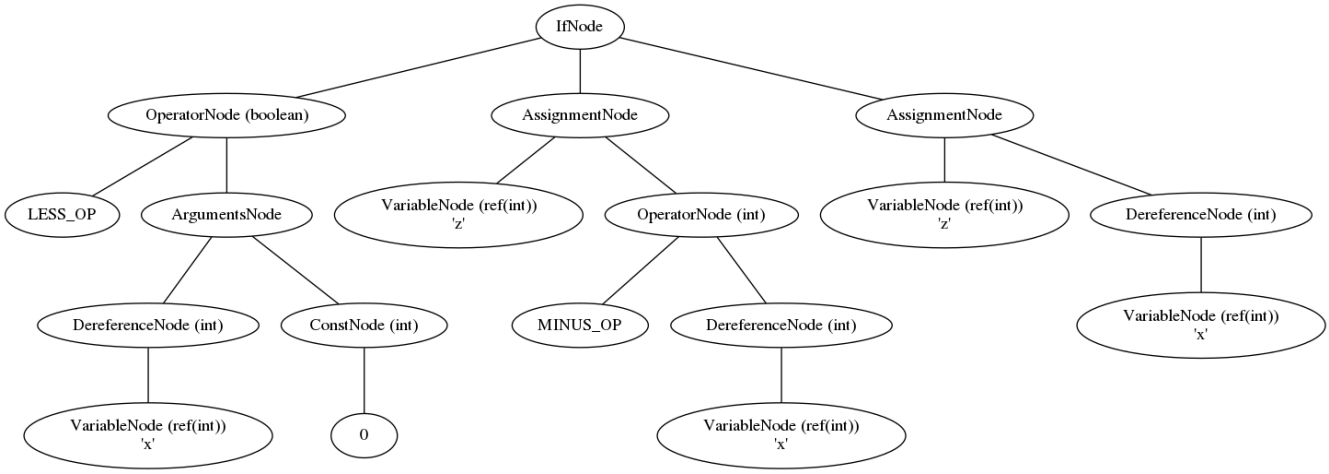


Figure 3: Abstract Syntax Tree

Ambiguous Grammars

Definition (Ambiguous grammar for sequence)

A grammar is ambiguous for a sequence, s, if there is more than one parse tree for s

Definition (Ambiguous grammar)

A grammar is ambiguous if it is ambiguous for any sequence

The following is an ambiguous grammar for expressions with the binary operator "-", in which N represents a number.

$$E \to E" - "E$$

$$E \to N$$

For example, a string like "1-2-3" has two different parse trees

Left and Right Associative operators

To remove ambiguity and treat "-" as a left associative operator (as usual) we can rewrite the grammar to

$$E \to E" - "T$$

$$E \to T$$

$$T \to N$$

and to treat "-" as right associative (not the usual interpretation) we use

$$E \to T" - "E$$

$$E \to T$$

$$T \to N$$

Operator Precedence

With a grammar like

$$E \to E\text{"}+\text{"}E \mid E\text{""}E \mid F$$

the precedence of the operators is not specified, so that a string like "1+2*3" is ambiguous; it can be interpreted as either "1+(2*3)" or "(1+2)*3", where the brackets just show grouping. To ensure that "+" has lower precedence than "*" and remove the ambiguity, we can rewrite the grammar as

$$E \to E\text{"}+\text{"}T \mid T$$

$$T \to T\text{""}F \mid F$$

which also treats both "+" and "*" as left associative.

The following is an ambiguous grammar for lists with elements corresponding to the nonterminal X.

$$L \to \epsilon \mid X \mid LL$$

For example, a sentential form corresponding to a single X can be derived via the following three derivation sequences (as well as many others)

$$L \Rightarrow X$$

$$L \Rightarrow LL \Rightarrow L\epsilon = L \Rightarrow X$$

$$L \Rightarrow LL \Rightarrow LLL \Rightarrow LL\epsilon = LL \Rightarrow \epsilon L = L \Rightarrow X$$

which generate different parse trees

In the above grammar, the production $L \to LL$ generates ambiguity because, if it is used in a derivation step, it generates a sequence containing LL, and then, for example, either L may again be expanded to LL to get the same sequence LLL. This can be resolved by replacing the production with, say,

$$L \to LX$$

In addition, $L \to \epsilon$ and $L \to LX$ together generate any sequence of zero or more occurences of X, including a single occurrence, meaning that the second alternative $L \to X$ is redundant and introduces ambiguity. An unambiguous grammar that generates the same language (set of strings) is

$$L \to \epsilon \mid LX$$

Chomsky Hierarchy of Grammars

Nonterminal symbols are uppercase and terminal symbols lowercase. Greek letters stand for possibly empty sequences of terminals and nonterminals, and ϵ is the empty sequence.

- **Type 3** Left (or right) linear grammars
  - Finite automation (state machine)
  - Regular expression
  - Left (or right) linear grammar

$$A \to \epsilon \mid aB \mid a$$

- **Type 2** Context-free grammars
  - Pushdown automation (automation with a stack)
  - Context-free grammar

$$A \to \alpha$$

- **Type 1** Context-sensitive grammars
  - Context-sensitive grammar

$$\beta A \gamma \to \beta \alpha \gamma$$

- **Type 0** Unrestricted grammars (α can't be ϵ)
  - Unrestricted grammar
  - Turing machine equivalent

$$\alpha \to \beta$$

Stack Machine

```
1    LOADCON 3
2    LOADCON 4
3    MPY
4    LOADCON 2
5    ADD
```

See Figure 4

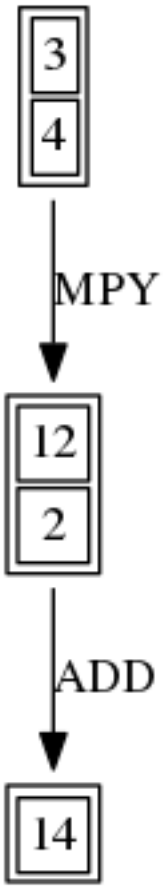


Figure 4: Stack Machine Example 1

```
1    LOADCON 2
2    LOADCON 3
3    LOADCON 4
4    MPY
5    ADD
```

See Figure 5

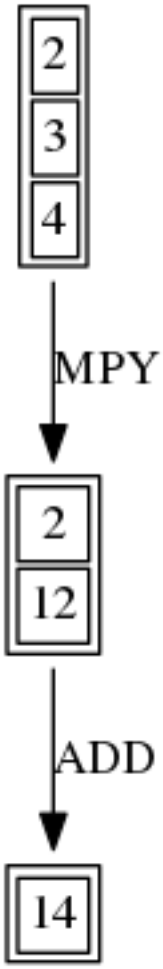


Figure 5: Stack Machine Example 2

If Then Else

if x<0 then z:=−x **else** z:=x

```
1     0  LOADCON 3 ——addr(x)
2     2  LOADFRAME
3     3  LOADCON 0
4     5  LESS
5     ————————————
6     6  LOADCON 10
7     8  BR—FALSE
8     ————————————
9     9  LOADCON 3 ——addr(x)
10    11 LOADFRAME
11    12 NEGATE
12    13 LOADCON 4 ——addr(z)
13    15 STOREFRAME
14    ————————————
15    16 LOADCON 6
```

```
16 |     18 BR
17 |     ──────────────
18 |     19 LOADCON 3 ──addr(x)
19 |     21 LOADFRAME
20 |     22 LOADCON 4 ──addr(z)
21 |     24 STOREFRAME
22 |     25 ...
```