

Daniel **Fitz**  
43961229



University of Queensland  
**CSSE3002** – The Software Process

Lecture 1

## Table of Contents

---

<b>1</b>	<b>Phases of a Compiler</b>	<b>2</b>
1.1	Lexical Analysis . . . . .	2
1.2	Syntax Analysis . . . . .	2
1.3	Type Checking a.k.a Static Semantic Analysis . . . . .	2
1.4	Code Generation . . . . .	2
<b>2</b>	<b>Context-free Grammars</b>	<b>2</b>
	Definition (Context-free Grammar) . . . . .	2
	Definition (Direct derivation step) . . . . .	2
	Definition (Derives) . . . . .	4
	Definition (Nullable) . . . . .	4
	Definition (Language) . . . . .	4
	Definition (sentence) . . . . .	4
	Definition (sentential form) . . . . .	4
2.1	Parse trees . . . . .	4
	Derivation Sequence (leftmost) . . . . .	5
<b>3</b>	<b>Concrete Syntax Tree</b>	<b>5</b>
3.1	Abstract Syntax Tree . . . . .	6
3.2	Ambiguous Grammars . . . . .	6
	Definition (Ambiguous grammar for sequence) . . . . .	6
	Definition (Ambiguous grammar) . . . . .	6
3.3	Left and Right Associative operators . . . . .	6
3.4	Operator Precedence . . . . .	6
<b>4</b>	<b>Chomsky Hierarchy of Grammars</b>	<b>7</b>
<b>5</b>	<b>Stack Machine</b>	<b>8</b>
5.1	If Then Else . . . . .	8

## 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,  $\Sigma$ , of terminal symbols
- A finite nonempty set of nonterminal symbols (disjoint from the terminals)
- A finite nonempty set of productions of the form  $A \rightarrow \alpha$ , where  $A$  is a nonterminal symbol, and  $\alpha$  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  $\alpha$  and  $\beta$  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

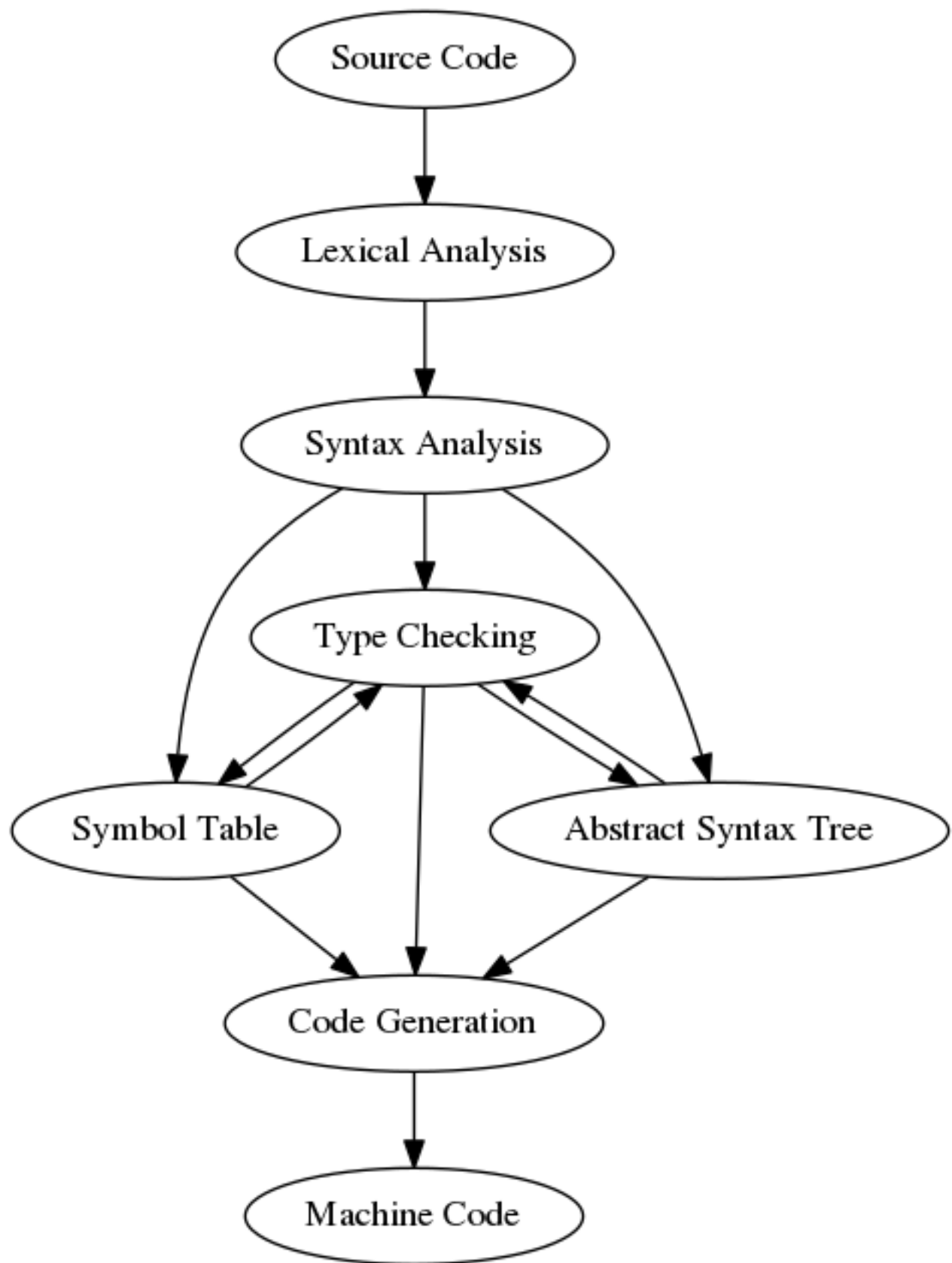


Figure 1: Phases of a Compiler

replace the  $N$  by the sequence  $\gamma$  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  $\alpha$  derives a sequence  $\beta$ , written

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

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

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

### Definition (Nullable)

A possibly empty sequence of symbols,  $\alpha$ , 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

- $\epsilon$  is nullable
- any terminal symbol is not nullable
- a sequence of the form  $S_1 S_2 \dots S_n$  is nullable if all of the constructs  $S_1, \dots, S_n$  are nullable
- a set of alternatives  $S_1 \mid \dots \mid S_n$  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.

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

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  $\alpha$  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  $\alpha$  as children.
- A derivation sequence generates a parse tree by applying each derivation step 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

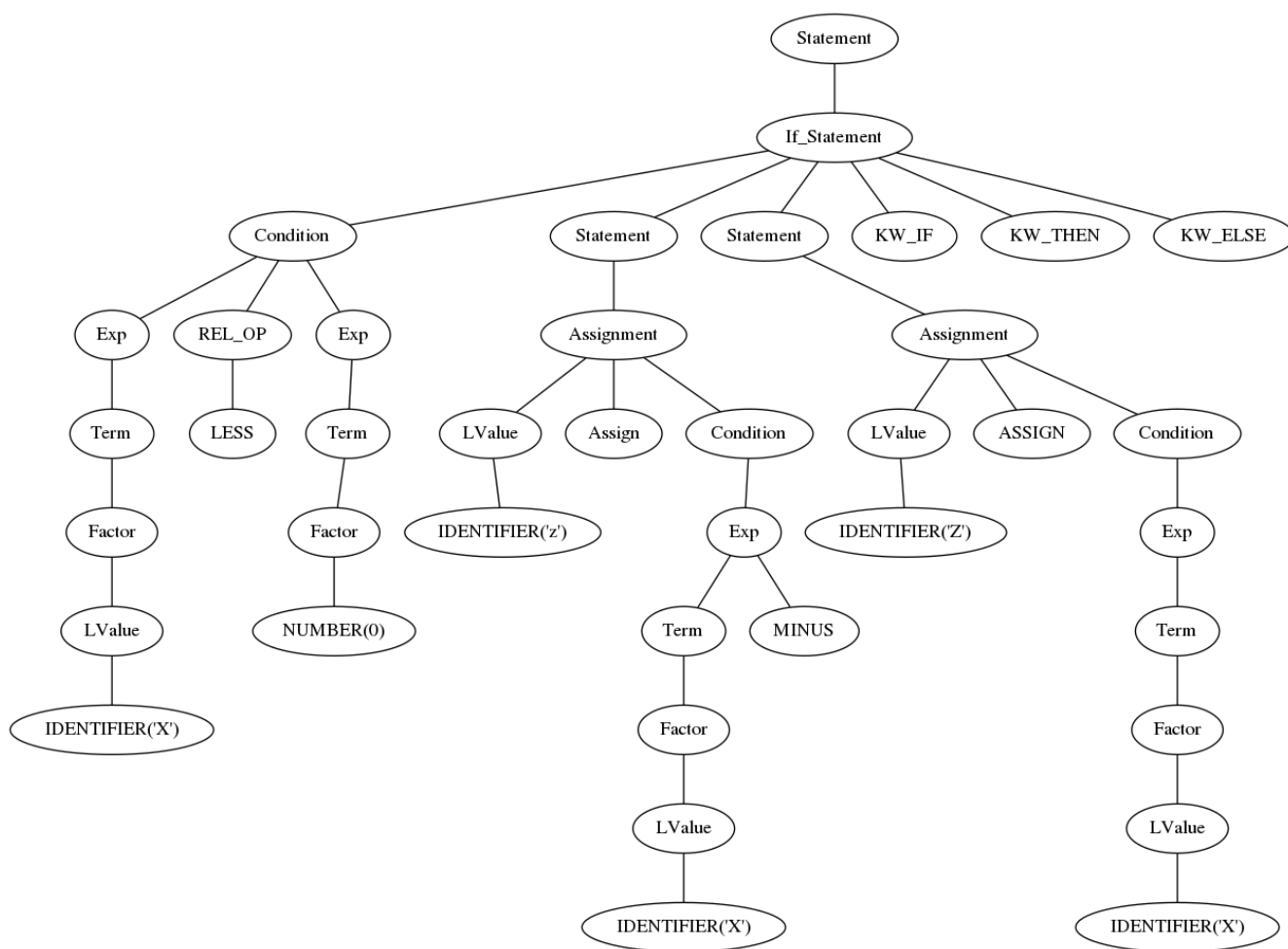


Figure 2: Concrete Syntax Tree

### Derivation Sequence (leftmost)

3+4

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

### Concrete Syntax Tree

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

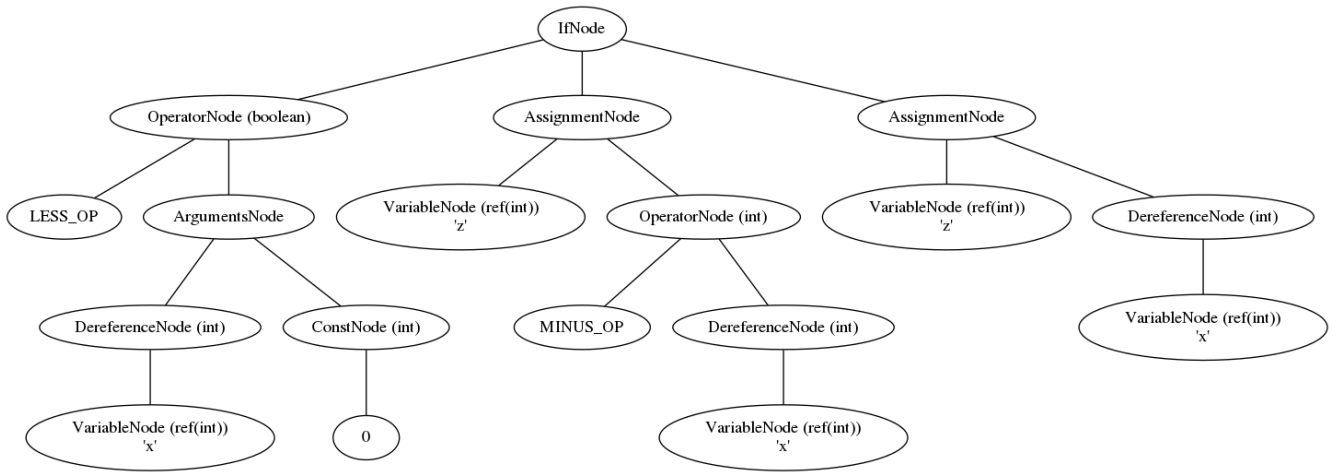


Figure 3: Abstract Syntax Tree

## Abstract Syntax Tree

See Figure 3

## 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.

$$E \rightarrow E' - E$$

$$E \rightarrow N$$

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

$$E \rightarrow E' - T$$

$$E \rightarrow T$$

$$T \rightarrow N$$

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

$$E \rightarrow T' - E$$

$$E \rightarrow T$$

$$T \rightarrow N$$

## Operator Precedence

With a grammar like

$$E \rightarrow E' + E \mid E' E \mid F$$

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

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

which also treats both “+” and “\*” as left associative.

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

$$L \rightarrow \epsilon \mid X \mid LL$$

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  $L \rightarrow LL$  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,

$$L \rightarrow LX$$

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

$$L \rightarrow \epsilon \mid LX$$

## Chomsky Hierarchy of Grammars

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

- **Type 3** Left (or right) linear grammars

- Finite automation (state machine)
- Regular expression
- Left (or right) linear grammar

$$A \rightarrow \epsilon \mid aB \mid a$$

- **Type 2** Context-free grammars

- Pushdown automation (automation with a stack)
- Context-free grammar

$$A \rightarrow \alpha$$

- **Type 1** Context-sensitive grammars

- Context-sensitive grammar

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

- **Type 0** Unrestricted grammars ( $\alpha$  can't be  $\epsilon$ )

- Unrestricted grammar
- Turing machine equivalent

$$\alpha \rightarrow \beta$$



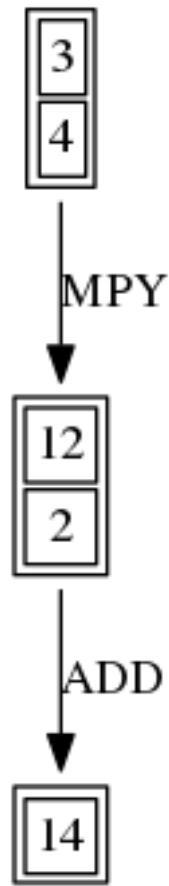


Figure 4: Stack Machine Example 1

## Stack Machine

---

```
LOADCON 3
LOADCON 4
MPY
LOADCON 2
ADD
```

See Figure 4

```
LOADCON 2
LOADCON 3
LOADCON 4
MPY
ADD
```

See Figure 5

### If Then Else

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

```
0  LOADCON 3 --addr(x)
```

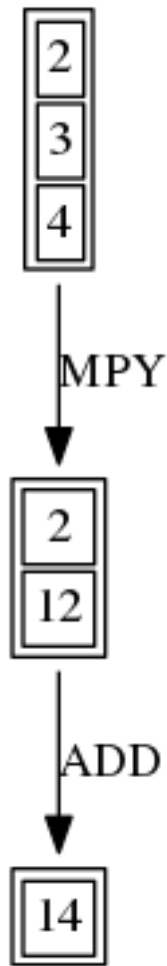


Figure 5: Stack Machine Example 2

```

2  LOADFRAME
3  LOADCON 0
5  LESS
-----
6  LOADCON 10
8  BR-FALSE
-----
9  LOADCON 3 --addr(x)
11 LOADFRAME
12 NEGATE
13 LOADCON 4 --addr(z)
15 STOREFRAME
-----
16 LOADCON 6
18 BR
-----
19 LOADCON 3 --addr(x)
21 LOADFRAME
22 LOADCON 4 --addr(z)
24 STOREFRAME
25 ...

```