Compiler Construction

Lecture # 04

Mr. Usman Wajid

usman.wajid@nu.edu.pk

January 31, 2023

National University
of Computer & Emerging Sciences

Phases of Compiler: Lexical Analysis

1. **Lexical Analysis:**
   - 1^{st} phase of Compiler, also known as Scanner

Phases of Compiler: Lexical Analysis

1. **Lexical Analysis:**
   - 1^{st} phase of Compiler, also known as Scanner
   - it verifies that input character sequence is lexically valid

Phases of Compiler: Lexical Analysis

1. **Lexical Analysis:**
   - 1^{st} phase of Compiler, also known as Scanner
   - it verifies that input character sequence is lexically valid
   - groups stream of characters into meaningful sequence of lexemes

Phases of Compiler: Lexical Analysis

1. **Lexical Analysis:**
   - 1^{st} phase of Compiler, also known as Scanner
   - it verifies that input character sequence is lexically valid
   - groups stream of characters into meaningful sequence of lexemes
   - each lexeme is assigned a token of the form (token-name, attribute-value)

Phases of Compiler: Lexical Analysis

1. **Lexical Analysis:**
   - 1^{st} phase of Compiler, also known as Scanner
   - it verifies that input character sequence is lexically valid
   - groups stream of characters into meaningful sequence of lexemes
   - each lexeme is assigned a token of the form (token-name, attribute-value)
   - discards white space and comments

Lexical Analysis: Tokens

Lexical Analysis: Tokens

- **Tokens** (token-name, attribute-value)

Lexical Analysis: Tokens

- **Tokens** (token-name, attribute-value)
  - **Token-name:** is an abstract symbol that is used during syntax analysis

Lexical Analysis: Tokens

- **Tokens** (token-name, attribute-value)
  - **Token-name:** is an abstract symbol that is used during syntax analysis
  - **Attribute-value:** points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation

Lexical Analysis: Tokens example

- source language input,

```
position = initial + rate * 60
```

Lexical Analysis: Tokens example

- source language input,

```
position = initial + rate * 60
```

- Symbol table,

Lexemes	**Tokens**
position	(id,1)
=	(=)
initial	(id,2)
+	(+)
rate	(id,3)
*	(*)
60	(60)

Lexical Analysis: Tokens example

- source language input,

```
position = initial + rate * 60
```

- Symbol table,

Lexemes	**Tokens**
position	(id,1)
=	(=)
initial	(id,2)
+	(+)
rate	(id,3)
*	(*)
60	(60)

- After lexical analysis,

```
(id,1) (=) (id,2) (+) (id,3) (*) (60)
```

Phases of Compiler: Syntax Analysis

2. **Syntax Analysis:**

Phases of Compiler: Syntax Analysis

2. **Syntax Analysis:**
    - 2^{nd} phase of Compiler, also known as Parser

Phases of Compiler: Syntax Analysis

2. **Syntax Analysis:**
   - 2^{nd} phase of Compiler, also known as Parser
   - the first component of the tokens, i.e., token-name, is used to construct a syntax tree

Phases of Compiler: Syntax Analysis

2. **Syntax Analysis:**
   - 2^{nd} phase of Compiler, also known as Parser
   - the first component of the tokens, i.e., token-name, is used to construct a syntax tree
   - reflects the grammatical structure of the token stream

Phases of Compiler: Syntax Analysis

2. **Syntax Analysis:**

   - 2^{nd} phase of Compiler, also known as Parser
   - the first component of the tokens, i.e., token-name, is used to construct a syntax tree
   - reflects the grammatical structure of the token stream
   - in Syntax tree, each inner node represents an operation and the children of the node represents the arguments of the operation

Phases of Compiler: Semantic Analysis

3. **Semantic Analysis:**

Phases of Compiler: Semantic Analysis

3. **Semantic Analysis:**
    - the syntax tree and symbol table is used as input in parallel to check the semantic consistency

Phases of Compiler: Semantic Analysis

3. **Semantic Analysis:**

   - the syntax tree and symbol table is used as input in parallel to check the semantic consistency
   - an important role of semantic analysis is type checking

Phases of Compiler: Semantic Analysis

3. **Semantic Analysis:**

   - the syntax tree and symbol table is used as input in parallel to check the semantic consistency
   - an important role of semantic analysis is type checking
   - An an example, compiler report an error if a floating point number is used in index array

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

   - An intermediate machine-like assembly language representation is generated in this phase

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

   - An intermediate machine-like assembly language representation is generated in this phase
   - For example, a three-address intermediate code contains three operands per instruction in assembly language

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

   - An intermediate machine-like assembly language representation is generated in this phase
   - For example, a three-address intermediate code contains three operands per instruction in assembly language
   - ```
     t1 = intofloat (60)
     t2 = id3 * t1
     t3 = id2 + t2
     id1 = t3
     ```
     ```

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

   - An intermediate machine-like assembly language representation is generated in this phase
   - For example, a three-address intermediate code contains three operands per instruction in assembly language
   - ```
     t1 = intofloat (60)
     t2 = id3 * t1
     t3 = id2 + t2
     id1 = t3
     ```
   - each three- address code has at most one operator on the right side
     ```

Phases of Compiler: Intermediate Code Generation

4. **Intermediate Code Generation:**

   - An intermediate machine-like assembly language representation is generated in this phase
   - For example, a three-address intermediate code contains three operands per instruction in assembly language
   - ```
     t1 = intofloat (60)
     t2 = id3 * t1
     t3 = id2 + t2
     id1 = t3
     ```
   - each three- address code has at most one operator on the right side
   - temporary names such as `t1`, `t2` and `t3` are used to store the computed value
     ```

Phases of Compiler: Code Optimization

5. **Code Optimization**

Phases of Compiler: Code Optimization

5 Code Optimization

- It improves the intermediate code to fine-tune the target code

Phases of Compiler: Code Optimization

5. **Code Optimization**

   - It improves the intermediate code to fine-tune the target code
   - hence, improves time and space efficiency

Phases of Compiler: Code Optimization

5 Code Optimization

- It improves the intermediate code to fine-tune the target code
- hence, improves time and space efficiency
- the optimized code (in the previous example) would be

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Phases of Compiler: Code Generation

6. **Code Generation:**
   - It maps intermediate machine-like representation into target language

Phases of Compiler: Code Generation

6. **Code Generation:**
   - It maps intermediate machine-like representation into target language
   - the generated code (in the previous) would be,

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Phases of Compiler: Code Generation

6. **Code Generation:**
   - It maps intermediate machine-like representation into target language
   - the generated code (in the previous) would be,

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

   - the first operand of each instruction specifies a destination

Phases of Compiler: Code Generation

6. **Code Generation:**
   - It maps intermediate machine-like representation into target language
   - the generated code (in the previous) would be,

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

   - the first operand of each instruction specifies a destination
   - `R1`, `R2` and `R2` are memory registers

Phases of Compiler: Code Generation

6. **Code Generation:**
   - It maps intermediate machine-like representation into target language
   - the generated code (in the previous) would be,

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

   - the first operand of each instruction specifies a destination
   - `R1`, `R2` and `R2` are memory registers
   - The `F` in each instruction depicts floating-point numbers