Compiler Construction

Lecture # 06

Mr. Usman Wajid

usman.wajid@nu.edu.pk

February 13, 2023

National University
of Computer & Emerging Sciences

Phases of Compilers: Lexical Analysis

- **Lexical Analysis:** Typical tasks performed by lexical analyzer are,

Phases of Compilers: Lexical Analysis

- **Lexical Analysis:** Typical tasks performed by lexical analyzer are,

  1. Remove white space and comments

Phases of Compilers: Lexical Analysis

- **Lexical Analysis:** Typical tasks performed by lexical analyzer are,

  1. Remove white space and comments
  2. Encode constants as tokens

Phases of Compilers: Lexical Analysis

- **Lexical Analysis:** Typical tasks performed by lexical analyzer are,

  1. Remove white space and comments
  2. Encode constants as tokens
  3. Recognize keywords

Phases of Compilers: Lexical Analysis

- **Lexical Analysis:** Typical tasks performed by lexical analyzer are,

    1. Remove white space and comments
    2. Encode constants as tokens
    3. Recognize keywords
    4. Recognize and store identifier names in a global symbol table

Lexical Analysis

- **Removal of White spaces & Comments**
  - Most languages allow arbitrary amounts of white spaces to appear between tokens.
  - Comments are ignored and treated as a white space
  - if white space is eliminated by the lexical analyzer, the parser will never consider it

Lexical Analysis

- **Reading Ahead:**
  - Before a token is decided, a lexical analyzer may need to read some more characters
  - Any input buffer is maintained for this purpose
  - Example, >=

Lexical Analysis

- **Encode Constants:** Integer constants can be allowed either by,

  1. creating a terminal symbol, say num, for such constants. Example, Input 16+28+50 will be transformed into,

     `<num,16><+><num,28><+><num,55>`

Lexical Analysis

- **Encode Constants:** Integer constants can be allowed either by,

  1. creating a terminal symbol, say num, for such constants. Example, Input 16+28+50 will be transformed into,

     ```
     <num,16><+><num,28><+><num,55>
     ```

  2. Or by incorporating the syntax of integer constants into the grammar. Example,
     Input 16+28+50 will be transformed into,

     ```
     <16><+><28><+><50>
     ```

Lexical Analysis

- **Encode Constants:** Integer constants can be allowed either by,

1. creating a terminal symbol, say num, for such constants. Example, Input 16+28+50 will be transformed into,

   ```
   <num,16><+><num,28><+><num,55>
   ```

2. Or by incorporating the syntax of integer constants into the grammar. Example,
   Input 16+28+50 will be transformed into,

   ```
   <16><+><28><+><50>
   ```

3. Here, the terminal symbol has no attribute, so its tuple is simple <+>

Lexical Analysis

Keywords

Most languages use fixed reserved words such as for, do and if etc. These reserved words are called keywords

Identifiers

User defined character strings are called identifiers. Grammars routinely treat identifiers as terminals to simplify the parser.

Example,

input: `count = count + increment;`

Terminal stream: `id = id + id`

Tokens, Patterns, & Lexemes

Tokens

Tt is a group of characters with logical meaning. It is a logical building block of a language. It consist of a token name and an optional attribute value.

Example, `<id,rate> <+>`

Pattern

It is a rule that describes the character that can be grouped into tokens. A regular expression maps input stream with patterns to identify the token

Example: Pattern/rule for **id** can be given by the regular expression:
`[A-Z,a-z]`$[A - Z, a - z, 0 - 9]^*$

Tokens, Patterns, & Lexemes

Lexeme

Each individual character stream that is mapped with a pattern and is recognized as a token.

Example: ”int” is identified as a token keyword. here ”int” is lexeme and keyword is token.

Tokens, Patterns, & Lexemes

Lexeme

Each individual character stream that is mapped with a pattern and is recognized as a token.

Example: ”int” is identified as a token keyword. here ”int” is lexeme and keyword is token.

Example: `float key =1.2;`

Lexeme	**Token**	**Pattern**
float	float	Float
key	id	`[A-Z,a-z]`$[A-Z, a-z, 0-9]^*$
=	relop	`< > <= >= = != ==`
1.2	num	Any numeric constant
;	;	;

Lexical Errors

- Lexical analyzer may not always predict errors in source code without incorporating with other components

Lexical Errors

- Lexical analyzer may not always predict errors in source code without incorporating with other components
- **Example:** `fi (a == f(x))`
  - A lexical analyzer may validate `fi` as a valid lexeme pattern and recognize it as an id token
  - A syntax analyzer then needs to address the error in `fi` as a syntax error based on its position in the syntax tree

Lexical Errors

”Panic mode” Recovery

If a lexical analyzer is unable to proceed because non of the patterns for tokens matches any prefix of the remaining input, then it goes into ”Panic mode” recovery

In such a case, all invalid successive characters are truncated until a valid pattern is achieved

Lexical Errors

”Panic mode” Recovery

If a lexical analyzer is unable to proceed because non of the patterns for tokens matches any prefix of the remaining input, then it goes into ”Panic mode” recovery

In such a case, all invalid successive characters are truncated until a valid pattern is achieved

Other possible actions can be,

- Insert a missing character into the remaining input
- replace character by another character
- transpose two adjacent characters

Lexical analysis: Input Buffering

Input Buffering

Determining the next lexeme often requires reading the input beyond the end of that lexeme

Lexical analysis: Input Buffering

Input Buffering

Determining the next lexeme often requires reading the input beyond the end of that lexeme

Example:

- The end of an identifier is determined by reading a space character after it

Lexical analysis: Input Buffering

Input Buffering

Determining the next lexeme often requires reading the input beyond the end of that lexeme

Example:

- The end of an identifier is determined by reading a space character after it
- reading only single > can be determined a single lexeme as there could be an = after it

Lexical analysis: Buffer Pointers

- two pointers to the input are maintained,
  1. **lexemebegin pointer:** marks the beginning of the current lexeme, whose extent we are attempting to determine
  2. **Forward Pointer:** that scans ahead until a pattern is found