Project 1: Lexical Analyzer

Dayana Gonzalez Cruz
CST-405: Principles of Compiler Design Lecture & Lab
Sep. 9th, 2025
Accompanying Presentation: https://vimeo.com/1117306115?share=copy
Github Link: https://github.com/

Section I. Programming Languages Review

Regular Expressions

Regular expression syntax is a notation by which to define string patterns to be identified within a text (Lambert & Louden, 2012). The software used in this course, Flex, uses a unique syntax to define regular expressions, refered to as rules (Paxson, et. al, 2001). This syntax shares many similarities with the POSIX standard for regular expression notation. Regular expression notation consists of "three basic operations: concatenation, repetition, and choice or selection" (Lambert & Louden, 2012, pg. 206).

Concatenation is usually denoted by a sequence of characters enclosed by square brackets; it indicates that the pattern must contain this specific sequence of characters itself. Repetition can be denoted by a a plus sign (+, signifying one or more repirions of the preceeding character or group), an asterisk (*, signifying 0 or more repitions), or a range ({min, max}, signifying a minimum and maximum repitition count). Choice is denoted by a pipe (|), signifying the word or, meaning it can be either the character or group on the left or right of the vertical bar (Lambert & Louden, 2012).

One example of a regular expression implemented in the flex script accompanying this project is IDENTIFIER [A-Za-z_]+[A-Za-z0-9_]*. This rule represents a pattern for a string consisting of letters, digits, and underscores the first character of which is not a digit.

Finite Automata

Finite state machines (FSM) determine whether a sequence of inputs exists in a defined set. They are "essentially graphs, like transition diagrams" (Aho, et. Al, 2007, pg. 147). In other words, a FSM can test whether is string matches a pattern. Given a character or state, a FSM has a transition to a different state. There are failure states that indicate that the pattern can no longer be matched from the last given input. There are also success stages, indicating that the current sequence of characters given is a match for the pattern if the input terminates there. A FSM terminates transitions when input is no longer given. It is then possible to transition away from a success state to a state where a pattern is no longer matched.

This is useful in the realm of programming languages and compilation as it allows us to identify patterns like strings, identifiers, numbers, delimiters, and keywords that make up statements which can be translated into machine code a processor understands how to execute.

Context Free Grammar

A context free grammar (CFG) defines a set of non-terminal and terminal rules by which a sentence (or statement in programming) can be formed in the grammar's language (Lambert & Louden, 2012). The phrases (or patterns of terminals/non-terminals) that form the grammar are called production rules. The transitions from a token to a non-termimal or vice-versa is called a derivation.

Figure 1. *Example CFG*

```
(1)  sentence → noun-phrase verb-phrase .
(2)  noun-phrase → article noun
(3)  article → a | the
(4)  noun → girl | dog
(5)  verb-phrase → verb noun-phrase
(6)  verb → sees | pets
```

Note. Obtained from *Programming Langauges* (3rd Ed.), Lambert & Louden, 2012, Pg. 209, Cengage Learning.

Figure 2. *Example Derivation*

```
sentence ⇒ noun-phrase verb-phrase  . (rule 1)
         ⇒ article noun verb-phrase . (rule 2)
         ⇒ the  noun verb-phrase  . (rule 3)
         ⇒ the girl  verb-phrase . (rule 4)
         ⇒ the girl  verb noun-phrase . (rule 5)
         ⇒ the girl  sees  noun-phrase . (rule 6)
         ⇒ the girl  sees  article noun . (rule 2)
         ⇒ the girl  sees  a  noun . (rule 3)
         ⇒ the girl  sees  a  dog  . (rule 4)
```

Note. Obtained from *Programming Langauges* (3rd Ed.) by Lambert & Louden, 2012, Pg. 209, Cengage Learning.

In Figure 2, one can observe the process of deriving a sentence from a production rule containing two non-terminals. By referencing our production rule-set, we can sequentially replace each non-terminal with a terminal rule, terminating in a valid sequence of tokens (terminals), forming a sentence in the given language.

Section II. Stages of the Compilation Process

As an overview, the stages of the compilation process are lexical analysis, syntax analysis, semantic analysis, intermediate representation (IR) code generation, IR code optimization, and assembly code generation for this course.

Lexical Analysis

In the initial stage of compilation, source code (i.e. a string of text from a file or command-line argument) is passed to a lexical analyzer (also called a scanner or lexer), by character, in sequence (Aho et. al., 2007). The lexical analyzer utilizes logic rooted in concepts of finite automata and regular expression string pattern matching to identify tokens in the programming

langauge rules defined by the programmer. Flex will match the longest sequence for a token if there are multiple matches (Paxson et. al., 2001).

Syntax Analysis

As the lexical anylzer identifies tokens, they are passed to a syntax analyzer, also reffered to as a parser (Aho et. al., 2007). The parser then applies rules from a context-free grammar to construct non-terminal symbols from the terminal symbols (tokens) given in sequence by the lexer. As it identifies non-terminal symbols, patterns of tokens, it constructs an abstract syntax tree where the tokens are the leaves. The AST represents derivations of tokens to non-terminals, up to the start-symbol of a grammar.

In addition to the AST, the parser is reponsible for generating a symbol table containing the variables with the code, including their identifiers, types, scopes, and values.

Semantic Analysis

Given the AST and symbol table by the parser, the semantic analyzer is responsible for traversing the AST to detect semantic errors, namely usage of undeclared identifiers, redeclaration of identifiers, type mismatch in assignments, type mismatch in expressions, type mismatch in data collection indexing, and out of bounds errors in data collection indexes, among others (Aho et. al., 2007).

Intermediate Code Representation

In the next stage of compilation, the AST is simplified into a Three-Address Code (TAC) intermediate representation that has a similar composition to assembly code syntax (Aho et. al. , 2007).

IR Code Optimization
In order to improve the execution time of the final assemnly code, optimizations are performed upon the IR code including constant folding, dead code elmination, local reduction in strength, and more.

Assembly Code Generation

In the final stage of compilation, the IR code is converted into assembly code which can be converted into machine code executable and be run locally. This involves assigning literals and variables present in the TAC to registers and managing register allocation as a whole, so as not to run out of available registers (Aho et. al., 2007).

In more complex compiler implementations, a linker is also erquired to link various libraries of programmign code and multiple soruce code files.

Section III. Verification of Software Installation

Installation of Flex


```
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ flex --version
flex 2.6.4
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ flex -help
Usage: flex [OPTIONS] [FILE]...
Generates programs that perform pattern-matching on text.

Table Compression:
  -Ca, --align      trade off larger tables for better memory alignment
  -Ce, --ecs        construct equivalence classes
  -Cf               do not compress tables; use -f representation
  -CF               do not compress tables; use -F representation
  -Cm, --meta-ecs   construct meta-equivalence classes
  -Cr, --read       use read() instead of stdio for scanner input
  -f, --full        generate fast, large scanner. Same as -Cfr
  -F, --fast        use alternate table representation. Same as -CFr
  -Cem              default compression (same as --ecs --meta-ecs)

Debugging:
  -d, --debug       enable debug mode in scanner
  -b, --backup      write backing-up information to lex.backup
  -p, --perf-report write performance report to stderr
  -s, --nodefault   suppress default rule to ECHO unmatched text
  -T, --trace       flex should run in trace mode
  -w, --nowarn      do not generate warnings
```

Screenshot verifies that Flex is installed on a linux environment with the ourput of the flex command with the –help and –version arguments.

Installation of Bison


```
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ bison --version
bison (GNU Bison) 3.8.2
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ bison --help
Usage: bison [OPTION]... FILE
Generate a deterministic LR or generalized LR (GLR) parser employing
LALR(1), IELR(1), or canonical LR(1) parser tables.

Mandatory arguments to long options are mandatory for short options too.
The same is true for optional arguments.

Operation Modes:
  -h, --help              display this help and exit
  -V, --version           output version information and exit
      --print-localedir   output directory containing locale-dependent data
                          and exit
```

Screenshot verifies that Bison is installed on a linux environment with the output of the bison command with the –help and –version arguments.

Installation of a Linux environment


```
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ lsb_release -a
LSB Version:    n/a
Distributor ID: Arch
Description:    Arch Linux
Release:        rolling
Codename:       n/a
[ A dayi ][ O tufdesktop ][ ~ ][ 🖥 ]  ♡ 
```

Screenshot verifies access to a linux development environment with the output of the lsb_release command with the -a argument displaying the OS version.

Section IV. Lexer Implementation with Flex

Lexer defines rules in Flex scripting language as per the grammar defined in the course materials. Rudimentary lexical error logging is implemented for malformed identifiers and unidentifiable characters, denoting line number.

Characters for arithemetic operations, keywords, delimiters, and punctuators are defined as tokens. A pattern to identiy a number (currently just a natural number) is established. Return statements are implemented and commented out to faciliate communication with the parser in the following stage of compilation using the Flex/Bison comptabliity workflow.

Lexer outputs token type, string pattern, and the line on which it was identified.

Source Code

```
int k = 4.0 + 2.0;

int u [ ];
float k;
int u  = 1;
int 4u = j; // Illegal identifier - starts with digit
if ( 1 == u)
{
    write (k);
}
else if (u > 1)
{
    write (u / 4);
} else
{
    write(u);
}

if ( u > 0 && u <= 5 ) { write (u + 1);}
float x = u + 1 - 3 / 4 * ( 1 + 1);

int u = 0;
while(u < 5)
{
    u = u + 1;
}

int hellofunc(int x, float y)
{
    return ( x + y);
}

hellofunc(u, x);
```

Lexer Output

Ln.1: LEXER: INTEGER TYPE KEYWORD : int

Ln.1: LEXER : IDENTIFIER : k
Ln.1 : LEXER : ASSIGN OP: =
Ln.1: LEXER : FLOAT: 1.0
Ln.1 : LEXER : PLUS OP: +
Ln.1: LEXER : FLOAT: 2.0
Ln.1 : LEXER : SEMICOLON: ;
Ln.3: LEXER: INTEGER TYPE KEYWORD : int
Ln.3: LEXER : IDENTIFIER : u
Ln.3 : LEXER : OPEN BRACKET: [
Ln.3 : LEXER : CLOSING BRACKET:]
Ln.3 : LEXER : SEMICOLON: ;
Ln.4: LEXER : FLOAT TYPE KEYWORD : float
Ln.4: LEXER : IDENTIFIER : k
Ln.4 : LEXER : SEMICOLON: ;
Ln.5: LEXER: INTEGER TYPE KEYWORD : int
Ln.5: LEXER : IDENTIFIER : u
Ln.5 : LEXER : ASSIGN OP: =
Ln.5 : LEXER : INT: 1
Ln.5 : LEXER : SEMICOLON: ;
Ln.6: LEXER: INTEGER TYPE KEYWORD : int
Ln.6: LEXER : Lexical Error: Illegal Identifier: 4u
Ln.6 : LEXER : ASSIGN OP: =
Ln.6: LEXER : IDENTIFIER : j
Ln.6 : LEXER : SEMICOLON: ;
Ln.6 : LEXER : // COMMENT: // Illegal identifier - starts with digit
Ln.7: LEXER: IF KEYWORD : if
Ln.7 : LEXER : OPEN PARENTHESIS: (
Ln.7 : LEXER : INT: 1
Ln.7: LEXER: LOGICAL EQUAL TO : ==
Ln.7: LEXER : IDENTIFIER : u
Ln.7 : LEXER : CLOSING PARENTHESIS:)
Ln.8 : LEXER : OPEN CURLY BRACE: {
Ln.9 : LEXER : PRINT KEYWORD: write
Ln.9 : LEXER : OPEN PARENTHESIS: (
Ln.9: LEXER : IDENTIFIER : k
Ln.9 : LEXER : CLOSING PARENTHESIS:)
Ln.9 : LEXER : SEMICOLON: ;
Ln.10 : LEXER : CLOSING CURLY BRACE: }
Ln.11: LEXER: ELSE KEYWORD : else
Ln.11: LEXER: IF KEYWORD : if
Ln.11 : LEXER : OPEN PARENTHESIS: (
Ln.11: LEXER : IDENTIFIER : u
Ln.11: LEXER: LOGICAL GREATER THAN : >
Ln.11 : LEXER : INT: 1
Ln.11 : LEXER : CLOSING PARENTHESIS:)
Ln.12 : LEXER : OPEN CURLY BRACE: {
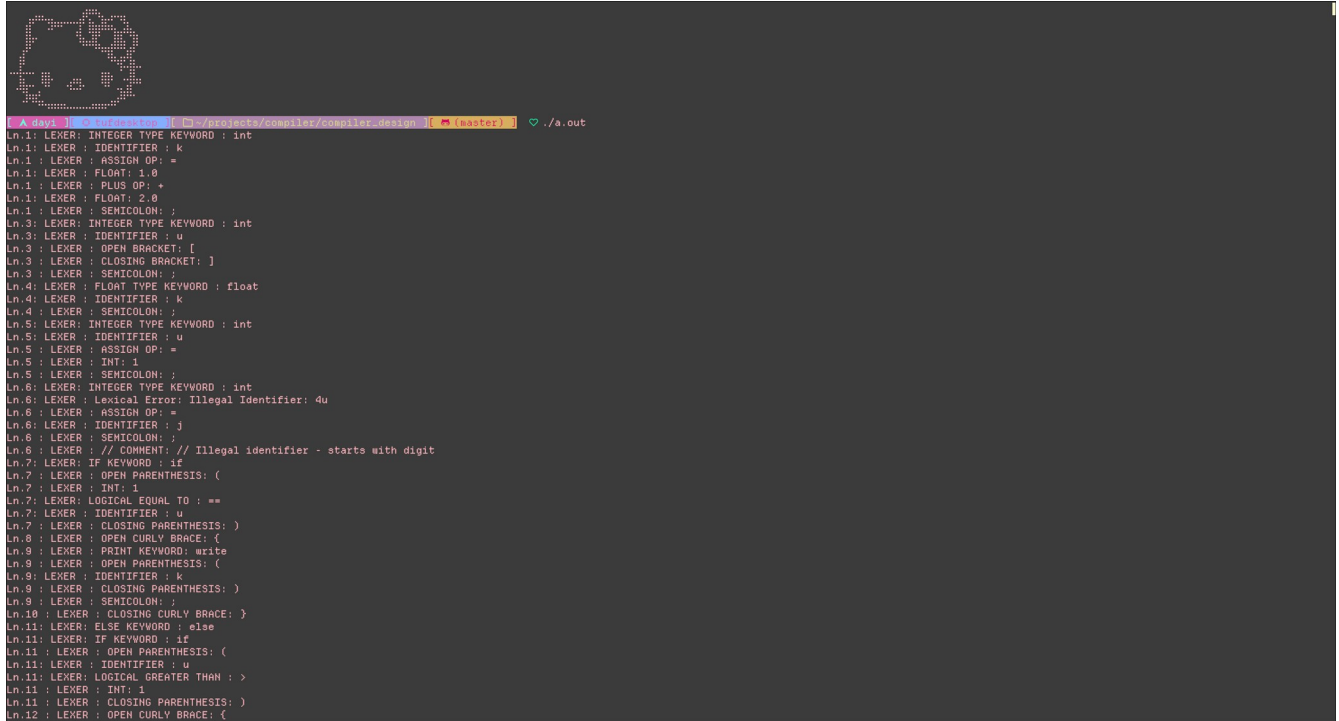Ln.13 : LEXER : PRINT KEYWORD: write

Ln.13 : LEXER : OPEN PARENTHESIS: (
Ln.13: LEXER : IDENTIFIER : u
Ln.13 : LEXER : DIV OP: /
Ln.13 : LEXER : INT: 4
Ln.13 : LEXER : CLOSING PARENTHESIS:)
Ln.13 : LEXER : SEMICOLON: ;
Ln.14 : LEXER : CLOSING CURLY BRACE: }
Ln.14: LEXER: ELSE KEYWORD : else
Ln.15 : LEXER : OPEN CURLY BRACE: {
Ln.16 : LEXER : PRINT KEYWORD: write
Ln.16 : LEXER : OPEN PARENTHESIS: (
Ln.16: LEXER : IDENTIFIER : u
Ln.16 : LEXER : CLOSING PARENTHESIS:)
Ln.16 : LEXER : SEMICOLON: ;
Ln.17 : LEXER : CLOSING CURLY BRACE: }
Ln.19: LEXER: IF KEYWORD : if
Ln.19 : LEXER : OPEN PARENTHESIS: (
Ln.19: LEXER : IDENTIFIER : u
Ln.19: LEXER: LOGICAL GREATER THAN : >
Ln.19 : LEXER : INT: 0
Ln.19: LEXER: LOGICAL AND : &&
Ln.19: LEXER : IDENTIFIER : u
Ln.19: LEXER: LOGICAL LESS THAN OR EQUAL : <=
Ln.19 : LEXER : INT: 5
Ln.19 : LEXER : CLOSING PARENTHESIS:)
Ln.19 : LEXER : OPEN CURLY BRACE: {
Ln.19 : LEXER : PRINT KEYWORD: write
Ln.19 : LEXER : OPEN PARENTHESIS: (
Ln.19: LEXER : IDENTIFIER : u
Ln.19 : LEXER : PLUS OP: +
Ln.19 : LEXER : INT: 1
Ln.19 : LEXER : CLOSING PARENTHESIS:)
Ln.19 : LEXER : SEMICOLON: ;
Ln.19 : LEXER : CLOSING CURLY BRACE: }
Ln.20: LEXER : FLOAT TYPE KEYWORD : float
Ln.20: LEXER : IDENTIFIER : x
Ln.20 : LEXER : ASSIGN OP: =
Ln.20: LEXER : IDENTIFIER : u
Ln.20 : LEXER : PLUS OP: +
Ln.20 : LEXER : INT: 1
Ln.20 : LEXER : MINUS OP: -
Ln.20 : LEXER : INT: 3
Ln.20 : LEXER : DIV OP: /
Ln.20 : LEXER : INT: 4
Ln.20 : LEXER : MUL OP: *
Ln.20 : LEXER : OPEN PARENTHESIS: (
Ln.20 : LEXER : INT: 1

Ln.20 : LEXER : PLUS OP: +
Ln.20 : LEXER : INT: 1
Ln.20 : LEXER : CLOSING PARENTHESIS:)
Ln.20 : LEXER : SEMICOLON: ;
Ln.22: LEXER: INTEGER TYPE KEYWORD : int
Ln.22: LEXER : IDENTIFIER : u
Ln.22 : LEXER : ASSIGN OP: =
Ln.22 : LEXER : INT: 0
Ln.22 : LEXER : SEMICOLON: ;
Ln.23: LEXER: WHILE KEYWORD : while
Ln.23 : LEXER : OPEN PARENTHESIS: (
Ln.23: LEXER : IDENTIFIER : u
Ln.23: LEXER: LOGICAL LESS THAN : <
Ln.23 : LEXER : INT: 5
Ln.23 : LEXER : CLOSING PARENTHESIS:)
Ln.24 : LEXER : OPEN CURLY BRACE: {
Ln.25: LEXER : IDENTIFIER : u
Ln.25 : LEXER : ASSIGN OP: =
Ln.25: LEXER : IDENTIFIER : u
Ln.25 : LEXER : PLUS OP: +
Ln.25 : LEXER : INT: 1
Ln.25 : LEXER : SEMICOLON: ;
Ln.26 : LEXER : CLOSING CURLY BRACE: }
Ln.28: LEXER: INTEGER TYPE KEYWORD : int
Ln.28: LEXER : IDENTIFIER : hellofunc
Ln.28 : LEXER : OPEN PARENTHESIS: (
Ln.28: LEXER: INTEGER TYPE KEYWORD : int
Ln.28: LEXER : IDENTIFIER : x
Ln.28 : LEXER : COMA: ,
Ln.28: LEXER : FLOAT TYPE KEYWORD : float
Ln.28: LEXER : IDENTIFIER : y
Ln.28 : LEXER : CLOSING PARENTHESIS:)
Ln.29 : LEXER : OPEN CURLY BRACE: {
Ln.30 : LEXER : RETURN KEYWORD: return
Ln.30 : LEXER : OPEN PARENTHESIS: (
Ln.30: LEXER : IDENTIFIER : x
Ln.30 : LEXER : PLUS OP: +
Ln.30: LEXER : IDENTIFIER : y
Ln.30 : LEXER : CLOSING PARENTHESIS:)
Ln.30 : LEXER : SEMICOLON: ;
Ln.31 : LEXER : CLOSING CURLY BRACE: }
Ln.33: LEXER : IDENTIFIER : hellofunc
Ln.33 : LEXER : OPEN PARENTHESIS: (
Ln.33: LEXER : IDENTIFIER : u
Ln.33 : LEXER : COMA: ,
Ln.33: LEXER : IDENTIFIER : x
Ln.33 : LEXER : CLOSING PARENTHESIS:)

Ln.33 : LEXER : SEMICOLON: ;

Screenshot of Lexer Output

Screenshot depicts output of runnign executable produced by compiling lexer generated by Flex, with specification of input from file source.txt.

References

Aho, A. V., Ullman, J. D., Sethi, R., & Lam, M. S. (2007). *Compilers*
   (2nd ed.). Pearson Education.
Lambert, K. A. & Louden, K. C. (2012). *Programming languages.* (3rd ed.). Cengage Learning.
Paxson, V., Poskanzer, J., Gong, K., & Jacobson, V. (2001). *Flex – a scanner generator [Online
   Article].* University of California.
      https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.htm