CS323 Lab 3

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Flex Tutorial

- Introduction to Project (Phase 1)

The Lexical-Analyzer Generator `Lex`

- `Lex`, or a more recent tool `Flex`, allows one to specify a lexical analyzer by specifying regexps to describe patterns for tokens
- Often used with Yacc/Bison to create the frontend of compiler

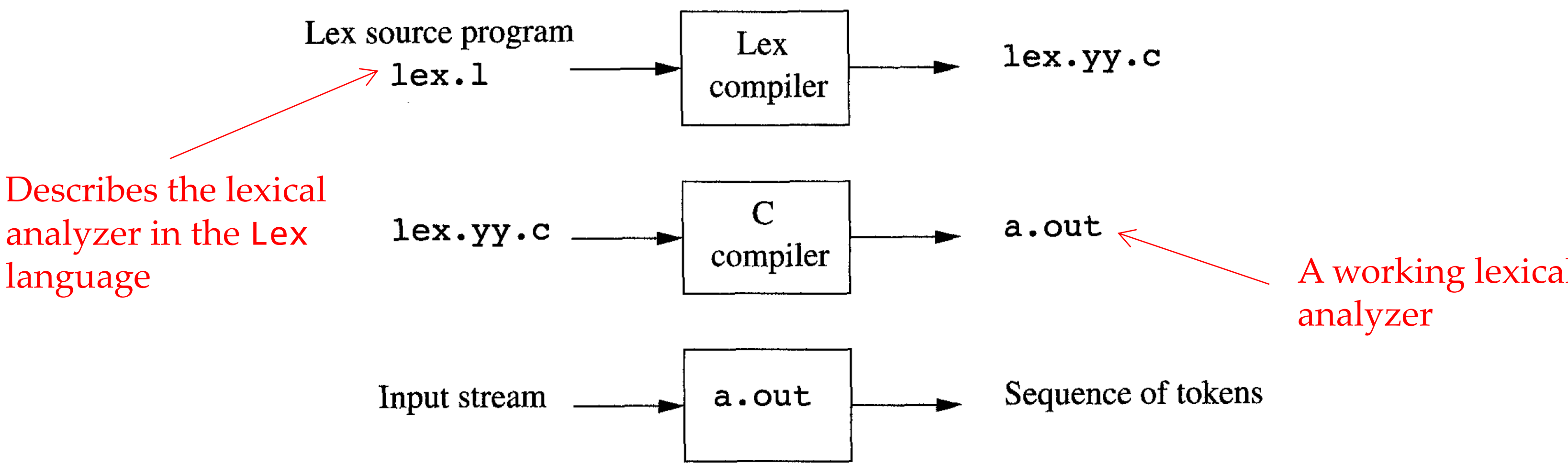


Figure 3.22: Creating a lexical analyzer with Lex

Structure of Lex Programs

- **A Lex program has three sections separated by %%**
  - Declaration (声明)
    - Variables, constants (e.g., token names)
    - Regular definitions
  - Translation rules (转换规则) in the form "Pattern {Action}"
    - Each pattern (模式) is a regexp (may use the regular definitions of the declaration section)
    - Actions (动作) are fragments of code, typically in C, which are executed when the pattern is matched
  - Auxiliary functions section (辅助函数)
    - Additional functions that can be used in the actions

Lex Program Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Anything in between %{ and }% is copied directly to `lex.yy.c`.

In the example, there is only a comment, not real C code to define manifest constants

Regular definitions that can be used in translation rules

Section separator

Lex Program Example Cont.

Continue to recognize other tokens

Return token name to the parser

```
{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%
```

Literal strings*

Place the lexeme found in the symbol table

A global variable that stores a pointer to the symbol table entry for the lexeme. Can be used by the parser or a later component of the compiler.

* The characters inside have no special meaning (even if it is a special one such as *).

Lex Program Example Cont.

- Everything in the auxiliary function section is copied directly to the file `lex.yy.c`
- Auxiliary functions may be used in actions in the translation rules

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Variables defined and set automatically by the lexical analyzer `Lex` generates

Conflict Resolution

- When the generated lexical analyzer runs, it analyzes the input looking for prefixes that match any of its patterns. *
- **Rule 1:** If it finds multiple such prefixes, it takes the longest one
  - The analyzer will treat <= as a single lexeme, rather than < as one lexeme and = as the next
- **Rule 2:** If it finds a prefix matching different patterns, the pattern listed first in the `Lex` program is chosen.
  - If the keyword patterns are listed before identifier pattern, the lexical analyzer will not recognize keywords as identifiers

* See Flex manual for details (Chapter 8: How the input is matched) at http://dinosaur.compilertools.net/flex/

Flex

- Flex的前身是Lex。Lex是1975年由Mike Lesk和当时还在贝尔实验室做暑期实习的Eric Schmidt（前谷歌CEO），共同完成的一款基于Unix环境的词法分析程序生成工具。虽然Lex很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。

- 1987年伯克利实验室（隶属美国能源部的国家实验室）的Vern Paxson使用C语言重写Lex，并将这个新程序命名为Flex（Fast Lexical Analyzer Generator）。无论从效率上还是稳定性上，Flex都远远好于它的前辈Lex。

*我们在Linux下使用的是`Flex`在BSD License下的版本（和Bison不同，Flex不属于GNU计划）。

An Example Flex Program

- A word-count program (see the code under lab3/wc)
- Build the program with the following commands (or "`make wc`")
  - `flex lex.l` (you will see a lex.yy.c file generated)
  - `gcc lex.yy.c -lfl -o wc.out`

```
yepang@Ubuntu-LYP:~/Desktop/CS323-2021F/lab2/wc$ ./wc.out inferno3.txt
#lines  #words  #chars  file path
162     1088    6525    inferno3.txt
```

A Closer Look

```
%{
    // just let you know you have macros!
    // C macro tutorial in Chinese: http://c.biancheng.net/view/446.html
    #define EXIT_OK 0
    #define EXIT_FAIL 1

    // global variables
    int chars = 0;
    int words = 0;
    int lines = 0;
%}
letter [a-zA-Z]

%%
{letter}+ { words++; chars+=strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }

%%
int main(int argc, char **argv){
    char *file_path;
    if(argc < 2){
        fprintf(stderr, "Usage: %s <file_path>\n", argv[0]);
        return EXIT_FAIL;
    } else if(argc == 2){
        file_path = argv[1];
        if(!(yyin = fopen(file_path, "r"))){
            perror(argv[1]);
            return EXIT_FAIL;
        }
        yylex();
        printf("%-8s%-8s%-8s%s\n", "#lines", "#words", "#chars", "file path");
        printf("%-8d%-8d%-8d%s\n", lines, words, chars, file_path);
        return EXIT_OK;
    } else{
        fputs("Too many arguments! Expected: 2.\n", stderr);
        return EXIT_FAIL;
    }
}
```

The structure is the same as in a Lex program:

1. Declaration

2. Translation rules

3. Auxiliary functions

More on Flex patterns

Flex supports a rich set of conveniences:

Character classes	**`[0-9]`**	This means alternation of the characters in the range listed (in this case: **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**). More than one range may be specified, e.g. **`[0-9A-Za-z]`** as well as specifying individual characters, as with **`[aeiou0-9]`**.
Character exclusion	**^**	The first character in a character class may be **^** to indicate the complement of the set of characters specified. For example, **`[^0-9]`** matches any non-digit character.
Arbitrary character	**.**	The period matches any single character **except newline**.
Single repetition	**`x?`**	0 or 1 occurrence of **`x`**.

More on Flex patterns

Non-zero repetition	**x+**	**x** repeated one or more times; equivalent to **xx***.
Specified repetition	**x{n,m}**	**x** repeated between **n** and **m** times.
Beginning of line	^x	Match **x** at beginning of line only.
End of line	**x$**	Match **x** at end of line only.
Context-sensitivity	**ab/cd**	Match **ab** but only when followed by **cd.** The lookahead characters are left in the input stream to be read for the next token.
Literal strings	**"x"**	This means **x** even if **x** would normally have special meaning. Thus, **"x*"** may be used to match **x** followed by an asterisk. You can turn off the special meaning of just one character by preceding it with a backslash, .e.g. **\\.** matches exactly the period character and nothing more.
Definitions	**{name}**	Replace with the earlier defined pattern called **name**. This kind of substitution allows you to re-use pattern pieces and define more readable patterns.

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf

Flex Exercise: C Identifier

- Count the occurences of valid C identifiers
  - A valid C identifier starts with an English letter or an underscore followed by any number of English letters, digits, or underscores
- We make some assumptions to simplify the task
  - Only these reserved words may appear: char, int, return, while, if, else
  - There are no preprocessor commands (e.g., #include <stdio.h>)
  - There are no function calls

Flex Exercise: C Identifier

- Please modify the lex.l under lab3/identifier directory
- Build the lexer
  - `make idcount`
- Run the counting program
  - `./idcount.out test.c`
- If you get the following output, your implementation is correct.

```
line 1: main
line 3: a
line 4: BBA
line 4: a_
line 5: _
line 7: a0
line 7: _
line 7: b0
line 8: _
line 8: b0
line 9: b
line 9: b1
line 9: b0
line 9: b2
line 10: c
There are 15 occurrences of valid identifiers
```

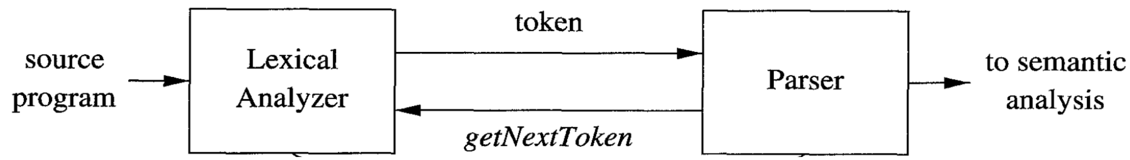
Outline

- Flex Tutorial
- Introduction to Project (Phase 1)

Project Goal

- Design & implement a compiler for SUSTech Programming Language (SPL), a Turing-complete C-like programming language without advanced features (e.g., macros, pointers)
- **Compiler input:** A piece of SPL source code
- **Compiler output:** MIPS32 assembly code (runnable in the Spim simulator)

Phase 1



- Implement a SPL parser, which can perform lexical analysis and syntax analysis on SPL source code
  - Flex will be used to implement the lexical analysis module
  - Bison will be used to implement the syntax analysis module
  - The syntax analysis module invokes the lexical analysis module during parsing
- Parser output:
  - For a syntactically valid SPL program, your parser should output the parse tree (will be introduced in Chapter 3)
  - Otherwise, your parser should output all lexical & syntax errors

SPL Specification

Allowed tokens:

```
INT      -> /* integer in 32-bits (decimal or hexadecimal) */
FLOAT    -> /* floating point number (only dot-form) */
CHAR     -> /* single character (printable or hex-form) */
ID       -> /* identifier */
TYPE     -> int | float | char
STRUCT   -> struct
IF       -> if
ELSE     -> else
WHILE    -> while
RETURN   -> return
DOT      -> .
SEMI     -> ;
COMMA    -> ,
ASSIGN   -> =
```

```
LT       -> <
LE       -> <=
GT       -> >
GE       -> >=
NE       -> !=
EQ       -> ==
PLUS     -> +
MINUS    -> -
MUL      -> *
DIV      -> /
AND      -> &&
OR       -> ||
NOT      -> !
LP       -> (
RP       -> )
LB       -> [
RB       -> ]
LC       -> {
RC       -> }
```

https://github.com/sqlab-sustech/CS323-2023F/blob/main/spl-spec/token.txt

SPL Specification

The grammar rules:

```
Stmt -> Exp SEMI
      | CompSt
      | RETURN Exp SEMI
      | IF LP Exp RP Stmt
      | IF LP Exp RP Stmt ELSE Stmt
      | WHILE LP Exp RP Stmt
```

https://github.com/sqlab-sustech/CS323-2023F/blob/main/spl-spec/syntax.txt

```
Exp -> Exp ASSIGN Exp
     | Exp AND Exp
     | Exp OR Exp
     | Exp LT Exp
     | Exp LE Exp
     | Exp GT Exp
     | Exp GE Exp
     | Exp NE Exp
     | Exp EQ Exp
     | Exp PLUS Exp
     | Exp MINUS Exp
     | Exp MUL Exp
     | Exp DIV Exp
     | LP Exp RP
     | MINUS Exp
     | NOT Exp
     | ID LP Args RP
     | ID LP RP
     | Exp LB Exp RB
     | Exp DOT ID
     | ID
     | INT
     | FLOAT
     | CHAR
```

Example

```
int test_1_r01(int a, int b)
{
  c = 'c';
  if (a > b)
  {
    return a;
  }
  else
  {
    return b;
  }
}
```

A syntactically valid program*

* Here, the vairable c is used without definition.
This error will be caught during semantic analysis.

The parse tree:

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: test_1_r01
        LP
        VarList (1)
          ParamDec (1)
            Specifier (1)
              TYPE: int
            VarDec (1)
              ID: a
          COMMA
          VarList (1)
            ParamDec (1)
              Specifier (1)
                TYPE: int
              VarDec (1)
                ID: b
        RP
      CompSt (2)
        LC
        StmtList (3)
          Stmt (3)
            Exp (3)
              Exp (3)
                ID: c
              ASSIGN
              Exp (3)
                CHAR: 'c'
            SEMI
          StmtList (4)
            Stmt (4)
              IF
              LP
              Exp (4)
                Exp (4)
                  ID: a
                GT
                Exp (4)
                  ID: b
              RP
              Stmt (5)
                CompSt (5)
                  LC
                  StmtList (6)
                    Stmt (6)
                      RETURN
                      Exp (6)
                        ID: a
                      SEMI
                  RC
              ELSE
              Stmt (9)
                CompSt (9)
                  LC
                  StmtList (10)
                    Stmt (10)
                      RETURN
                      Exp (10)
                        ID: b
                      SEMI
                  RC
        RC
```

Example

```
1  int test_1_r03()
2  {
3          int i = 0, j = 1;
4    float i = $;
5    if(i < 9.0){
6      return 1
7    }
8    return @;
9  }
```

```
Error type A at Line 4: unknown lexeme $
Error type B at Line 6: Missing semicolon ';'
Error type A at Line 8: unknown lexeme @
```