Crafting a Compiler with C (VI)

資科系
林偉川

Scanner generator

- How regular expressions and related information are presented to generators
- Obtain a lexical analyzer by generating automatically from regular description of tokens
- ScanGen is a scanner generator designed to be transportable and highly language and machine independent
- ScanGen produces tables that are used by a driver routine to create a complete scanner
- Example is the definition extends Micro's token to include real, string literals and error token

2

ScanGen token definition

- Input to ScanGen is divided into three sections each headed by a reserved word (Options, Class, Definition)
  - Options ➔ list of inputs, produced output tables, state optimization phase
  - Character classes for state transition
    - Transition table size = the number of states * the number of characters
    - Any character not mentioned in any character class definition is ignored
    - All RE and FA are defined in terms of character classes

3

ScanGen token definition

- Token in terms of RE using infix ”.” (catenation) and “,”(alternative), postfix (* | +), unary Not and Epsilon (λ)
- A basic pattern can be followed by a repetition operator such as b? for an optional b; b^* for a possibly empty sequence of b; and b^+ for a non-empty sequence of b
- Two composition operators: concatenation (invisible operator) such as ab; alternative (|) such as ab^* | cd? Matches anything that is matched by ab^* or by cd?

4

ScanGen token definition

- The repetition operators have the highest precedence; next comes the concatenation operator; and the | operator have the lowest precedence. How about ab^* | cd? ➔$(a(b^*))$ | $(c(d?))$
- Escape characters to force these characters to stand for themselves rather than being taken as operators ➔ prefix operator (\) such as * denotes *, \\ the backslash, another is the “, which is used to surround the escaped part: “*” denotes the *, “+?” denotes +?, “””” denotes “”
- A regular description is like a CFG in EBNF, with the restriction that non-terminal can be used before it has been fully defined

5

ScanGen token definition

- The regular description for the identifier is listed followed:
  letter → [a-zA-Z]
  digit → [0-9]
  underscore → _
  letter_or_digit → letter | digit
  underscored_tail → underscore letter_or_digit$^+$
  identifier → letter letter_or_digit* underscore_tail*
  ➔after some simplification
  identifier → [a-zA-Z] [a-zA-Z0-9]* (_ [a-zA-Z0-9]$^+$)*

6

ScanGen token definition

- For typical programming language, ScanGen produces 50 states and 30 character classes by using 2 dimensional array
- Each entry is 2 bytes ➔ 3000 bytes are used
- Table array is the best way for accessing the value

7

ScanGen token definition

Options

List, tables, optimize

8

ScanGen token definition

```
Class
    E = 'E', 'e';
    OtherLetter = 'A'..'D', 'F'..'Z', 'a'..'d', 'f'..'z';
    Digit = '0'..'9';
    Blank = ' ';
    Dot = '.';
    Plus = '+';
    Minus = '-';
    Equal = '=';
    Colon = ':';
    Comma = ',';
    Semicolon = ';';
    Lparen = '(';
    Rparen = ')';
    Quote = '"';
    Underscore = '_';
    Tab = 9;
    Linefeed = 10;
```

9

Parts of Regular expression

- Regular expression :
  - -- Not(Eol)*Eol // 註解
  - ID= L (L|D)*(_(L|D)+)* //變動
  - Lit=D+ . D+ //常數
  - RealLit = (D+(λ|.)) | (D*. D+) //實數常數
  - String = (" (Not(") | "")* ") //字串
- 3 distinct regulars are used to recognize integer, real, string literal

10

ScanGen token definition

```
Definition
    Token EmptySpace {0} = (Blank, Linefeed, Tab)+;
    Token Comment {0} =
        Minus . Minus . (Not(Linefeed))* . Linefeed;
  Letter = E, OtherLetter;
  Token Identifier {1} = Letter.(Letter, Digit, Underscore)*;
    Exception
      'BEGIN'   {4},
      'END'   {5},
      'Read'   {6},
      'Write'   {7};
    Token IntLit {2,1} = Digit+;
    Token RealLit {2,2} = IntLit.Dot.IntLit.
                          (Epsilon, E.(Epsilon, Plus, Minus).IntLit);
    Token StrLit {2,3} = Quote{Toss}.
                      (Not(Quote, Linefeed), Quote{Toss}. Quote)*.Quote{Toss};
    Token RunOnStringLit {3} = Quote{Toss}.
                    (Not(Quote, Linefeed), Quote{Toss}. Quote)*.Linefeed{Toss};
    Token LparenToken {8}  = Lparen;
    Token RparenToken {9}  = Rparen;
    Token SemicolonToken {10}  = Semicolon;
    Token CommaToken {11}  = Comma;
    Token AssignOp {12}  = Colon . Equal;
    Token PlusOp {13}  = Plus;
    Token MinusOp {14}  = Minus;
```

11

Major token definition in ScanGen

- Token definition
  Token name {major, minor} = RE;
  - Major token code identifies the token to the parser
    - The same tokens are recognized by different RE
    - Major code is 0 means this token is to be delete rather than passed ➔ noisy token like comment
    - Blank token (composed of blanks, tabs, and newlines) that consumes the space between interesting tokens
  - Minor token code (optional) may be used by semantic processing routine (token subclasses may have different semantic interpretations)

12

Major token definition in ScanGen

- Token definition may have an exception clause (headed by Except) ➔ reserved words is exception for identifier

  This can reduce the size of FA needed by the scanner
- Each exception (defined by a literal with major and minor codes) must match the associated RE
- Character class names or Not expressions may be suffixed with {Toss}
- When a character class name marked with {Toss} is matched in a regular expression, the character matched is tossed away rather than saved

13

Major token definition in ScanGen

- All character class names and Not expressions not marked with {Toss} are implicitly assumed to save any characters they match
- Implement a scanner by using a driver routine with the ScanGen generated tables

14

Major token definition in ScanGen

- A FA must look ahead while scanning ➔ examine a character that may not be part of the token to verify that an entire token has been seen
- Scanning an identifier, the scanner must keep reading until a character that is not part of that identifier (like ; or " ")
- The lookahead character should not be lost because it may be needed as the head of the next token to be scanned (getc(FILE *) and ungetc(int, FILE*))

15

Driver routine for ScanGen

- gtec() return the current input character being processed and advances the position in the file to the next character, consuming the current character
- ungetc() puts a character back on the file being read from, so the next call to getc() will return that character ➔ buffering mechanism
- The table that controls scanning is the action table
- Action[state][ch] indicates a move action (keeping scanning) or a halt action (a token has been recognized) ➔ lookahead needed to be consulted and scanned characters may need to be either tossed or retained

16

Action table values of ScanGen

- Action table contains 6 different value
  - ERROR
    - Token error. No valid token can be recognized
  - MOVEAPPEND
    - Move to the next state, consume the current character and append it to token string
  - MOVENOAPPEND
    - Move to the next state, consume the current character and do not append it to token string

17

Action table value of ScanGen

- Action table contains 6 different value
  - HALTAPPEND
    - Consume the current character and append it to token string ➔ a valid token is found
  - HALTNOAPPEND
    - Consume the current character and do not append it to token string ➔ a valid token is found
  - HALTREUSE
    - Do not consume the current character and a valid token is found

18

Action table value of ScanGen

- For actions moving, the next state to visit is stored in the next_state[state][ch], 0 indicates no next state
- For halt actions, the major and minor codes are obtained by calling

  lookup_code(state, cur_char, major, minor) ➔ for major and minor code processing

  After looking up the major and minor codes, exception are handled by calling

  check_exceptions(major, minor, next_token) ➔ for exception handling

19

Action table value of ScanGen

case 1. If major and minor indicate a token class that has exceptions, token_text is examined to see if it really is an exception. if it is, major and minor are reset to the code of exception

case 2. If token_text is not an exception, the original codes are returned

20

ScanGen Driver Program (I)

```
#define reset() { ind=0; token_text[ind]='\0'; \
   state=STARTSTATE; }
extern enum scan_state
   next_state[NUMSTATES][NUMCHARS];
extern FILE *srcfile;
void scanner(codes *major, codes *minor, char *token_text) {
  enum scan_state state;
  int ind, c;
  reset();
```

21

ScanGen Driver Program (I)

```
  while (TRUE) {
    c=getc(srcfile);
    switch (action[state][c]) {
    case ERROR:
       /* do lexical error recovery ungetc() …*/
       break;
    case MOVEAPPEND:
       state=next_state[state][c];
       token_text[ind++]=c; break;
```

22

ScanGen Driver Program (I)

```
case MOVENOAPPEND:
    state=next_state[state][c]; break;
 case HALTAPPEND:
    lookup_codes(state, c, major, minor);
    token_text[ind++]=c; token_text[ind++]='\0';
    check_exceptions(major, minor, token_text);
    if (*major == 0) {  // do not return this token
      reset(); continue;
    }
    return;
```

23

ScanGen Driver Program (I)

```
case HALTNOAPPEND:
    lookup_codes(state, c, major, minor);
    token_text[ind]='\0';
    check_exceptions(major, minor, token_text);
    if (*major == 0) {  // do not return this token
      reset(); continue;
    }
    return;
```

24

ScanGen Driver Program (I)

```
  case HALTREUSE:
      lookup_codes(state, c, major, minor);
      token_text[ind]='\0';
      check_exceptions(major, minor, token_text);
      ungetc(c, srcfile);
      if (*major == 0) {   // do not return this token
        reset(); continue;
      }
      return;
    }
  }
}
```

25

Lex

- Produce an entire scanner module that can be compiled and linked with other compiler module
- Broader in the approach than ScanGen
- Lex associates regular expressions with arbitrary code fragments
- When an expression is matched, the code segment is executed
- No major/minor code or toss flags are provided because all these can be realized in the user-written code segment

26

Lex

- A Lex definition of a scanner is equivalent to the definition of ScanGen
- Lex's section is separated by %% delimiter
- The first section defines character classes and auxiliary regular expression
- Character classes are delimited by [and]
- Except for \, ^, and –, individual characters are unquoted and catenated without any separator

27

Lex token

- [xyz] represents the class that can match an x, y, or z
- [x-z] is the same as [xyz], - is the range of character
- \ is the escape character➔\n, \t, \\, \10 (octal 10)
- [^xy] is the character class that matches all characters except x and y. ^ is the complement operator
- [ab][cd] can match ad, ac,bc, bd
- begin can be matched by "begin" or [b][e][g][i][n]

28

Lex token

- | is the alternation operator
- Parentheseses can be used to control grouping of subexpression ➔ match reserved word 'end'

  ➔ (E|e)(N|n)(D|d)
- * (Kleene closure) and + (Positive closure) and ? (optional inclusion) ➔ Expr? matches Expr 0 times or once ➔ Expr | λ
- { } is the macroexpansion of a symbol defined in the first section. E.g. Digit ➔[0-9], {Digit}+ ➔ [0-9]+

29

Lex definition for Extended Micro

```
E             [Ee]
OtherLetter   [A-DF-Za-df-z]
Digit         [0-9]
Letter        {E} | {OtherLetter}
IntLit        {Digit}+
%%
[ \t\n]+                                        { /* delete */          }
[Bb][Ee][Gg][Ii][Nn]                            { minor=0; return(4);  }
[Ee][Nn][Dd]                                    { minor=0; return(5);  }
[Rr][Ee][Aa][Dd]                                { minor=0; return(6);  }
[Ww][Rr][Ii][Tt][Ee]                            { minor=0; return(7);  }
{Letter}({Letter} | {Digit} | _)*               { minor=0; return(1);  }
{IntLit}                                        { minor=1; return(2);  }
({IntLit}[.]{IntLit})({E}[+-]?{IntLit})?        { minor=2; return(2);  }
\"([^\"\n] | \"\")*\"                           { stripquotes();
                                                  minor=3; return(2);  }
\"([^\"\n] | \"\")*\n                           { stripquotes();
                                                  minor=0; return(3);  }
"("                                             { minor=0; return(8);  }
")"                                             { minor=0; return(9);  }
";"                                             { minor=0; return(10); }
","                                             { minor=0; return(11); }
":="                                            { minor=0; return(12); }
"+"                                             { minor=0; return(13); }
"-"                                             { minor=0; return(14); }
%%
```

Lex second section

- A table of RE and corresponding command
- When an expression is matched, its associated command is executed
- An input sequence is matched is stored in the string variable yytext whose length yylength
- Commands may alter yytext in any way and then write the altered text to the output file
- yylex() may be called from the parser, value return is the token code of token scanned by Lex

31

Lex second section

- Token like white space can be deleted simply by having their associated command not returning anything. Scanning continues until a command with a return in it is executed
- No special provisions for exception list as ScanGen does
- An identifier is recognized, a subprogram check_exceptions() is called to recognize exceptions and return the correct token code

32

Lex second section

- Lex allows RE to overlap (match common input sequences)
- In the case of overlap, two rules apply
- The longest possible match is performed. The Lex automatically performs buffering. If two expressions match the same string, the earlier expression (in order of definition in the Lex specification) is preferred

33

Lex second section

- Exceptions are handled by placing special expressions before the expression that matches the general pattern
- The major code is returned by yylex(), a minor code can be return by assigning to a shared variable ➔ no special provision is made for major or minor token numbers
- No tossing mechanism, just reprocess the token text by stripquotes()

34

Lex second section

- EOF token is signaled by having yylex() return the integer value zero
- yylex() uses 3 user-defined functions to handle character I/O
  - input() retrieve a single character, 0 on end of file
  - output(c) write a single character to the output
  - unput(c) put a single character back on the input to be re-read
  - yylex() encounters EOF, it calls yywrap() ➔ wrap-up input processing (return 1 if there is no more input and 0 to arrange input() to provide more input)

35

Lex second section

- Lex supplies read/write characters from/to the standard input/output
- Compiler write may supply the input(), output(), unput(), and yywrap() functions
- Lex requires code segments be written in C, this makes it less generally usable than ScanGen which produces just tables
- Lex is just the kind of scanner generator tool available to compiler writer

36

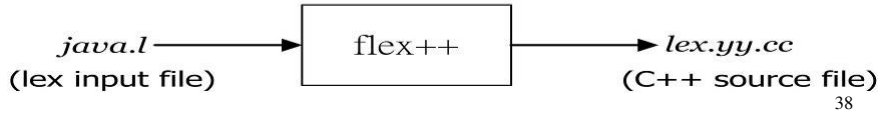
Lex definition for Extended Micro

```
void stripquotes(void) {
  int frompo, topos=0, numquotes=2;
  for (frompos=1; frompos<yyleng; frompos++) {
     yytext[topos++]=yytext[frompos];
     if (yytext[frompos]==‘”’ && yytext[frompos+1]==‘”’){
       frompos++; numquotes++;
     }
  }
  yyleng -= numquotes;
  yytext[yyleng]=‘\0’;
}
```

37

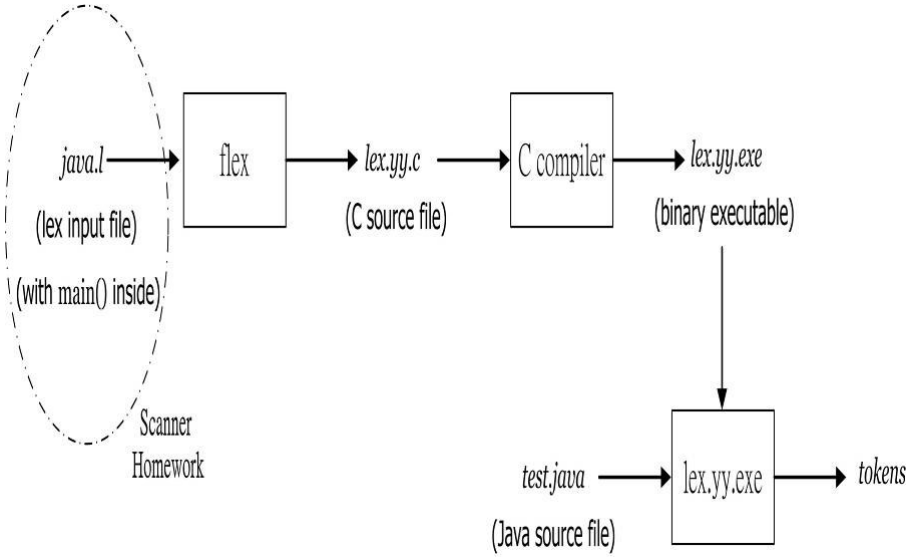
flex - fast lexical analyzer generator

- Flex is a tool for generating scanners
- flex 是可以自由取得的 lex 版本, 它是隨著 4.4BSD 一起散佈, 也屬於 GNU 計 畫.
- 用 flex 產生 C 的原始碼, 用 flex++ 產生 C++ 的原始碼.
- Flex source is a table of regular expressions and corresponding program fragments.
- Generates ***lex.yy.c*** which defines a routine ***yylex()***



38

要寫一個 Java 的 Lexical Analyzer



Format of the Input File

- The flex input file consists of three sections, separated by a line with just %% in it:

  definitions
  %%
  rules
  %%
  user code

- definition：使用者自己定義的變數，都放在這個地方
- rules：parser對token match的規則
- user code：最後產生的lex.yy.c最底下會有一模一樣的code

40

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

  name definition
- Example:

  DIGIT [0-9]

  ID [a-z][a-z0-9]*

41

Definitions Section

- 在此區間裡可以宣告一些在rule中的code要使用的變數(寫法跟c一模一樣)，而這些code必須用%{ 與 %} 將跨行的code包起來，因為在這個區間的code都會被完完整整、一字不漏地output至lex.yy.c檔中，所以在compile lex.yy.c檔時，才不會產生error!

42

Definitions Section

- 也可以宣告一些rule的變數，讓rule的寫法更簡潔
  寫法為：***name*** ***definition***

Ex :

```
number     [0-9]+
identifier   [a-zA-Z][a-zA-Z_0-9]*
%%
{number} printf("%s this token is a number\n", yytext);
{identifier} printf("%s this token is a identifier\n", yytext);
```

意思其實就是...

```
{[0-9]+} printf("%s this token is a number\n", yytext);
{[a-zA-Z_][a-zA-Z_0-9]*} printf("%s this token is a
identifier\n", yytext);
```

43

Rules Section

- The rules section of the flex input contains a series of rules of the form:(要對input file切token的規則全寫在這裡。寫法的規則是：)

  pattern action

- Example:

  {ID} printf("An identifier: %s\n", yytext);

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

44

Action

- If the action contains a '{', the action spans till the balancing '}' is found, as in C.
- An action consisting only of a vertical bar ('|') means "same as the action for the next rule."
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

45

Action

- action則是當pattern match後，執行相對應的code(跟c一模一樣) 會原封不動地寫入output file。若action的code太多，則可以用{ }跨行將code包起來

  Ex:

  ```
  [0-9]+ ECHO;printf("this is a number!\n");
  ```

  等同於...

  ```
  [0-9]+ {
        ECHO;
        printf("this is a number!\n");
          }
  ```

  有一個特別的word可以用在action中，*ECHO* 可以印出yytext(match pattern的字串)中的內容至output中

46

Precedence Problem

- For example: a "<" can be matched by "<" and "<=".
- The one matching most text has higher precedence.
- If two or more have the same length, the rule listed first in the flex input has higher precedence.

47

User Code Section

- The user code section is simply copied to *lex.yy.c* verbatim.
- The presence of this section is optional; if it is missing, the second %% in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in %{ and %} is copied verbatim to the output (with the %{}'s removed).

48

計算字數和行數的範例

- Lex的input file必須是*.l 的檔案 (副檔名為l) 接著只要輸入指令：flex test.l，然後Lex就會自動產生一個output file：lex.yy.c
- 接著只要compile這個lex.yy.c 就可以執行這個token parser了：gcc lex.yy.c -ll，而-ll是為了include lex的library
- 用範例於下一頁去產生 *lex.yy.c*
- 使用yacc 範例 檔名為??.y

49

計算字數和行數的範例

```
%{     int num_lines = 0, num_chars = 0;  %}

%%
\n       ++num_lines; ++num_chars;
.        ++num_chars;
%%
main()  {
   yylex();
   printf( "# of lines = %d, # of chars = %d\n", num_lines,
   num_chars );
}
```

50