Lex and Yacc Specifications of C

Mini Assignment -2

Kartik Srinivas - ES20BTECH11015

A Course Homework Assignment

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

April 17, 2022

1 Lexical Specifications

1.1 Checking the type!

This was an interesting example in the lexical specification

```
{L}({L}|{D})*   { count(); return(check_type()); }
```

Note that the Language clearly describes something that starts with a letter and then contain either a letter or a digit. But i could also accept the type "int" and also accept the string "tin"? how do I defer between these?. To do this , we need to call some function that checks whether this is an identifier like "tin" or an actual data type like "int". But since Lex takes the longest match and the first one is preferred if the match is of same length, "int" will be taken under TYPE-NAME instead of IDENTIFIER since the TYPENAME identifier rule appears first!

1.1.1 What it should be:

```
int check_type()
{
  if (yytext == type_name)
    return(TYPE_NAME);
  else return(IDENTIFIER);

}
```

1.1.2 What it is:

```
int check_type()
{
  return(IDENTIFIER); //because ''type_name'' rules are above !
}
```

1.2 String Literal

The regular expression for this is quite tricky: Its either a single letter or multiple letters contained within a " ", but to show a " , we need to use an escape sequence in Lex to show the quotations and anything else (other than a quotation can remain inside).

```
L?\"(\\.|[^\\"])*\"
```

1.3 Comments

The C comment starts with a "/*" . A specific subroutine is written to parse the entire thing until another '*" is seen, then according to what comes after the '*" it makes a decision to stop according to whether the next one is a '/' or not.

```
"/*"        { comment(); }

"------------FUNCTION---------------"
```

```
comment()
{
  char c, c1;

loop:
  while ((c = input()) != '*' && c != 0)
    putchar(c);

  if ((c1 = input()) != '/' && c != 0)
  {
    unput(c1);
    goto loop;
  }

  if (c != 0)
    putchar(c1);
}
```

1.4 The Count Function

Each time any rule is matched , the maximum number of columns occupied by the text as input is also recorded. There is an iteration from the start to the end of the string and the each time a newline is encountered it resets columns to zero , while the addition of a new character increases the number of columns by one. At the end the re is an ECHO; command to print what exactly was parsed as a token

```
void count()
{
  int i;

  for (i = 0; yytext[i] != '\0'; i++)
    if (yytext[i] == '\n')
      column = 0;
    else if (yytext[i] == '\t')
      column += 8 - (column % 8);
    else
      column++;

  ECHO;
}
```

1.5 The dot in the ending

The dot in the ending is to simply parse all the things that have not been seen yet...i.e. the "bad" characters

```
.     { /* ignore bad characters */ }
```

1.6 Alternative Operator representations

Sometimes some characters like a { which are crucial in C have alternative representations using different characters. For example:

```
("{"|"<%")     { count(); return('{'); }
```

Here both of the representations are parsed accordingly and the CODE returned back is the ASCII code of the open curly bracket !

2 Grammatical Specifications

2.1 If else ambiguity

Here is the C - grammar specification for if else statement:

```
selection_statement
  : IF '(' expression ')' statement
  | IF '(' expression ')' statement ELSE statement
  | SWITCH '(' expression ')' statement
```

Now the following statement can have 2 parse trees !

```
if(a) s; if (b) c; else d;
---"This can be parsed in 2 ways!"--
if(a){
    s;
    if(b);
    c;
}
else d;
-------"OR"--------
if(a){
    s;
    if(b){
    c;
    }
    else d;
}
```

The C parser solves this issue by pairing up the else statement with the closest if .i.e it uses the second type in the above example!

2.2 An Example

Let us study this using a simple example, I wrote a C file that has the following code

```
#include<stdio.h>
int main(){
long long int *volatile kartikpointer;  //declaration
long long int kartik;
kartik = 64209;
kartikpointer = &kartik;
printf("kartik points to  = %p ", kartikpointer);
return 0;}
```

2.2.1 Declaration

Rule:

```
declaration
  : declaration_specifiers ';'
  | declaration_specifiers init_declarator_list ';'----"used in line 3, 4"
```

The second is clearly the format of the lines 3 and 4 of the program, to dig deeper we shall try to see what the declaration specifiers and declaration list do

2.2.2 Declaration list

```
init_declarator_list
  : init_declarator
  | init_declarator_list ',' init_declarator

```

It is Interesting to note that the init - declarator list is defined as a left linear grammar(an extra comma has been added to seperate different identifiers)

2.2.3 Declaration Specifiers

```
declaration_specifiers
  : storage_class_specifier
  | storage_class_specifier declaration_specifiers
  | type_specifier
  | type_specifier declaration_specifiers
  | type_qualifier
  | type_qualifier declaration_specifiers


init_declarator_list
  : init_declarator
  | init_declarator_list ',' init_declarator
  ;

init_declarator
  : declarator
  | declarator '=' initializer
  and finally :-

declarator
  : pointer direct_declarator
  | direct_declarator
----------------------AND FINALLY --------------------
direct_declarator
  : IDENTIFIER
```

We use many of these rules! "Type specifier" has been used repeatedly -long long int has 3 type specifiers and the direct declarator finally takes us to either an Identifier or a pointer with an identifier ! which is precisely what happens in line 3 and 4 of our code !!!

2.3 Precedence and Assosciativity

The C grammar creates precedence of AND expressios over OR expressions! (because the OR expressions contain the AND expressions and its not the other way around... so the and expression is parsed under the OR expression tree... i.e. AND is closer to the leaf(terminal symbols))

2.3.1 Precedence

```
logical_and_expression
  | logical_and_expression AND_OP inclusive_or_expression
  ;

logical_or_expression
  : logical_and_expression
  | logical_or_expression OR_OP logical_and_expression
  ;
```

2.3.2 Associativity

```
logical_and_expression
  | logical_and_expression AND_OP inclusive_or_expression
  ;
```

The grammar is clearly left linear (i.e.) the logical and expression can repeatedly derive it self from the left side. i.e an expression of the form:-
$a\&\&b\&\&c \longrightarrow ((a\&\&b)\&\&c)$
(the inclusive or expression can yield terminal expressions of the form a , b, c)

2.4 Preprocessor Grammar

Often the preprocessor needs to analyze what Macros it has to expand and what files need to be imported. to do this the C preprocessor also has to parse the C program. Source : [RK88]

2.4.1 If's and Endif's

$$preprocessor - conditional \longrightarrow$$
$$if - line \ + text \ + elif - part_{opt} + \ \#endif$$

This shows how the #if - #endif statements are parsed in C Notice how there **may not be ambiguity** here because of the clearcut presence of an #endif statement indication when

the if statement is ending

$$if - line \longrightarrow$$
$$\#if\ constant - expression$$
$$\#ifdef\ identifier$$
$$\#ifndef\ identfier$$

2.5 Function Declarations

The declaration is simple,it contains declaration specifiers like we had covered in 2.1 and then it contains a compound statement(multiple statements for definition)

```
function_definition
  : declaration_specifiers declarator declaration_list compound_statement
```

3 Conclusion

This completes the summary of about 10 things I liked about the Lexical and Grammatical specifications of C.

References

[RK88] D.M. Ritchie and B.W. Kernighan. *The C programming language.* Bell Laboratories, 1988.