Compiler Project – TD1 : Lex & Yacc

{Ghislain.Charrier, Bogdan.Pasca} @ens-lyon.fr

February 5th, 2010

General view on the project

The compilation sequence of programs using Lex and Yacc is presented in Figure 1. As you can see from this figure the *lexical analysis* and *syntax analysis* are two of the three steps required to compile a program and are therefore crucial for the success of the project.

This lab session will serve as an introduction to Lex and Yacc. For more information please consult the **Further Reading** section and the first course `http://perso.ens-lyon.fr/daniel.hirschkoff/PCo/docs/elements-de-compil.pdf`.

The **Lex** tool receives at the input a set of user defined **patterns** that it uses to scan the *source code*. Each time the source code matches one of the patterns a defined action is executed by Lex (one of the action is that of returning the tokens).

The **Yacc** tool receives at the input the user grammar. Starting from this grammar it generates the C source code for the **parser**. Yacc invokes Lex to scan the *source code* and uses the tokens returned by Lex to build a **syntax tree**.

Code generation (not to be further discussed in this lab session) is done by parsing this tree (usually the tree passes through several intermediary representations).

Figure 2 shows the naming conventions used by lex and yacc. This figure will be further detailed in the following sections.

Lex

Lex is officially known as a "Lexical Analyzer". It's main job is to break up an input stream into more into meaningful units, or tokens. For example, consider breaking a text file up into individual words.

More pragmatically, Lex is a tool for automatically generating a **lexer** (also known as **scanner**) starting from a lex specification (*.l file 2).

The skeleton of a **lex specification file** is given in Figure 3.

The *rules section* is composed of tuples of the form <pattern, action>. As it can be seen from the following examples, are specified by regular expressions.

	Example
<pattern> <action to take when matched>	[A-Za-z]+ printf("this is a word") ;
<pattern> <action to take when matched>	[0-9]+ printf("this is a number") ;

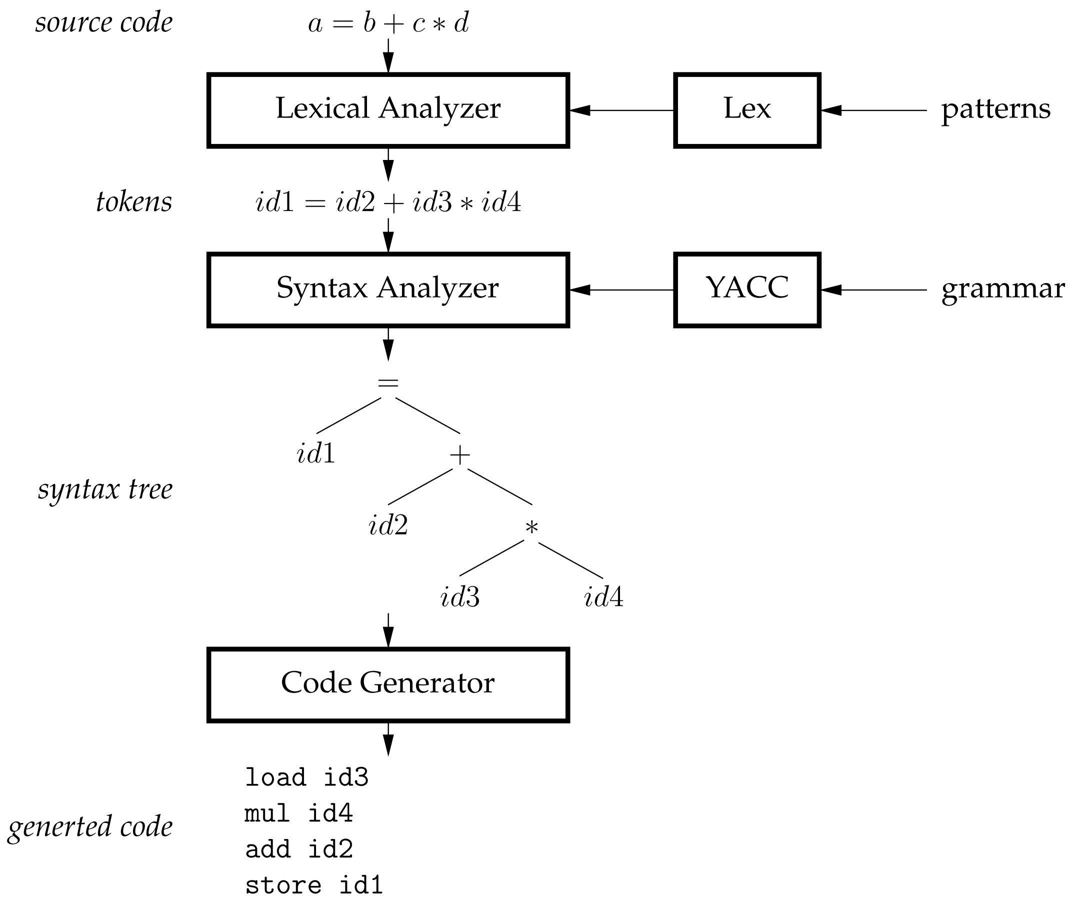

FIG. 1 – Compilation sequence (`epapers.com`)

Regular Expressions

In the following we denote by c=character, x,y=regular expressions, m,n=integers, i=identifier.

.	: matches any single character except newline
*	: matches 0 or more instances of the preceding regular expression
+	: matches 1 or more instances of the preceding regular expression
?	: matches 0 or 1 of the preceding regular expression
|	: matches the preceding or following regular expression
[ ]	: defines a character class
( )	: groups enclosed regular expression into a new regular expression
"..."	: matches everything within the " " literally
x | y	: x or y
{i}	: definition of i
x/y	: x, only if followed by y (y not removed from input)
x{m,n}	: m to n occurrences of x
ˆx	: x, but only at beginning of line
x$	: x, but only at end of line
"s"	: exactly what is in the quotes (except for "\" and following character)

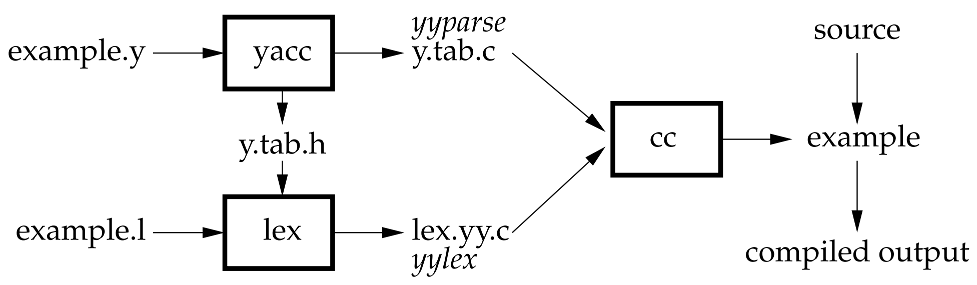

FIG. 2 – Building a compiler using lex and yacc (`epapers.com`)

scanner.l	*lex.yy.c is generated(Fig 2)*
%{ < C global variables, prototypes, comments > %}	*This part will be embedded into lex.yy.c*
DEFINITION SECTION	*substitutions, code and start states ; will be copied into *.c*
%%	
RULES SECTION	*define how to scan and what action to take for each token*
%%	
<auxiliary C subroutines>	*any user code. For example, a main function to call the scanning function yylex().*

FIG. 3 – Skeleton of a *.l specification file

Meta-characters do not match themselves, because they are used in the preceding regular expressions. To match a meta-character, prefix it with \. To match a backslash, tab or newline use \\, \t, \n. Some other meta-characters are :

```
( ) [ ] { } < > + / , ^ * | . \ " $ ? - %
```

Some simple examples of regular expressions :

```
an integer:                  [1-9][0-9]*
a word:                      [a-zA-Z]+
a (possibly) signed integer: [-+]?[1-9][0-9]*
a floating point number:     [0-9]*"."[0-9]+
```

In addition of the regular expressions presented above, lex uses its own **extended set** :

c	: any character except meta-characters (see below)
[...]	: the list of enclosed chars (may be a range)
[ˆ...]	: the list of chars not enclosed
.	: any ASCII char except newline
xy	: concatenation of x and y
x ?	: an optional x (same as x*)

How to compile and run the scanner ?

```
lex example.l
cc lex.yy.c -o example -ll
./example
```

Terminate the program with CTL^D.

Download and run the Lex example for detecting numbers and words found at `http://perso.ens-lyon.fr/bogdan.pasca/teaching_projcompil.php`.

Exercise 1 : Number ranges

Find a regular expression to validate numbers in the range : 0 to 000...127.

Exercise 2 : Floating-point number validator

Find good regular expression for floating-point numbers. It should validate numbers of the form 124 or .45 .

Exercise 3 : Email-address validator

Find good regular expression for validating email addresses.

Exercise 4 : Roman-numeral validator

Find a regular expression that validates roman numerals. Take a look at `http://www.yourdictionary.com/crossword/romanums.html` for details on roman numerals.

Working with Lex

The following example counts the lines of text file.

```
%{
int yylineno = 0;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

Exercise 5 : Token counter

Extend the previous example so to count the number of words and numbers of an input file.

Exercise 6 : Code tokenizer

Write the scanner that tokenizes the following code (taken from an Ubuntu of a *.bashrc* file).

```
# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
alias grep='grep --color=auto'
alias fgrep='fgrep --color=auto'
alias egrep='egrep --color=auto'
fi
```

YACC

YACC stands for **Y**et **A**nother **C**ompiler **C**ompiler. Its GNU version is called **Bison**. YACC translates any grammar of a language into a **parser** for that language. Grammars for YACC are described using a variant of Backus Naur Form (BNF). A BNF grammar can be used to express context-free languages. By convention, a YACC file has the suffix .y.

We use as a running example the grammar of a calculator that supports two operations : + and -. The code of this elementary calculator can be obtained from : `http://perso.ens-lyon.fr/bogdan.pasca/teaching_projcompil.php`

YACC file structure

```
                                                          ------------------------------
%{
    #include <stdio.h>                                    Part to be embedded into the *.c
    int yylex(void);
    void yyerror(char *);                                 ------------------------------
%}                                                        %Definition Section
%token INTEGER                                            (token declarations)
                                                          ------------------------------
%%                                                        Production rules section:
program:                                                  define how to "understand" the
        program expr '\n'     { printf("%d\n", $2);}      input language, and what actions
        |                                                 to take for each "sentence".
        ;
expr:
        INTEGER                 { $$ = $1; }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        ;                                                 ------------------------------
%%                                                        < C auxiliary subroutines >
void yyerror(char *s) {                                   Any user code. For example,
    fprintf(stderr, "%s\n", s);                           a main function to call the
}                                                         parser function yyparse()

int main(void) {
    yyparse();
    return 0;
}                                                         ------------------------------
```

Lex & YACC

Yacc generates a parser in file y.tab.c and an **include** file y.tab.h (see Fig 2). Lex includes this file (y.tab.h) and uses the definitions for token values found in this file for the returned tokens. Take as an example our INTEGER token from the calculator example. The corresponding part of the y.tab.h file looks like this :

```
#define INTEGER 258
```

Consequently, each time we have a **return INTEGER ;** statement in our *.l file, this will get replaced in our case by **return 258 ;**.

To obtain tokens yacc calls the function **yylex**. The function yylex has a return type **int** for returning tokens. In other words, the include file y.tab.h defines for each token of the *.y file its unique corresponding integer value. As this file is included in the *.l file, the token names are replaced by these unique identifiers (integers).

Token values 0-255 are reserved for character values. For example, if you had a rule such as :

```
[-+]                return *yytext;  /* return operator ID*/
```

the character value for minus or plus is returned so there is no need to redefine **PLUS, MINUS** tokens.

In addition, values associated with the token are returned by lex in variable yylval (this has nothing to do with the token int ID). For example, we find INTEGER tokens to which we associate the numerical value matched expression.

```
[0-9]+           {
                    yylval = atoi(yytext);
                    return INTEGER;
                }
```

The type of yylval is determined by YYSTYPE. The default type is integer and therefore works well for the previous example. If needed, this can be changed to other types.

We won't enter the grammar details now, as they where already presented at course.

How to compile and run the scanner + parser ?

```
yacc -d example.y
lex example.l
cc lex.yy.c y.tab.c -o example
./example
```

Terminate the program with CTL^D.

The **-d** option causes yacc to generate definitions for tokens and place them in the file y.tab.h. The **main()** function calls **yyparse()** which automatically calls **yylex()**

Exercise 7 : Fully Functional Calculator

Augment the running example with the * and / operations and parenthesis support. The parser should be able to process expressions of the form : 42/(7+2*3).

Exercise 8 : Floating-Point Calculator

Using the regular expressions found at Exercise 2 augment the previous example of the calculator. Think about the type of yylval.

Exercise 9 : Abstract-Syntax Tree

Build an then print the abstract-syntax tree for the grammar of the calculator in Exercise 7.

Further Readings

– `http://epaperpress.com/lexandyacc/`
– `http://203.208.166.84/dtanvirahmed/cse309N/LexYacc.ppt`
– `http://pltplp.net/lex-yacc/`
– `http://diveintopython.org/regular_expressions/roman_numerals.html`