COMPILER CONSTRUCTION (Parser Generator-YACC) **Fall 2019**

Lecture 21: Sept. 25, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

21.1 Parser Generators

A parser generator is used for construction of front end of a compiler. YACC is a LALR (Look ahead LR) parser generator. YACC provides a tool to produce a parser for a given grammar. YACC is a program designed to compile a LALR (1) grammar. It is used to produce the source code of the syntactic analyser of the language produced by LALR(1) grammar. YACC stands for "Yet another compiler compiler". YACC was created by S C. Johnson [5], a computer scientist at Bell Labs / AT&T, and it is available as a command in UNIX and LINUX (also Ubuntu) OS, and it is used for production of compilers.

21.1.1 YACC

The Fig. 21.1 shows the steps for creating a parser generator. At begin, a file, say `translate.y`, that contains a `yacc` specification of the translator, is prepared. The Linux command,

```
$ yacc translate.y
```

transforms the *translate.y* into a C program, called *y.tab.c* using the LALR method we discussed in LALR algorithm.

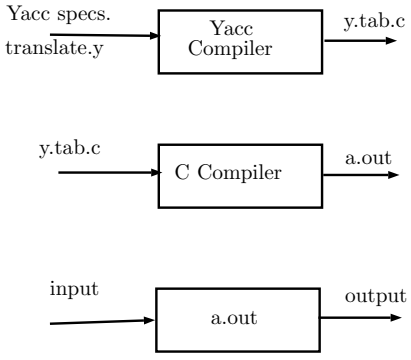

Figure 21.1: Creating an input/output translator with yacc

The program `y.tab.c` is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. Then LALR parsing table is compacted. We compile the *y.tab.c* along with "ly" library that contains the LR parsing program, using the command

```
$cc y.tab.c -ly
```

we obtain the desired object program *a.out* that performs the translation specified by the original Yacc program.

A Yacc source program has three parts:

```
declarations
%%
translation rules
%%
supporting C routines
```

Example 21.1 *Construct a simple desk calculator using YACC.*

We would use following grammar:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \textbf{digit}$$

where **digit** is a token of single-digit 0 to 9. The YACC program `deskcal.y` for this is shown below.

```
%{
#include <ctype.h>
%}

%token DIGIT
%%
line  : expr '\n'     { printf("%d\n", $1); }
      ;
expr  : expr '+' term  { $$ = $1 + $3; }
      | term
      ;
term  : term '*' factor { $$ = $1 * $3; }
      | factor
      ;
factor : '(' expr ')'   { $$ = $2; }
       | DIGIT
       ;
```

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

Follows the commands of YACC, and *gcc*, to get *a.out*, which can be run to perform simple two digit calculations using operators $*$ and $+$. Note that the input and result will be in single digit values. □

Declaration part There are two sections in the declaration part of a Yacc program, both are optional. In the first section, we put ordinary C declarations, delimited by %{ and %}. Here we put declarations of any temporaries used by the translation rules or procedures of the second and third sections. In our case, this section contains only the include-statement:

$$\#\texttt{include} < \texttt{ctype.h} >$$

that causes the C preprocessor to include the standard header file `ctype.h` that contains the predicate `isdigit`. Next, the statement %`token DIGIT` declares $DIGIT$ to be a token. The tokens declared in this section can then be used in the second and third parts of the Yacc specification. If Lex is used to create the lexical analyser that passes token to the Yacc parser, then these token declarations are also made available to the analyser generated by Lex.

Translation rules part The part of the YACC specification after first %% pair, we put translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing:

$$\langle head \rangle \rightarrow \langle body \rangle_1 \mid \langle body \rangle_2 \mid ... \mid \langle body \rangle_n$$

would be written as :

$$\begin{aligned} \langle head \rangle : & \langle body \rangle_1 \ \ \{\langle semantic\ action \rangle_1\} \\ & \mid \langle body \rangle_2 \ \ \{\langle semantic\ action \rangle_2\} \\ & ... \\ & \mid \langle body \rangle_n \ \ \{\langle semantic\ action \rangle_n\} \\ & ; \end{aligned} \tag{21.1}$$

In a YACC production, unquoted strings of letters and digits that are not delcared to be tokens are taken to be non-terminals. A quoted single character, e.g., 'c', is taken to be a

terminal symbol c, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for 'c' to the parser, as integer). Alternative bodies can be separated by vertical bar, and a semicolon follows each head with its alternatives and their semantic actions, as shown in equation 21.1. The first head is taken to be the start symbol.

The Yacc semantic actions are sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the non-terminal of the head, while $\$_i$ refers to the value associated with the ith grammar symbol (terminal or non-terminal) of the body. The semantic action is performed whenever we reduce the associated production, so normally the semantic action computes a value for $$ in terms of $\$_i$'s. In the YACC specification we have written two statements:

$$E \rightarrow E + T \mid T$$

and their associated semantic actions are:

```
expr : expr '+' term  { $$ = $1 + $3; }
     | term
     ;
```

The special identifiers $$, $1 and $3 refer to items on the parser's stack. Note that non-terminal term in the first production is the third grammar symbol of the body, while + is the second. The semantic action associated with the first production adds the value of the expr and the term of the body and assigns the result as the value for the non-terminal expr of the head. We have omitted the semantic action for the second production with single grammar symbol in the body. In general, $$ = $1; is the default semantic action.

Note that we have added a new starting production,

```
line : expr '\n'  { printf("%d\n", $1); }
```

to the Yacc specification. This production says that an input to the desk calculator is to be an expression followed by a new line character. The semantic action associated with this production prints the decimal value of the expression followed by a new line character.

Supporting C-routine part The third part of a Yacc specification consists of supporting C-routines. A lexical analyser by the name `yylex`() is to be provided. The lexical analyser yylex() produces tokens consisting of a token name and its associated attribute value. If a token name such as DIGIT is returned, the token name must be declared in the first section of the Yacc specification. The attribute value associated with a token is communicated to the parser through Yacc-defined variable *yylval*.

In our lexical analyser, if the input character is digit, the value of digit is stored in the variable *yyval*, and the token DIGIT is returned. Otherwise the character itself is returned as the token name.

21.2 Review Questions

1. YACC is an acronym for:
   A) Yes Another Compile Compiler
   B) Yet Another Compile Compiler
   C) Yet Another Compiler Compiler
   D) Yes Another Compiler Compiler

2. The YACC takes C code as input and outputs
   A) Top down parsers
   B) Bottom up parsers
   C) Machine code
   D) None of the mentioned

3. Yacc semantic action is a sequence of
   A) Tokens
   B) Expression
   C) Statement
   D) Rules

4. Input of Lex is
   A) Set to regular expression
   B) Statement
   C) Numeric data
   D) ASCII data

21.3 Exercises

1. In the example 21.1, what are the semantic actions? Explain.

2. Extend the example 21.1 to calculate subtraction and division.

3. For the example 21.1, when it is run, input the following expressions and justify the result produced by a.out: $2 + 3$, $2 * 3$, $2 + (2 + 2)$, $2 * (2 + 3)$, $3 * 4$.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.

[5] Johnson, Stephen C. (1975). *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories Technical Reports. AT&T Bell Laboratories Murray Hill, New Jersey 07974 (32)