Yacc: A Syntactic Analysers Generator

Compiler-Construction Tools

The compiler writer uses specialised tools (in addition to those normally used for software development) that produce components that can easily be integrated in the compiler and help implement various phases of a compiler.

- Scanner generators
- Parser generators
- Syntax-directed translation
- Code-generator generators
- Data-flow analysis: key part of code optimisation

Relationship between Parser and Scanner

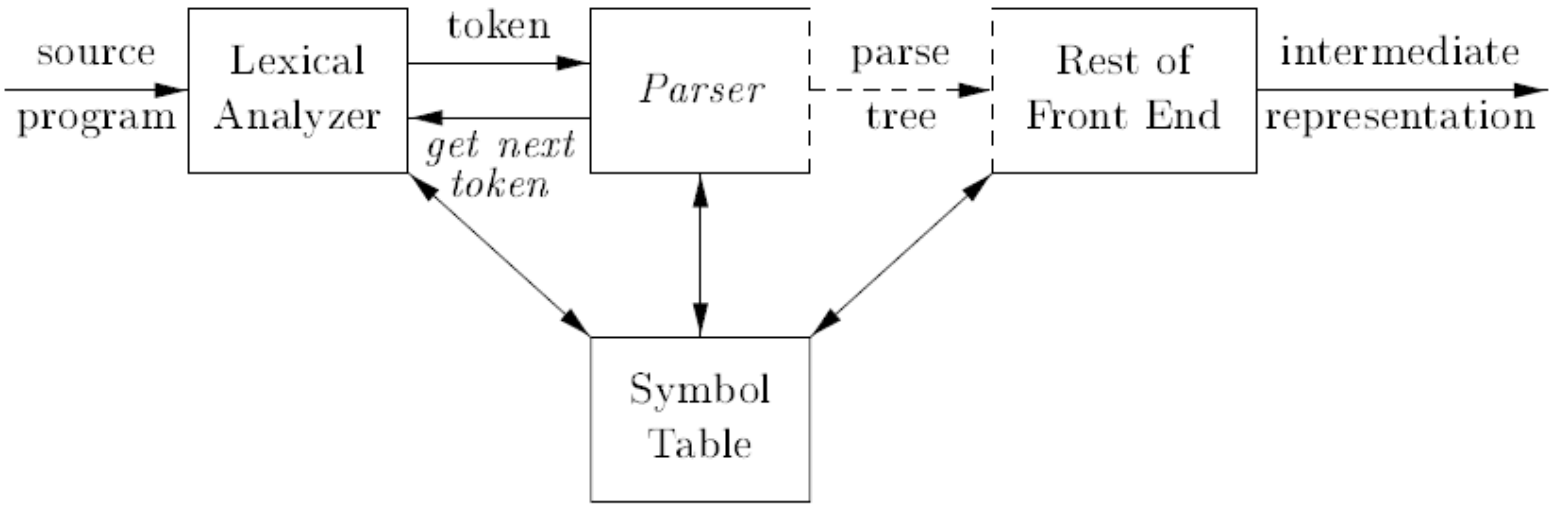

Figure 4.1: Position of parser in compiler model

Parser guides scanner by asking tokens one by one. Interaction with the symbol table is crucial.

Lex and Yacc

There exist various tools for generating parsers.
We will discuss **Yacc**, the companion of Lex: the programs they generate can share variables and procedures and therefore can be compiled jointly.

Lex & YACC are known to GNU/Linux users as Flex & Bison, where Flex is a Lex implementation by Vern Paxson and Bison the GNU version of YACC.

Typically, the Lex actions are not visible on the standard output; rather Lex return values and control to the caller (i.e. the parser). Function `yylex()` returns the token names, while the token values are shared by means of global variables such as `yylval` of type `int`.

Yacc

LALR(1) parser generator whose first version is due to S.C. Johnson (AT&T Bell Laboratories) in the early 1970s.

To construct a translator, Yacc operates as follows:

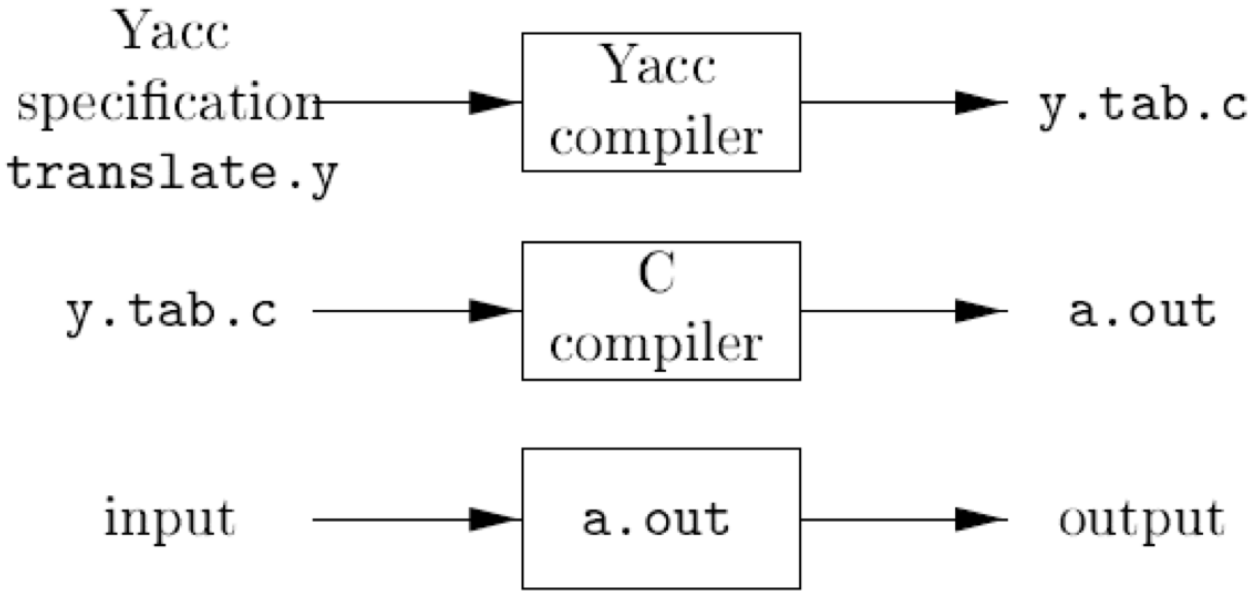

Figure 4.57: Creating an input/output translator with `Yacc`

Users only need to care of the Yacc source (file .y where the grammar is defined); everything else is implemented automatically.

Yacc Basics

- The parser generated by yacc is a C function called
- `yyparse( )` is an LALR(1) parser
- `yyparse( )` calls `yylex( )` repeatedly to obtain the next input token
- The function `yylex( )` can be hand-coded in C or generated by lex
- `yyparse( )` returns an integer value
  - 0 is returned if parsing succeeds and end of file is reached
  - 1 is returned if parsing fails due to a syntax error

Yacc Specifications

Structure of a Yacc program:

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

A production of the form

$$nonterm \rightarrow corpo_1 \mid \cdots \mid corpo_k$$

is expressed in Yacc by the rules:

```
nonterm : body_1 {semantic action_1}
          ...
        | body_k {semantic action_k}
        ;
```

Declarations

The declaration part of a Yacc program has two (both optional) sections:

- ordinary C declarations delimited by %{ and %} (e.g. standard header files)
- declarations of grammar tokens of the form

  ```
  %token DIGIT
  ```

  These declarations are made available to the analyzer generated by Lex when `lex.yy.c` is compiled together with the Yacc output.

Supporting C-Routines

The third part of a Yacc specification contains procedures in C:

- The lexical analyzer is provided as `yylex()`. Usually this is produced by Lex.
- Other procedures such as error recovery routines.

If the pair (`token-name`, `attribute-value`) is produced by `yylex()`, then `token-name` must be declared in the first section of the Yacc specification, while `attribute-value` is communicated to the parser through the variable `yylval` (defined by Yacc).

Attribute Values and Semantic Actions

- Every grammar symbol has an associated attribute value
- An attribute value can represent anything we choose
  - The value of an expression
  - The data type of an expression
  - The translated code
- Yacc associates an attribute with every token and non-terminal
  - Token attributes are returned by the scanner in the yylval variable
  - Non-terminal attributes are computed while parsing
- Attribute values are pushed an popped on a semantic stack
  - The semantic stack operates in parallel with the parser stack
- A semantic action in Yacc is a code fragment delimited by { }
  - Executed when yacc matches a rule in the grammar
  - Semantic Actions can be used to make calls to semantic routines

Example

Construct a simple desk calculator that reads an aritmetic expression, evaluates it and then prints its numeric value.

Start with the following grammar for arithmetic expressions:

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T * F | F \\ F &\rightarrow (E) | \textbf{digit} \end{aligned}$$

The token **digit** is a single digit between 0 and 9.

Example ctd.

```
%{
#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;
}
```

Creating Yacc analyzers

The Lex library `ll` provides the driver program `yylex()` to Yacc. When using Lex and Yacc together, we can replace the routine `yylex()` in the third part of the Yacc specification by the statement

```
#include "lex.yy.c"
```

Since the Lex output file is compiled as part of the Yacc output file `y.tab.c`, the lexical analyzer can then have access to Yacc's names for tokens.