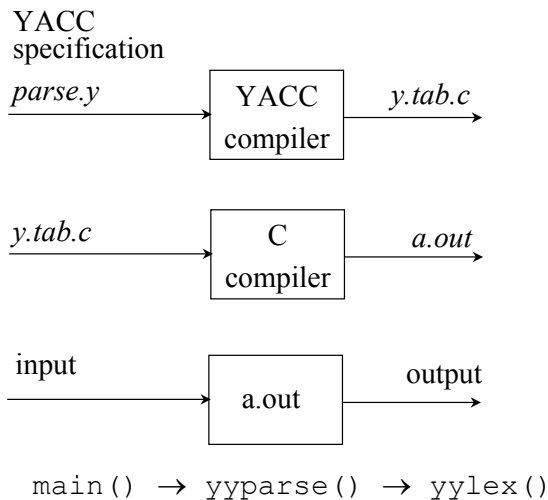


PARSER GENERATOR YACC:

(more details - see YACC manual. e.g. in Web)

- + available under UNIX
- + input : grammar and semantic actions
- + output: LALR(1) parse tables and shift-reduce parser
- + lexical analyzer must be provided separately (typically *LEX*)

Function diagram:



General form of the YACC source program:

```
declarations
%%
translation rules
%%
support routines
```

The declarations part has two optional sections:

- + ordinary C declarations, delimited by `%{` and `%}`

```
%{
#include <ctype.h>
#include "lex.yy.c"
typedef struct {
. . . .
}VALUE
%}
```

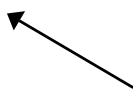
+YACC's internal declarations

```
%token VAR (equivalent %term VAR)
%token ASSIGN
%token VARIABLE
%start
```

are equivalent: `%token VAR ASSIGN VARIABLE`
(by convention, upper case letters are used for tokens)

More examples on declarations:

```
%left '+' '-'
%right UMINUS '^'
%nonassoc '<'
```



Disambiguating rules solve possible conflicts

The transition rules part

contains grammar production and associated semantic action in the form:

non terminal : *right hand side (with | for alternatives)*
{semantics actions};

- + RHSs may include terminals (quoted single characters. e.g. 's', TOKENS) or nonterminals, defined elsewhere
- + note separators | and delimiters ;
- + actions (C statements) may include variables of the form `$n` or `$$`

```
/ S-attributed grammars with a single attribute
```

```
/ previous attribute value: $0 (useful for rules with inherited attributes)
```

```
/ default action: $$ = $1
```

Example:

production `T → T * F | F` in the YACC specification:

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

The supporting C-routines part

- + lexical analyzer by the name `yylex()`
- + error recovery routines
- + complete function `main()` can be realized here

Ambiguous grammars in YACC and methods of conflicts solutions:

- + for ambiguous grammars, LALR algorithm in YACC generates parsing action conflicts
- + readable description of the characteristic automaton (parse table) and additional description is generated into the file `y.output` when `-v` option is used in the YACC command line

Constructor will ***implicitly*** resolve all parsing conflicts using following two rules:

reduce-reduce: choosing the conflicting production

listed first in the YACC specification

shift-reduce: favor on shift

```
/*
 * This Yacc spec illustrates the error
 *
 *      n rules never reduced
 *
 * Here, the rule "C : 'c'" is never reduced
 * because C never appears on the RHS
 * of any other rule.
 *
 * This is an example of y.output file
 */
%%
X : A B
A : 'a'
B : 'b'
C : 'c'
```

Example of `y.output` file:

```
state 0
    $accept : _X $end

    a shift 3
    . error

    X goto 1
    A goto 2

state 1
    $accept : X_$end

    $end accept
    . error
```

```

state 2
  X :  A_B

      b shift 5
      . error
      B goto 4

state 3
  A :  a_      (2)

      . reduce 2

state 4
  X :  A B_    (1)

      . reduce 1

state 5
  B :  b_      (3)

      . reduce 3

```

Rule not reduced: C : c

```

5/127 terminals, 4/600 nonterminals
5/300 grammar rules, 6/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
4/601 working sets used
memory: states,etc. 17/2000, parser 2/4000
5/601 distinct lookahead sets
0 extra closures
2 shift entries, 1 exceptions .....

```

General mechanism for resolving shift/reduce conflicts :

- + precedence and associativity is attached to each **production** and **terminal** involved in a conflict
- + precedence and associativity of *terminals* can be assigned in the declaration part
 - / Tokens (terminals) are given precedences in the order in which they appear in the declaration part
 - / Tokens (terminals) with the lowest precedence are first
 - / precedence in the same declaration (line) is identical
- + precedence of the production is the same as its rightmost terminal has

YACC reduces in case of a conflict:

- + if the precedence of the production is greater then precedence of the input symbol
- or
- + if the precedences are the same and the associativity of the production is **left**

Example:

```

%left '+' '-'
%left '*' '/'
/* associativity and precedence is fully given */
%%
E : E '+' E
  | E '*' E
  | E

E + E + Lookahead +:   reduces
E + E + Lookahead *:   shifts

```

When the rightmost terminal does not supply the correct precedence to a production, we can **redefine the precedence** of production:

LHS : RHS %prec <terminal>

Note: conflicts, resolved using this precedence and associativity mechanism YACC doesn't report

Type consistency in YACC

- + all attributes used in YACC must be the same type
- + default attribute values: integer
- + **alternative solution:** use the union to encapsulate all types of attributes. Then the global variable `yyval` is of the same type (YYSTYPE)

Example:

```
%union {
    char *s;    /* identifier */
    long i;    /* integer literal */
    int l;     /* label */
}
```

and for any token or nonterminal we can specify of the associated attribute as one of the fields in this union.

```
%token <s> ID or %term <s> ID /* terminals */
%type <l> ifprefix whileprefix /* nonterminals*/
```

Error handling

standard method: after any error appearance, compiler (executable program) reports "Syntax error" and terminates

user defined method: error recovery mechanisms can be implemented - compiler reports the error, refresh and continue (e.g. on the next line) :

- + select "major" nonterminals (e.g. A), with which the error recovery is associated and add the rule

$A \rightarrow \text{error } \alpha$

into the YACC specification (error is keyword).

- + On encountering an error, YACCC pops the symbols from the stack until finds a state, that has a shift action on the token `error`.
- + parser shifts the token `error` on the top of the stack and proceeds to skip ahead in the input until it has found sentential form α (if exists) and transfers it onto the stack
- + top of the stack is reduced to the corresponding ("major") nonterminal and user-defined error recovery routine is invoked. After its terminating, `yyerror` procedure is called to restore the normal course of syntax analysis.

Example:

```
lines :lines expr '\n'
      |lines '\n'
      /* epsilon */
      |error '\n' {yyerror("new input");
                  yyerrork;}
      ;
```