Parser Generators: YACC

- A parser generator takes a grammar as input, and produces a parsing table for it.

  For CFGs, `yacc` produces a LALR table (lookahead LR) and stack actions to model the actions of a PDA for parsing a given CFG.

  In addition to yes/no decisions, the parser can act as a translator of the language of input CFG to another language, via actions associated with each rule.

yacc assumes that the lexical analysis is done by some module. Default is lex (yylex() function)).

```
%{
#include "lex.yy.c"
%}

..

yyparse() calls yylex()
```

YACC has access to token types and values via yytext, yylval.

```
token types                 %token
start symbol                %start
C-includes                  %{.. %}
grammar variable types      %type
                            %union
```

- YACC uses synthesized attributes.

  rule format:

  ```
  non-terminal : RHS {actions } ;
  ```

  defaults: first rule shows the start symbol unless there is %start.

Each non-terminal has a *single* attribute; to carry more information, use %union.

rule	$X \rightarrow$	Y_1	Y_2	$\ldots Y_n$
YACC attr	$$	$1	$2	$n

ex:

```
a : a + t { $$ = add($1,$3)} ;

f : ID    { $$ = get_val($1)};
```

If no action is specified, default is `$$ = $1`. But don't rely on defaults; this may be misleading during development.

empty RHS : `x : ;`

- If you run yacc with `-v` option, it produces a file that shows the configuration sets, which is the LR(1) machine (LR(1) items, action/goto table, shift/reduce, red/red conflicts).

- Parsing ambiguous or non-LALR grammars with yacc: Even if the grammar is not LALR, yacc will give you an action/goto table. Either fix the grammar, or rely on defaults to resolve conflicts, or override the defaults to get the desired behaviour.

- action defaults: in a shift/red conflict, shift. In a reduce/reduce conflict, reduce by the first rule in the rule order.

  Yacc's remedies for these are very *ad hoc*. Also, relying on defaults (or

overriding them) makes the parsing decisions implicit in yacc but not explicit in the grammar. Not a transparent way to write a compiler.

- Overriding the default in shift/red conflicts:

  define associativity and precedence of operators

```
%left op
%right op
%noassoc op
```

ex: $E \rightarrow E + E \mid E * E \mid ..(E)$

```
%left '+'
%left '*'
```

```
          stack          input
case 1:   $E+E           *E...     shifts
case 2:   $E+E           +E...     reduces
case 3:   $E*E           +E...     reduces
```

what about $E \longrightarrow E - E \mid E * E \mid -E$

but you can't have

```
%left  '-'
%right '-'
```

- ERROR HANDLING in YACC: the strategy is to get rid of the stuff on the stack until something viable is found.

pop stack until a variable with defined goto is found

skip until the FOLLOW of that variable

synchronize

- Error productions:

  `non-terminal : error synchronizing-set ;`

  pop stack until a state with $X \Rightarrow error \cdot$ is found

  shift $error$ onto stack

  discard until synch-set member is found in input

  continue (issue `yyerrorok;`)

- How many errors due to error productions? Not infinitely many. Yacc won't use the error production again until 3 sucessfull shifts are done

  LALR grammars may do some reduction before they hit a dead-end, but they don't shift before a dead-end.

- Debugging yacc programs:

```
%{
#define YYDEBUG
%}
...

yydebug=1; %before calling the yyparse()
```