Yet Another Compiler-Compiler

Automatic generation of CF parsers

YACC – Yet Another Compiler-Compiler

- YACC (Bison) is a parser generator for LALR(1) grammars
  - Given a description of the grammar generates a C source for the parser

- The input is a file that contains the grammar description with a formalism similar to the BNF (Backus-Naur Form) notation for language specification
  - non terminal symbols – lowercase identifiers
    - expr, stmt
  - terminal symbols– uppercase identifiers or single characters
    - INTEGER, FLOAT, IF, WHILE, ‘;’, ‘.’
  - Grammar rules (production rules)
    - expr: expr ‘+’ expr | expr ‘*’ expr ; $E \rightarrow E+E|E*E$

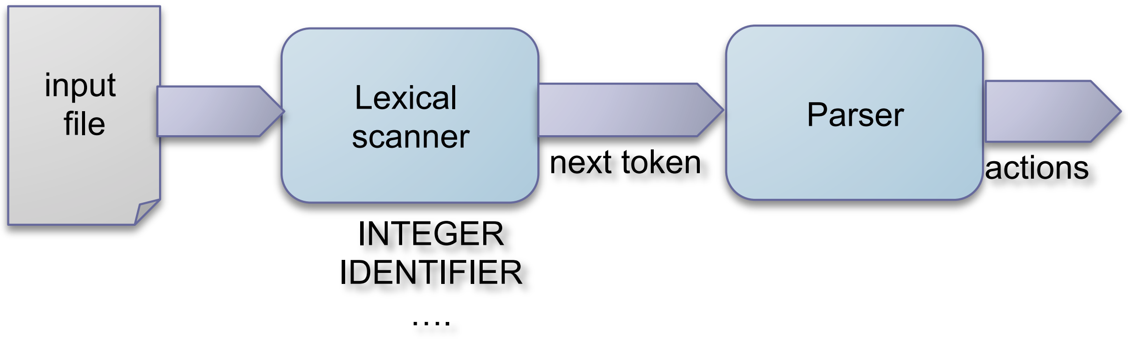
YACC – values and actions

- A data type and a semantic values is associated to each element
  - Data type: INTEGER, IDENTIFIER
  - Value: the numeric value of the integer, the reference of the identifier in the symbol table
- A value can be associated also to syntactic categories
  - The result of the evaluation of an arithmetic expression
- Actions correspond to C code associated to each production rule
  - When a production rule is used for a reduction the associated action is executed
  - In general the action is used to combine the values associated to the elements in the right side of the production rule to obtain the values associated to the non terminal symbol in the left side
    - expr: expr '+' expr { $$ = $1 + $3;};

```
        1    2    3
```

YACC – integration with the lexical scanner

- A lexical scanner is exploited to detect the terminal symbols in the parsed file
  - The parser generated by YACC makes a call to the lexical scanner when it needs to read the next terminal symbol for the input



- The parse is implemented by the function yyparse() that needs the support of the lexical scanner (e.g. yylex()), the error handling functions and the caller procedure

Grammar file

- The grammar is defined in a text file (usually with extension .y)

```
%{
C declarations (include/define/global variables)
%}
YACC definitions (terminal/non-terminal symbols and their properties)
%%
grammar ruels and associated actions
%%
C code (it is copied into the generated file after the yyparse() function)
```

Terminal symbols

- They are a class of equivalent elements from the syntax point of view
  - they are represented by numerical codes associated to their identifier
    - In C language they correspond to a set of #define statements
    - The lexical scanner yylex() must return the code corresponding to the class of the element matched in the input text (the lex option –d is used to generate the file xx.tab.h that contains the defines for the terminal symbol classes)

```
while   return WHILE;  /* lex rule */
```

    - The terminal symbols are declared in the YACC declaration section

```
%token WHILE
```

    - The literal tokens are used as the corresponding character constants in C (f.i. ‘+’) and there is not required to declare them explicitly unless there is the need to specify the associated data type, their precedence or associative property (the associated code is the corresponding ASCII ancoding)

Rule definition- 1

- A grammatical rule has the following structure

result: components….. ;

  - <result> is the non terminal symbol to which the right side of the production rule is reduced
  - The right side of the production rule is a sequence of components that consist of terminal symbols, non terminal symbols and actions (C code between {…})

$$\text{expr:} \qquad \text{expr } \overbrace{\text{'+'}}^{\text{terminal symbol}} \text{ expr } \underbrace{\{\$\$=\$1+\$3;\ \}}_{\text{ACTION}}$$

Rule definition- 2

- Alternative reductions for the same non terminal symbol can be listed

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

- If the right side of the production rule is empty, the rule is satisfied by the empty string

```
expr:     /* empty */ |
          expr1 ;
```

- The rule is recursive if the terminal symbol in the left side (<result>) appears also in the right side (it is better to avoid right recursion…)

```
exprseq:  expr |
          exprseq ',' expr ;
```

Semantics definition- 1

- The semantics depends on the values associated to the grammar tokens and on the actions that are caused by a reduction (when a production rule is selected)

$$\begin{tikzcd} \text{expr} & \text{`+'} & \text{expr} \\ & \text{expr} \arrow[ul] \arrow[u] \arrow[ur] & \end{tikzcd}$$

The execution of the rule

expr: expr '+' expr;

must assign to the symbol expr, resulting from the reduction, the value of the sum of the values associated to the two expr symbols in the right side

  - By default the int data type is associated to any symbol
  - Another C data type can be associated to all the symbols by the declaration in the C section

```
#define YYSTYPE <tipoC>
```

Semantics definition- 2

- If different data types are to be used for different symbols
  - The completed list of data tyeps must be specified in the YACC declarations

C types

```
%union {
double  val;
char   *sptr;
}
```

Names associated to types in YACC

  - One of the declared data types is associated to a terminal/non terminal symbol with a the declarations %token and %type

```
%token <val>   NUM

%type   <sptr>  string_ass
```

Semantics definition- 3

- The action is a C code block that is executed when a production rule is applied (reduction)
  - Actions can also appear between the symbols in the string of the right side of the production rule and, in this case, they are executed when the rule is partially matched (this makes rules less clear to understand)
  - The action C code can refer the semantic values associated to the rule tokens

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

  - The value of the n-th token is associated to the identifier $n
  - $$ represents the left side value
  - If no action is specified then $$=$1 by default
  - The data type of $n is that declared for the corresponding token (it can be eventually casted with $<type>n)

YACC declarations

- All the terminal symbols that do not correspond to single characters should be declared
  - The directive %token ID is used (if generates a #define)
  - It is possible to specify the precedence and the type of associative property (left/right) for the operators to simplify the grammar

```
%left symbols           }
%left <type>symbols     }  x op y op z  :  (x op y) op z
%right symbols          }
%right <type>symbols    }  x op y op z  :  x op (y op z)
%nonassoc symbols          x op y op z  :  syntax error
```

  - The operator precedence depends on the order of declarations

```
%left '+' '-'     | -
                  |    priority
%left '*' '/'     ↓ +
```

The interface with C

- The parse function yyparse() reads terminal symbols, executes the actions and returns when
  - The end of file is reached (return value 0)
  - A fatal syntax error is found (return value 1)
  - The macro YYACCEPT (return value 0) or YYABORT (return value 1) are called in an action
- The terminal symbols are detected by the lexical scanner – f.i. yylex() - that returns the corresponding code (ASCII code or YACC #define value)
  - The eventual semantic value associated to the terminal symbol must be stored into the global variable yylval
  - If a single data type YYSTYPE is used in YACC, then yylval is of type YYSTYPE, otherwise it is a C union data structure

The interface with C - yylval

- If yylval is of a single data type, in lex its value will be assigned as

```
 ….
 yylval = value;
 return ID;
}
```

- If more data type are used

```
%union {
  double   val;
  char     *sptr;
}
```

```
 ….
 yylval.sptr = string;
 return STRING;
}
```

  - If the assigned value is a pointer, the address should be in the global memory space or in the heap (dynamic allocation with malloc)

Error handling- 1

- A syntax error is issued when the parser reads a terminal input symbol that does not match any production rule
- The error condition can also be forced by a call to the macro YYERROR during the execution of an action
- The parser reports the error by calling the function

  `yyerror(char *s)`

  - This function must be implemented
  - By default the error message is "*parse error*" but the directive #define YYERROR_VERBOSE is specified the error string is more detailed
  - Another error, that can be reported sometimes, is "*parser stack overflow*" (too deep nesting of rules – f.i. right recursion). The stack is dynamically resized but an error is generated when its size reaches a maximum allowed dimension

Error handling - 2

- After the call to yyerror() the parser tries to recover from the error condition if a error recover function is implemented, otherwise it exists yyparse returning 1

- The variable yynerrs stores the number of encountered errors (it is a global variable for non-reentrant parsers)

- In general it is preferable to avoid halting the parser at the first error
  - The error handling policy can be defined by exploiting the special terminal symbol error in the production rule
  - The special symbol error is generated by the parser every time that a syntax error is found

Error handling- 3

```
stmt:  /* empty */ |
        stmt '\n' |
        stmt exp '\n' |
        stmt error '\n'
```

- If there is an error in exp
  - Some incomplete derivations in the parser stack and terminal symbols in the input are likely to be found before an input '\n' is matched
  - The parser forces the application of the rule removing part of the syntactical context from the stack and the input
    - it removes states and objects from the stack until the rule containing error is matched (it finds the previous stmt)
    - it pushes the symbol error into the stack
    - it reads input symbols until it finds a matching lookahead terminal symbol ('\n' in this case)

Error handling – the art of…

- In general it is difficult to decide (and implement) an optimal strategy for error handling
  - For instance the remaining part of the input line or the current command can be skipped in case of an error
  -

```
stmt: error ‘;’ /* in presence of an error move
                   to the next “;” */
```

  - We can try to balance the parentheses to avoid chained errors correlated to the first one

```
item: ‘(‘ expr ‘)’
      | ‘(‘ error ‘)’ /* it detects and error in expr
                         without generating errors for
                         parenthesis balancing */
```

Error handling– the art of… 2

- If the wrong error policy si used, a syntax error can be the cause of another one…
- To avoid an uncontrolled generation of error messages the parser does not report new error messages for a syntax error that is found just after the last one (at least two new symbols are to be read to generate a new error)
- The reporting of error messages can be reactivated by the call of the function yyerrok in the action

An example- calculator

- Parser to implement the operations of a multifunction calculator that has the following features
  - Arithmetic operators ('+', '-', '*', '/','^')
  - Predefined functions (sin, cos, exp, log,…) to be invoked as f(x)
  - Variables with variable names and assignments (v=1)

- Source file for YACC/Bison
- Source file for LEX
- Utility file in C (symbol table management)

- The parser C source is generated by the command `bison –d calc.y`
  - The genearted files are `calc.yy,tab.c` and `calc.yy.tab.h`

Parser generators

- Several generators for lexical scanners/syntactical parsers are available
  - They generate source code for different target languages
  - They implement different parsing strategies

  - BYACC/J generates LALR(1) parsers in Java
  - Coco/R generates LL(k) parsers in C, C++, C#, Java, Pascal...
  - CUP generates LALR(1) parsers in Java
  - JavaCC generates LL(k) parsers in Java
  - Lime generates LALR(1) parsers in PHP
  - .....

  - See http://en.wikipedia.org/wiki/Comparison_of_parser_generators