Compilers tutorial III: Syntactic analysis

Yacc is a powerful tool for automating the creation of parsers, also known as syntactic analysers, mainly used in language processing and compiler construction. It transforms formal grammars into executable code, making it invaluable for language analysis and processing. It primarily supports context-free grammars.

A bit of theory: *Yacc* takes the user-specified grammar rules and algorithmically constructs the corresponding LR parsing table. Specifically, it constructs an LALR(1) parser, that is, a Look Ahead Left-to-right Rightmost derivation parser. The parser takes as input the sequence of tokens passed by the lexical analyser (*lex*) and moves according to the parsing table. If it completes the derivation of the input sequence, reaching an accepting state, then the input is in the language of the grammar. Otherwise, a syntax error is found.

Using *lex* and *yacc* together

Lex and *yacc* were designed to work together: *lex* produces tokens that can be given as input into *yacc*. A *yacc* source file has three sections separated with the `%%` delimiter:

```
...definitions...
%%
...rules...
%%
...subroutines...
```

The *definitions* section contains C code delimited by `%{ ... %}` and token declarations. The *rules* section contains the grammar, in a notation similar to BNF (Backus-Naur form). The *subroutines* section contains any C functions needed. Integrating *lex* and *yacc* is achieved by placing `%token` declarations in the definitions section of the *yacc* source file:

```
%token NATURAL
```

This line declares a `NATURAL` token. The parser generated by *yacc* is written to a file called `y.tab.c`, along with an include file called `y.tab.h`. The *lex* specification must include `y.tab.h` so it can *return* tokens to the parser:

```
{digit}+                { yylval = atoi(yytext); return NATURAL; }
```

To obtain tokens, *yacc* calls the `yylex()` function, which returns an integer representing the identified token. The token *value* is passed to *yacc* in variable `yylval`. The type of `yylval` is `int` by default. For single-character tokens there is a convenient shortcut:

```
[()=,*/+-]              { return yytext[0]; }
```

With this rule, *lex* returns the character value for any of those single-character tokens. It would be functionally equivalent to declare a `%token` for each of them.

A simple calculator example

To use *yacc*, we write a grammar with fragments of C code enclosed within curly braces, called actions, attached to the grammar rules. Those code fragments are executed whenever a rule is used during the parsing of a sequence of tokens.

A grammar rule has a single nonterminal on the left-hand side of a production, followed by `:` and followed by the right-hand side of the rule, possibly followed by alternative rules using `|` as a separator. To illustrate this with an example, consider a small calculator capable of evaluating expressions such as `2+3*4` and printing the result. The rules section consists of the following grammar:

```
calculator: expression                  { printf("%d\n", $1); }
          ;

expression: NATURAL                     { $$ = $1; }
          | expression '+' expression   { $$ = $1 + $3; }
          | expression '-' expression   { $$ = $1 - $3; }
          | expression '*' expression   { $$ = $1 * $3; }
          | expression '/' expression   { $$ = $1 / $3; }
          ;
```

The parser maintains two stacks: a *parse stack* and a *value stack*. The parse stack represents the current state of the parser, consisting of terminals and nonterminals. The value stack associates a value with each element of the parse stack. For instance, when the parser *shifts* a `NATURAL` token to the parse stack, the corresponding `yylval` is pushed to the value stack.

When the parser *reduces* its stack, by popping the right-hand side of a production and pushing back the left-hand side nonterminal, the corresponding action is executed. For example, when the rule `expression: expression '+' expression` is applied, it will pop the three elements `expression '+' expression` and push back `expression`. The action `{ $$ = $1 + $3; }` will be executed.

Notice that the C code of actions can reference positions in the value stack, by using `$1, $2, $3, ..., $n` to reference the values of the right-hand side of the production, that is, the `n` values popped from the stack. Moreover, `$$` references the new value to be placed at the top of the stack. Therefore, the action `{ $$ = $1 + $3; }` adds the values of two expressions and pushes back the resulting sum. This way, the two stacks remain synchronized.

Generating and running the parser

Having the *yacc* specification in a file named `calc.y`, we obtain the C code for the parser by entering:

```
$ yacc -dv calc.y
```

Having the lexical specification in a file named `calc.l`, we supply a lexical analyser to read the input and pass the sequence of tokens to the parser:

```
$ lex calc.l
```

The source code generated by *yacc* is written to `y.tab.c` and `y.tab.h`, while the source code generated by *lex* is written to `lex.yy.c`. Compile and run the parser using:

```
$ cc y.tab.c lex.yy.c -o calc
$ ./calc
2+3*4               [input]
14                  [output]
```

There are some important observations about the *lex* and *yacc*:

- The parser is executed by calling `yyparse()`, which internally calls `yylex()` to obtain the tokens from the lexical analyser.
- The lexical analyser must `#include "y.tab.h"` to access the `%token` declarations specified in the syntactic analyser, so it can *return* them.
- The `yylval` variable is shared between *lex* and *yacc* to allow for token *values* to be passed from the lexer to the parser.
- When the parser finds a syntax error, it calls the `yyerror(char *)` function which we should supply.

Grammar ambiguity

The above grammar is ambiguous. For example, it allows for `3-2-1` to be parsed both as `(3-2)-1` and `3-(2-1)`. An ambiguous grammar can't be LR(k) so, more specifically, it can't be LALR(1). As a result, *yacc* warns of 16 shift/reduce conflicts for the above grammar. It will still generate a parser which uses *shift* as the default operation.

The command `yacc -dv calc.y` includes the `v` option, so it produces an extra text file with extension `.output` that gives verbose details on the LALR(1) states and all the conflicts. By inspecting the `.output` file it is possible to look at individual states and the operation done for each look-ahead token.

The conflicts are due to the unspecified *associativity* and *precedence* of operators. Arithmetic operators are generally *left associative* following the convention of evaluating expressions in a left-to-right manner. Another convention establishes that the multiplication `'*'` and division `'/'` operators have higher *precedence* than the addition `'+'` and subtraction `'-'` operators. Imposing these conventions could be achieved by rewriting the grammar to introduce terms and factors; more conveniently, *yacc* provides a way to specify how to solve conflicts without modifying the grammar, which we examine in the exercises below.

Exercises

1. Modify the specification in `calc.y` to allow for the usage of parentheses to explicitly specify the order of evaluation in expressions.

2. The grammar is ambiguous and the calculations are often incorrect. Test the program with `4*3-2`, for example. Solve all of the shift/reduce conflicts by specifying the precedence and associativity of operators.

We can specify the *precedence* and *associativity* of operators, simultaneously, using the keywords `%left`, `%right` and `%nonassoc` in the definitions section. For instance, `%left '+' '-'` states that the operators `'+'` and `'-'` are left associative and have equal precedence. Right associative operators use `%right` and non-associative operators use `%nonassoc`. Consider this example:

```
%left LOW
%left '+' '-'
%left HIGH
```

Operations are listed in order of increasing precedence. Operations listed on the same line have the same precedence. Therefore, this example specifies that `LOW` has a lower precedence than `'+'` and `'-'`, which in turn have a lower precedence than `HIGH`.

3. Modify the grammar to allow for the calculation of multiple independent expressions separated by commas. For example, entering `3-2-1,4*3-2,5*5/1*4` should output `0`, `10`, `100`.

4. Modify the grammar to accept if-then-else expressions that behave exactly like the ternary operator `?:` existing in C, Java and other programming languages. The syntax is: `if` *expression* `then` *expression* `else` *expression* (we need the tokens `IF`, `THEN` and `ELSE` from the lexical analysis exercises).

Notice that this modification introduces shift/reduce conflicts which can be solved by using the keyword `%prec` associated to the new grammar rule. Specifically, `%prec` appears after the rule, followed by a token or literal, and specifies that the precedence of the rule should be the precedence of the token or literal. In this case, the if-then-else rule should have the lowest precedence.

5. Modify the code to show the line and column numbers of syntax errors.

Author

Raul Barbosa (University of Coimbra)

References

Niemann, T. (2016) Lex & Yacc. https://epaperpress.com/lexandyacc

Levine, J. (2009). Flex & Bison: Text processing tools. O'Reilly Media.

Barbosa, R. (2023). Petit programming language and compiler. https://github.com/rbbarbosa/Petit

Aho, A. V. (2006). Compilers: Principles, techniques and tools, 2nd edition. Pearson Education.