

Compilers

Lecture 8

Murdoch J. Gabbay, Heriot-Watt University, Scotland

Remember:

LEX parses letters to words.

YACC parses words to sentences/programs.

We need to parse quickly and efficiently.

Table-driven parsing is one good way to implement a parser.

Different kinds of parsers: top-down

Table-driven parsing provides a powerful framework for parser construction.

But even for a simple parsing strategy, *e.g.* operator precedence parsing, constructing parse tables is laborious and error-prone to do by hand!

So automate the transition from production rule to parse table.

Table-driven parsing is the basis for *parser generators* or *compiler-compilers*.

(Some early parser generators were not table-driven, *e.g.* Tree-META (recursive descent). Most parser generators today adopt a bottom-up table-driven approach; they provide more powerful parsers.)

YACC (Yet Another Compiler-Compiler)

YACC takes a language specification, *i.e.* BNF-like grammar rules together with semantic actions, and generates a LALR(1) parser.

LALR(K) (**LookAhead LR(K)**) is a weak form of LR(K), the most general parsing strategy.

LR(K) generates many thousands of states for a typical programming language. LALR(K) addresses this state explosion by exploiting a sophisticated state merging strategy; it merges states which differ in only one lookahead symbol. This greatly reduces the state space, but can lead to reduce-reduce conflicts.

YACC is designed to work with LEX.

YACC source file structure

Definition section. Rules section. Subroutine section.

Syntactic skeleton:

```
/* DEFINITIONS */  
%%  
/* RULES */  
%%  
/* SUBROUTINES */
```

YACC for a propositional interpreter

Take the grammar

$$\begin{aligned} \langle \text{Prop} \rangle ::= & \langle \text{Prop} \rangle \text{ and } \langle \text{Prop} \rangle \mid \\ & \langle \text{Prop} \rangle \text{ or } \langle \text{Prop} \rangle \mid \\ & \text{not } \langle \text{Prop} \rangle \mid \\ & (\langle \text{Prop} \rangle) \mid \\ & \text{true} \mid \text{false} \end{aligned}$$

Your mission is: use LEX and YACC to construct an interpreter for this language.

OK, who can tell me what an interpreter is?

I know you know, and you know I know you know, but can you sum it up in one pithy phrase? Bet you can't ...

An interpreter is . . .

. . . a parser and an evaluator.

What's a compiler?

A compiler is a parser and a code-generator.

LEX and YACC

A YACC generated parser is implemented by a C function called `yyparse`.

The `yyparse` function expects a lexer function `yylex`. `yylex` can be generated by hand — but you'd probably want to use LEX.

`yyparse` terminates when the lexer function terminates, *i.e.* when `yylex` returns `0`.

YACC: the definition section

The definition section contains information on the tokens to recognize.

Single- and multi-character tokens are treated differently:

YACC: the definition section

Single-character tokens can be referenced using quote marks within the rules section, e.g. ' (' and ' , '

Multi-character tokens must be specified explicitly using a `%token` declaration in the definitions section.

For example to introduce `NUMBER` and `IDENT` to denote arbitrary numbers and identifiers, this should go in the definition section:

```
%token NUMBER IDENT
```

In the propositional example we need at least five tokens

```
%token AND OR NOT TRUE FALSE
```

YACC rules section

YACC rules let us encode a BNF-style grammar together with semantic actions.

The pattern for a rule is:

`non-terminal : right-hand-side { actions } ;`

Rules may be split over multiple lines.

Multiple definitions for a non-terminal can be represented using the vertical bar (|).

An example

Recall our example:

$$\begin{aligned} \langle \text{Prop} \rangle ::= & \langle \text{Prop} \rangle \text{ and } \langle \text{Prop} \rangle \mid \\ & \langle \text{Prop} \rangle \text{ or } \langle \text{Prop} \rangle \mid \\ & \text{not } \langle \text{Prop} \rangle \mid \\ & (\langle \text{Prop} \rangle) \mid \\ & \text{true} \mid \text{false} \end{aligned}$$

An example

The corresponding YACC rules look like this:

```
prop : prop AND prop
     | prop OR prop
     | NOT prop
     | '(' prop ')'
     | TRUE
     | FALSE
     ;
```

An example parse

The YACC subroutine section contains C code which is copied to the end of the output file `y.tab.c`.

A YACC parser is implemented as a C function `yyparse`. The subroutine section of a YACC specification will contain, by default, the following main function:

```
main() { yyparse(); }
```

In the propositional example we use main function:

```
main() { if (yyparse()) printf("FAILURE\n");  
        else  
                printf("SUCCESS\n"); }
```

Running YACC, take 1:

Apply YACC to the source file:

```
yacc prop3-1.y  
conflicts: 6 shift/reduce
```

what causes the conflicts?

Running YACC, take 1

Given an input string:

`true and true or false`

and assuming we have reached

`true and true`

in our analysis, then do we reduce using

$\langle \text{Prop} \rangle ::= \langle \text{Prop} \rangle \text{ and } \langle \text{Prop} \rangle$

to get $\langle \text{Prop} \rangle$, or shift the `or` expecting to reduce with

$\langle \text{Prop} \rangle ::= \langle \text{Prop} \rangle \text{ or } \langle \text{Prop} \rangle$

?

The grammar is ambiguous.

Eliminating ambiguity

The ambiguity on the previous slide has to do with operator precedence (and also associativity).

YACC has primitive support to specify operator precedence, and associativity rules.

Eliminating ambiguity

To give NOT higher precedence than AND and AND higher precedence than OR, and for AND and OR to associate to the right, then the following extensions to our declarations are required:

```
%token AND OR NOT TRUE FALSE
```

```
%right OR
```

```
%right AND
```

```
%nonassoc NOT
```

(%left declares an operator left associative.)

Running YACC, take 2

Apply YACC to the modified source file:

```
yacc -d prop3-1.y
```

The `-d` flag makes YACC generate a header file `y.tab.h`, as well as `y.tab.c`.

`y.tab.h` has definitions for the declared tokens, e.g.

```
# define AND 257
# define OR 258
# define NOT 259
# define TRUE 260
# define FALSE 261
```

We can include `y.tab.h` in the LEX source file for our lexer ...

Propositional lexer

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
#define EOS 0
}%
%%

[\\t ]+ /* ignore white space */ ;
"and"      { return AND; }
"or"       { return OR; }
"not"      { return NOT; }
"true"     { return TRUE; }
"false"    { return FALSE; }
.          { return yytext[0]; }
\\n        { return EOS; }

%%
```

Propositional lexer

(`\n` denotes EOS, the end-of-string. EOS is defined to be `0` and makes `yyparse` terminate.)

LEX and YACC: compilation and linking

Apply LEX to the source file:

```
lex prop3-1.l
```

Compile the YACC and LEX output files:

```
gcc -c lex.yy.c y.tab.c
```

Link the object files together with the LEX and YACC libraries:

```
gcc -o prop3.o lex.yy.o y.tab.o -ll -ly
```

running `prop3.o` does this ...

Propositional lexer in action

```
true  
SUCCESS
```

```
false  
SUCCESS
```

```
true and (false or not true)  
SUCCESS
```

```
true and not  
syntax error  
FAILURE
```

```
true and (false or not true  
syntax error  
FAILURE
```

Attaching semantic actions

Whenever a YACC parser uses a rule to perform a reduction step, the associated rule actions are performed:

non-terminal : **right-hand-side** { **actions** } ;

Action code can refer to the values of the right-hand-side symbols via the meta-symbols $\$1$, $\$2$, . . . **i.e.** where $\$1$ refers to argument position 1 and $\$2$ refers to argument position 2 and so on.

The value of the left-hand-side of a rule is denoted by the meta-symbol $\$\$$, **i.e.** the value of the reduced term.

Let's extend propositional parser with actions to **evaluate** the logical expressions.

Our propositional interpreter

```
state:  prop { if ($1) printf("your statement is true\n");
           else printf("your statement is false\n"); }
prop :   prop AND prop
        | prop OR  prop
        | NOT prop
        | '(' prop ')'
        | TRUE  { $$ = 1; }
        | FALSE { $$ = 0; }
        ;
```

(We exploited the C convention that zero is interpreted as **false** and any non-zero values is interpreted as **true**.)

Propositional interpreter in action

```
true  
your statement is true
```

```
false  
your statement is false
```

```
true and false  
your statement is false
```

```
true or false  
your statement is true
```

```
(true or not true) and not false  
your statement is true
```

Summary

YACC: defining tokens

YACC: specifying precedence & associativity of operators

YACC: rules – syntactic patterns and semantic actions

On-line propositional examples — see my webpage

`http://gabbay.org.uk`