CS143 Handout 13
Summer 2005 July 8, 2005

Introduction to yacc and bison

Handout written by Maggie Johnson and revised by Julie Zelenski.

yacc is a parser generator. It is to parsers what **lex** is to scanners. You provide the input of a grammar specification and it generates an LALR(1) parser to recognize sentences in that grammar. **yacc** stands for "yet another compiler compiler" and it is probably the most common of the LALR tools out there. Our programming projects are configured to use the updated version **bison**, a close relative of the yak, but all of the features we use are present in the original tool, so this handout serves as a brief overview of both. Our course web page include a link to an online bison user's manual for those who really want to dig deep and learn everything there is to learn about parser generators.

How It Works

yacc is designed for use with C code and generates a parser written in C. The parser is configured for use in conjunction with a **lex**-generated scanner and relies on standard shared features (token types, **yylval**, etc.) and calls the function **yylex** as a scanner co-routine. You provide a grammar specification file, which is traditionally named using a **.y** extension. You invoke **yacc** on the **.y** file and it creates the **y.tab.h** and **y.tab.c** files containing a thousand or so lines of intense C code that implements an efficient LALR(1) parser for your grammar, including the code for the actions you specified. The file provides an extern function **yyparse.y** that will attempt to successfully parse a valid sentence. You compile that C file normally, link with the rest of your code, and you have a parser! By default, the parser reads from **stdin** and writes to **stdout**, just like a **lex**-generated scanner does.

`% yacc myFile.y`	creates **y.tab.c** of C code for parser
`% gcc -c y.tab.c`	compiles parser code
`% gcc –o parse y.tab.o lex.yy.o –ll -ly`	link parser, scanner, libraries
`% parse`	invokes parser, reads from **stdin**

The Makefiles we provide for the projects will execute the above compilation steps for you, but it is worthwhile to understand the steps required.

`yacc` File Format

Your input file is organized as follows (note the intentional similarities to **`lex`**):

```
%{
Declarations
%}
Definitions
%%
Productions
%%
User subroutines
```

The optional Declarations and User subroutines sections are used for ordinary C code that you want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom. The optional Definitions section is where you configure various parser features such as defining token codes, establishing operator precedence and associativity, and setting up the global variables used to communicate between the scanner and parser. The required Productions section is where you specify the grammar rules. As in **`lex`**, you can associate an action with each pattern (this time a production), which allows you to do whatever processing is needed as you reduce using that production.

Example

Let's look at a simple, but complete, specification to get our bearings. Here is a **`yacc`** input file for a simple calculator that recognizes and evaluates binary postfix expressions using a stack.

```
%{
  #include <stdio.h>
  #include <assert.h>

  static int Pop();
  static int Top();
  static void Push(int val);
%}

%token T_Int

%%

S    :  S E '\n' { printf("= %d\n", Top()); }
     |
     ;


E    :  E E '+' { Push(Pop() + Pop()); }
     |  E E '-' { int op2 = Pop(); Push(Pop() - op2); }
     |  E E '*' { Push(Pop() * Pop()); }
     |  E E '/' { int op2 = Pop(); Push(Pop() / op2); }
     |  T_Int   { Push(yylval); }
     ;
```

```
%%

static int stack[100], count = 0;

static int Pop() {
    assert(count > 0);
    return stack[--count];
}
static int Top() {
    assert(count > 0);
    return stack[count-1];
}
static void Push(int val) {
    assert(count < sizeof(stack)/sizeof(*stack));
    stack[count++] = val;
}

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

A few things worth pointing out in the above example:

- All token types returned from the scanner must be defined using **`%token`** in the definitions section. This establishes the numeric codes that will be used by the scanner to tell the parser about each token scanned. In addition, the global variable **`yylval`** is used to store additional attribute information about the lexeme itself.
- For each rule, a colon is used in place of the arrow, a vertical bar separates the various productions, and a semicolon terminates the rule. Unlike **`lex`**, **`yacc`** pays no attention to line boundaries in the rules section, so you're free to use lots of space to make the grammar look pretty.
- Within the braces for the action associated with a production is just ordinary C code. If no action is present, the parser will take no action upon reducing that production.
- The first rule in the file is assumed to identify the start symbol for the grammar.
- **`yyparse`** is the function generated by **`yacc`**. It reads input from **`stdin`**, attempting to work its way back from the input through a series of reductions back to the start symbol. The return code from the function is 0 if the parse was successful and 1 otherwise. If it encounters an error (i.e. the next token in the input stream cannot be shifted), it calls the routine **`yyerror,`** which by default prints the generic "parse error" message and quits.

In order to try out our parser, we need to create the scanner for it. Here is the **lex** file we used:

```
%{
  #include "y.tab.h"
%}
%%
[0-9]+      { yylval = atoi(yytext); return T_Int;}
[-+*/\n]    { return yytext[0];}
.        { /* ignore everything else */ }
```

Given the above specification, `yylex` will return the ASCII representation of the calculator operators, recognize integers , and ignore all other characters. When it assembles a series of digits into an integer, it converts to a numeric value and stores in **`yylval`** (the global reserved for passing attributes from the scanner to the parser). The token type **`T_Int`** is returned.

In order to tie this all together, we first run **`yacc`**/**`bison`** on the grammar specification to generate the **`y.tab.c`** and **`y.tab.h`** files, and then run **`lex`**/**`flex`** on the scanner specification to generate the **`lex.yy.c`** file. Compile the two **`.c`** files and link them together, and voila— a calculator is born! Here is the **`Makefile`** we could use:

```
calc:    lex.yy.o y.tab.o
         gcc -o calc lex.yy.o y.tab.o  -ly -ll

lex.yy.c:   calc.l y.tab.c
            flex calc.l

y.tab.c:    calc.y
            bison -vdty calc.y
```

Tokens, Productions, and Actions

By default, **`lex`** and **`yacc`** agree to use the ASCII character codes for all single-char tokens without requiring explicit definitions. For all multi-char tokens, there needs to be an agreed-upon numbering system, and all of these tokens need to be specifically defined. The **`%token`** directive establishes token names and assigns numbers.

```
%token T_Int T_Double T_String T_While
```

The above line would be included in the definitions section. It is translated by **`yacc`** into C code as a sequence of **`#define`**s for **`T_Int`**, **`T_Double`**, and so on, using the numbers **`257`** and higher. These are the codes returned from **`yylex`** as each multicharacter token is scanned and identified. The C declarations for the token codes are exported in the generated **`y.tab.h`** header file. **`#include`** that file in other modules (in the scanner, for example) to stay in sync with the parser-generated definitions.

Productions and their accompanying actions are specified using the following format:

```
left_side    :  right_side1 { action1 }
             |  right_side2 { action2 }
             |  right_side3 { action3 }
             ;
```

The left side is the non-terminal being described. Non-terminals are named like typical identifiers: using a sequence of letters and/or digits, starting with a letter. Non-terminals do not have to be defined before use, but they do need to be defined eventually. The first non-terminal defined in the file is assumed to the start symbol for the grammar.

Each right side is a valid expansion for the left side non-terminal. Vertical bars separate the alternatives, and the entire list is punctuated by a semicolon. Each right side is a list of grammar symbols, separated by white space. The symbols can either be non-terminals or terminals. Terminal symbols can either be individual character constants, e.g. **`'a'`** or token codes defined using **`%token`** such as **`T_While`**.

The C code enclosed in curly braces after each right side is the associated action. Whenever the parser recognizes that production and is ready to reduce, it executes the action to process it. When the entire right side is assembled on top of the parse stack and the lookahead indicates a reduce is appropriate, the associated action is executed right before popping the handle off the stack and following the goto for the left side non-terminal. The code you include in the actions depends on what processing you need. The action might be to build a section of the parse tree, evaluate an arithmetic expression, declare a variable, or generate code to execute an assignment statement.

Although it is most common for actions to appear at the end of the right side, it is also possible to place actions in-between grammar symbols. Those actions will be executed at that point when the symbols to the left are on the stack and the symbols to the right are coming up. These embedded actions can be a little tricky because they require the parser to commit to the current production early, more like a predictive parser, and can introduce conflicts if there are still open alternatives at that point in the parse.

Symbol Attributes

The parser allows you to associate attributes with each grammar symbol, both terminals and non-terminals. For terminals, the global variable **`yylval`** is used to communicate the particulars about the token just scanned from the scanner to the parser. For non-terminals, you can explicitly access and set their attributes using the attribute stack.

By default, **`YYSTYPE`** (the attribute type) is just an integer. Usually you want to different information for various symbol types, so a union type can be used instead. You indicate what fields you want in the union via the **`%union`** directive.

```
%union {
   int intValue;
   double doubleValue;
   char *stringValue;
}
```

The above line would be included in the definitions section. It is translated by **`yacc`** into C code as a **`YYSTYPE typedef`** for a new union type with the above fields. The global variable **`yylval`** is of this type, and parser stores variables of this type on the parser stack, one for each symbol currently on the stack.

When defining each token, you can identify which field of the union is applicable to this token type by preceding the token name with the fieldname enclosed in angle brackets. The fieldname is optional (for example, it is not relevant for tokens without attributes).

```
%token <intValue>T_Int <doubleValue>T_Double T_While T_Return
```

To set the attribute for a non-terminal, use the **`%type`** directive, also in the definitions section. This establishes which field of the union is applicable to instances of the non-terminal:

```
%type<intValue> Integer IntExpression
```

To access a grammar symbol's attribute from the parse stack, there are special variables available for the C code within an action. **`$n`** is the attribute for the nth symbol of the current right side, counting from **`1`** for the first symbol. The attribute for the non-terminal on the left side is denoted by **`$$`**. If you have set the type of the token or non-terminal, then it is clear which field of the attributes union you are accessing. If you have not set the type (or you want to overrule the defined field), you can specify with the notation **`$<fieldname>n`**. A typical use of attributes in an action might be to gather the attributes of the various symbols on the right side and use that information to set the attribute for the non-terminal on the left side.

A similar mechanism is used to obtain information about symbol locations. For each symbol on the stack, the parser maintains a variable of type **`YYLTYPE`**, which is a structure containing four members: first line, first column, last line, and last column. To obtain the location of a grammar symbol on the right side, you simply use the notation `@n`, completely parallel to **`$n`**. The location of a terminal symbol is furnished by the lexical analyzer via the global variable **`yylloc`**. During a reduction, the location of the non-terminal on the left side is automatically set using the combined location of all symbols in the handle that is being reduced.

Conflict Resolution

What happens when you feed **`yacc`** a grammar that is not LALR(1)? **`yacc`** reports any conflicts when trying to fill in the table, but rather than just throwing up its hands, it has automatic rules for resolving the conflicts and building a table anyway. For a shift/reduce conflict, **yacc** will choose the shift. In a reduce/reduce conflict, it will reduce using the rule declared first in the file. These heuristics can change the language that is accepted and may not be what you want. Even if it happens to work out, it is not recommended to let **yacc** pick for you. You should control what happens by explicitly declaring precedence and associativity for your operators.

For example, ask **yacc** to generate a parser for this ambiguous expression grammar that includes addition, multiplication, and exponentiation (using `'^'`).

```
%token T_Int
%%

E   : E '+' E
    | E '*' E
    | E '^' E
    | T_Int
    ;
```

When you run yacc on this file, it reports:

```
conflicts: 9 shift/reduce
```

In the generated **y.output**, it tells you more about the issue:

```
...
State 6 contains 3 shift/reduce conflicts.
State 7 contains 3 shift/reduce conflicts.
State 8 contains 3 shift/reduce conflicts.
...
```

If you look through the human-readable **y.output** file, you will see it contains the family of configurating sets and the transitions between them. When you look at states 6, 7, and 8, you will see the place we are in the parse and the details of the conflict. Understanding all that LR(1) construction stuff just might be useful after all! Rather than re-writing the grammar to implicitly control the precedence and associativity with a bunch of intermediate non-terminals, we can directly indicate the precedence so that **yacc** will know how to break ties. In the definitions section, we can add any number of precedence levels, one per line, from lowest to highest, and indicate the associativity (either left, right, or nonassoc). Several terminals can be on the same line to assign them equal precedence.

```
%token T_Int
%left '+'
%left '*'
%right '^'
%%

E   : E '+' E
    | E '*' E
    | E '^' E
    | T_Int
    ;
```

The above file says that addition has the lowest precedence and it associates left to right. Multiplication is higher, and is also left associative. Exponentiation is highest precedence and it associates right. These directives disambiguate the conflicts. When we feed **yacc** the changed file, it uses the precedence and associativity as specified to break ties. For a shift/reduce conflict, if the precedence of the token to be shifted is higher than that of the rule to reduce, it chooses the shift and vice versa. The precedence of a rule is determined by the precedence of the rightmost terminal on the right-hand side (or can be explicitly set with the **%prec** directive). So if a **4 + 5** is on the stack and ***** is coming up, the ***** has higher precedence than the **4 + 5**, so it shifts. If **4 * 5** is on the stack and **+** is coming up, it reduces. If **4 + 5** is on the stack and **+** is coming up, the associativity breaks the tie, a left-to-right associativity would reduce the rule and then go on, a right-to-left would shift and postpone the reduction.

Another way to set precedence is by using the **%prec** directive. When placed at the end of a production with a terminal symbol as its argument, it explicitly sets the precedence of the production to the same precedence as that terminal. This can be used when the right side has no terminals or when you want to overrule the precedence given by the rightmost terminal.

Even though it doesn't seem like a precedence problem, the dangling else ambiguity can be resolved using precedence rules. Think carefully about what the conflict is: Identify what the token is that could be shifted and the alternate production that could be reduced. What would be the effect of choosing the shift? What is the effect of choosing to reduce? Which is the one we want?

Using **yacc**'s precedence rules, you can force the choice you want by setting the precedence of the token being shifted versus the precedence of the rule being reduced. Whichever precedence is higher wins out. The precedence of the token is set using the ordinary **%left**, **%right**, or **%nonassoc** directives. The precedence of the rule being reduced is determined by the precedence of the rightmost terminal (set the same way) or via an explicit **%prec** directive on the production.

Error handling

When a **`yacc`**-generated parser encounters an error (i.e. the next input token cannot be shifted given the sequence of things so far on the stack), it calls the default **`yyerror`** routine to print a generic "parse error" message and halt parsing. However, quitting in response to the very first error is not particularly helpful!

`yacc` supports a form of error re-synchronization that allows you to define where in the stream to give up on an unsuccessful parse and how far to scan ahead and try to cleanup to allow the parse to restart. The special **`error`** token can be used in the right side of a production to mark a context in which to attempt error recovery. The usual use is to add the error token possibly followed by a sequence of one or more synchronizing tokens, which tell the parser to discard any tokens until it sees that "familiar" sequence that allows the parser to continue. A simple and useful strategy might be simply to skip the rest of the current input line or current statement when an error is detected.
When an error is encountered (and reported via **`yyerror`**,) the parser will discard any partially parsed rules (i.e. pop stacks from the parse stack) until it finds one in which it can shift an error token. It then reads and discards input tokens until it finds one that can follow the error token in that production. For example:

```
Var  :  Modifiers Type IdentifierList ';'
     |  error ';'
     ;
```

The second production allows for handling errors encountered when trying to recognize **`Var`** by accepting the alternate sequence **`error`** followed by a semicolon. What happens if the parser in the in the middle of processing the **`IdentifierList`** when it encounters an error? The error recovery rule, interpreted strictly, applies to the precise sequence of an **`error`** and a semicolon. If an error occurs in the middle of an **`IdentifierList`**, there are **`Modifiers`** and a **`Type`** and what not on the stack, which doesn't seem to fit the pattern. However, yacc can force the situation to fit the rule, by discarding the partially processed rules (i.e. popping states from the stack) until it gets back to a state in which the **`error`** token is acceptable (i.e. all the way back to the state at which it started before matching the **`Modifiers`**). At this point the **`error`** token can be shifted. Then, if the lookahead token couldn't possibly be shifted, the parser reads tokens, discarding them until it finds a token that is acceptable. In this example, yacc reads and discards input until the next semi-colon so that the error rule can apply. It then reduces that to **`Var`**, and continues on from there. This basically allowed for a mangled variable declaration to be ignored up to the terminating semicolon, which was a reasonable attempt at getting the parser back on track. Note that if a specific token follows the error symbol it will discard until it finds that token, otherwise it will discard until it finds any token in the lookahead set for the nonterminal (for example, anything that can follow **`Var`** in the example above).

In our postfix calculator from the beginning of the handout, we can add an **`error`** production to recover from a malformed expression by discarding the rest of the line up to the newline and allow the next line to be parsed normally:

```
S   :  S E '\n'   { printf("= %d\n", Top()); }
    |  error '\n' { printf("Error! Discarding entire line.\n"); }
    |
    ;
```

Like any other yacc rule, one that contains an error can have an associated action. It would be typical at this point to clean up after the error and other necessary housekeeping so that the parse can resume.

Where should one put error productions? It's more of an art than a science. Putting error tokens fairly high up tends to protect you from all sorts of errors by ensuring there is always a rule to which the parser can recover. On the other hand, you usually want to discard as little input as possible before recovering, and error tokens at lower level rules can help minimize the number of partially matched rules that are discarded during recovery.

One reasonable strategy is to add error tokens fairly high up and use punctuation as the synchronizing tokens. For example, in a programming language expecting a list of declarations, it might make sense to allow error as one of the alternatives for a list entry. If punctuation separates elements of a list, you can use that punctuation in error rules to help find synchronization points—skipping ahead to the next parameter or the next statement in a list. Trying to add error actions at too low a level (say in expression handling) tends to be more difficult because of the lack of structure that allows you to determine when the expression ends and where to pick back up. Sometimes adding error actions into more than one level may introduce conflicts into the grammar because more than one error action is a possible recovery.

Another mechanism: deliberately add incorrect productions into the grammar specification, allow the parser to help you recognize these illegal forms, and then use the action to handle the error manually. For example, let's say you're writing a Java compiler and you know some poor C programmer is going to forget that you can't specify the size in a Java array declaration. You could add an alternate array declaration that allows for a size, but knowing it was illegal, the action for this production reports an error and discards the erroneous size token.

Most compilers tend to focus their efforts on trying to recover from the most common error situations (forgetting a semicolon, mistyping a keyword), but don't put a lot of effort into dealing with more wacky inputs. Error recovery is largely a trial and error process. For fun, you can experiment with your favorite compilers to see how good a job they do at recognizing errors (they are expected to hit this one perfectly), reporting errors clearly (less than perfect), and gracefully recovering (far from perfect).

Other Features

This handout doesn't detail all the great features available in yacc and bison, please refer to the man pages and online references for more details on tokens and types, conflict resolution, symbol attributes, embedded actions, error handling, and so on.

Bibliography

J. Levine, T. Mason, and D. Brown, Lex and Yacc. Sebastopol, CA: O'Reilly & Associates, 1992.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.