CS 434 Meeting 12 — 3/14/06

YACC

1. Yacc is a parser generator.

2. Basic form of input specification contains three main secitons

   (a) Declarations and directives for the parser generator

   (b) Rules of grammar

   (c) Program text (i.e. action routines, ...)

   separated by lines containing only "%%".

3. Grammar Rules

   - Each rule takes the form:

     non-term : BODY-1 | . . . | BODY-n ;

   - A BODY is just a list of token names, non-terminal names and literals.

   - A literal is a quoted character:

     expr : expr '+' term

     You will probably not use any literals.

   - Token names are used to identify tokens such as the keywords and token classes such as identifier or integer constant.
     - You can't use quoted strings to identify such tokens literally because all the scanner gives to the parser is a token class and token value.
     - So that Yacc can distinguish token names from non-terminal names, all token names must be declared as such.

   - Names are composed of letters, digits, underscores and periods. Each name must start with a letter. Case is significant.

4. Declarations.

   - To declare a token name, type

     %token name

     in the first section of the specification file

   - One can also specify the numeric value that the scanner will return when it sees this token. It is generally easier, however, to let YACC assign token numbers. This is true because YACC produces a file named 'y.tab.h' (which we will rename 'tn.h') that contains `#define`'s associating token names that can be used in the scanner with the numbers assigned.

5. Standardizing dates: An example.

```
%token MONTH
%token NUMBER

%%
date : day MONTH year
     | monthnum '/' day '/' year
     | MONTH day ',' year
     ;

monthnum : NUMBER ;
day  : NUMBER  ;
year : NUMBER  ;
```

   - The y.tab.h file produced when YACC processes this file looks like:

```
# define MONTH 257
# define NUMBER 258
```

Semantic Actions, Semantic Records and All That

1. Given an "automatic parser generator" like YACC, we need a way to indicate what "(semantic) actions" should be associated with each grammar rule when it is used in a parse to produce the desired intermediate form.

2. A grammar can be *annotated* to indicate where semantic processing should occur by naming the actions and including the names at appropriate places in the grammar.

- For example, the annotations on the production for the else-less if statement for a one-pass compiler that generated assembly language code might look like:

  $< \text{stmt} > \rightarrow$ **if** $< \text{expr} >$ *#gen-else-branch*
  then $< \text{stmt} >$ **end** *#gen-else-target*

3. A semantic action will typically produce a value to be associated with a node in the parse tree (not the syntax tree). It may need to access the values associated with the children of the node it is to label with a value.

   - Imagine semantic actions routines that generate code for expressions:

     $< \text{expr} > \rightarrow < \text{expr} > + < \text{expr} >$ *#gen-add*

     – Each semantic action will need to know where the value computed for each sub-tree of the expression will be found at run time (i.e. what register or memory locations). This information will be provided as the labels associated with nodes in the parse tree by the semantic action routines.

4. The parser should (can, must) help by keeping track of the values produced by semantic actions as the parser proceeds.

   - This is simple in a bottom-up parser. When the handle is reduced, the value produced by any action associated with the production used is placed in the stack with the non-terminal to which the handle was reduced.

5. The parser generator should (can, must) provide a way that actions can access values associated with terminals and non-terminals in the handle and specify the value to be associated with the non-terminal to which the handle is reduced.

Specifying Semantic Actions in YACC

1. Specifying actions in YACC is easy. You simply place C code which performs the action in braces (i.e. '{' and '}') at the point in the production where you want the action to occur.

   - You can put actions in the middle of rules. Putting them only at the ends, however, is a bit safer.

2. Passing semantic values around is a bit trickier. The mechanisms used are illustrated in Figure 1. The actions in the example grammar cause the parser to take whatever date is read and echo it in the "standard" form "March 7, 2004".

   - Yacc keeps the semantic values associated with a terminal or non-terminal symbol in its parse stack until the phrase containing the symbol is reduced. We will not worry for now about how the scanner passes a "token value" to Yacc (in addition to a token number).
   - In an action, $1,$2, ... can be used to refer to the values associated with the symbols in the right hand side of the production.
   - In an action, $$ can be used to refer to the value associated with the non-terminal on the left hand side of the production. Typically, $$ appears as the target of an assignment in an action.
   - Declarations of types and variables used in the actions can be placed in the first section of the YACC input file if preceded by a line containing the characters "%{" and followed by a line containing "%}".

3. If semantic values of any type other than integer are used, you must define the type name "YYSTYPE" to describe the type used (This definition can be included in the first section of the input file).

4. If the type YYSTYPE is a union type (it usually is) you should (can, must) tell YACC which member of the union will be associated with each terminal and non-terminal whose value is set or used.

   - For tokens this is done by including the member name between '<' and '>' symbols in each `%token` declaration.
   - For non-terminals a separate `%type` declaration is used.

Building Scanners with Lex

```
%{
typedef union{
  int num;
  char *str;
} YYSTYPE;

char *monthtab[] =
  {   "January",      "February",       "March",        "April",
      "May",          "June",           "July",         "August",
      "September",    "October",        "November",     "December"
};
%}

%token <str> MONTH
%token <num> NUMBER

%type <str> monthnum
%type <num> day year
%%
date : day MONTH year             { printf("%s %d, %d", $2, $1, $3); }
     | monthnum '/' day '/' year  { printf("%s %d, %d", $1, $3, $5); }
     | MONTH day ',' year         { printf("%s %d, %d", $1, $2, $4); }
     ;

monthnum : NUMBER           { $$ = monthtab[$1 - 1 ] ; }
         ;

day  : NUMBER ;

year : NUMBER ;
```

Figure 1:

- There is a tool for building scanners that is very similar to Yacc named Lex.
- While Yacc views the input file as a unit onto which it must impose some structure, Lex views the input as a stream of many small entities it must recognize.

  A Lex definition is therefore a series of descriptions of the different types of "small entities" that might appear paired with "actions" (more C code) describing what to do if they appear.
- In Lex, a notation for writing regular expressions is used to describe the types of tokens the scanner should look for.
- Like a Yacc input file, a Lex input file has three parts separated from one another by lines beginning with a pair of percent signs.

  1. The first section contains declaration of two types:
     - C declarations to be included as part of the final C source for the parser function "yylex()".
     - declarations of identifers that can be used as predefined regular expression in the next "rule" section.
  2. The second section is the actual rules describing the scanner. Each rule consists of a regular expression and some C code to execute when the regular expression is "matched".
     - The two must appear on a line together separated by some blank space.
     - It is typical to place the C code within braces (this is required if there is more than a single statement).
  3. The third section can be used to add definitions of functions called by scanner actions.
- The fun part is the syntax for regular expressions.
  - Any non-special character (all alphabetics, numerics and some punctuation) can be used as regular expressions that match themselves.

  - Two regular expressions (including simple ones like single characters) concatenated together match any string formed by concatenating anything that matched the first regular expression with something that matched the second.
  - A regular expression formed by placing a "|" between two other expression matches anything that matches either of the subexpressions.
  - parentheses can be used for grouping.
  - A regular expression followed by a "*" matches zero or more copies of the string matched by the original expression.
  - A regular expression followed by a "+" matches one or more copies of the original. A regular expression followed by a ? matches 0 or 1 copies of the original.
  - The regular expression . matches any single character.
  - The regular expression formed by placing a group of characters between square brackets (i.e. [and]) matches any single character in the group.
  - Within square brackets, a sequence of consequtive characters can abbreviated using a dash as in "[a-z]".
  - etc.

- There are many other notations supported (read the handout). The most significant is that Lex needs to provide some way to tell it that the user wants to match a string containing a character like "(" or "*" that would normally be interpreted as a metacharacter.
  - Any quoted string (double quotes) matches itself.
  - A character preceded by a backslash, "\", matches the character itself.

- To improve the clarity of a Lex specification, you can associate names with commonly used or complex subexpressions.
  - The definitions of such names appear in the first section of the Lex input file (i.e. before the first %%).
  - Each definition is just a line containing a name separated from a regular expression by blanks and or tabs.
  - To use such a name in a later regular expression, you just surround the name with curly braces.

```
alpha [a-zA-Z]
...
%%
{alpha}+     printf("I found a word");
```

- It is possible that several of the regular expressions used in rules in a Lex rule file might match prefixes of the same input. In such cases, Lex chooses the longest possible match and, among matches of equal length, the match with the earliest rule in the file.

- As in Yacc, the actions one pairs with the regular expressions in a Lex file are just bits of C code.
  - To return a token, simply include a "return" statement that returns the token's token number (which can be referred to symbolically using one of the #defined names from the y.tab.h file Yacc produces).
  - If you also need to associate a token value with the token, some extra code must be included in the action to assign the appropriate value to the variable "yylval".
  - "yylval" is actually defined external to Lex (by Yacc) and is of type YYSTYPE (the semantic value type used in the associated Yacc file).

- Within the actions, there are some special variables that can be used to access information about the strings matched:
  - yytext is the string matched.
  - yyleng is the length of the match.

- If you don't include a return in an action, Lex will happilly go on matching substrings to regular expressions.

- If no regular expression matches some portion of the input, Lex will copy the unmatched character to standard output.
- With these facts about Lex in mind, the following is an example of a Lex specfication appropriate for our dates example.

```
%{

#include "tn.h"

typedef union {
  int num;
  char *str;
} YYSTYPE;

extern YYSTYPE yylval;

char * makecopy( char * str);

%}

alfa [a-zA-Z]
digit [0-9]

%%
","       { return ',' ; }
"/"       { return '/' ; }
{alfa}+   { yylval.str = makecopy(yytext); return MONTH;
}
{digit}+  { yylval.num = atoi(yytext); return NUMBER; }
.         { }

%%

char * makecopy( char * str) {
 char * result;
 result = malloc( strlen( str) + 1);
 strcpy(result,str);
 return result;
}
```