CS 434 Fall 2003

C° Compiler Implementation Project
Phase ??? : Building a Parser with YACC

Due: November 6, 2003

Given the limited time available, I don't actually have you write a complete compiler in this course. In particular, rather than have you take the time to build a complete syntactic analyzer for C°, I only ask you to construct a parser and scanner for a subset of the language. This will give you experience using a parser and scanner generation tools, but will take much less time than the construction of a complete C° syntax analyzer. The construction of such a mini-parser will be the goal of this assignment. Since it will never really be part of the C° compilers you complete at the end of the semester, I haven't given it a phase number. In the remaining "phases" of the project, you will continue to use the scanner and parser I provide.

The language your parser should accept is defined by taking the C° grammar in the *Revised Report on the C° Programming Language* handout and:

1. replacing the two productions defining the non-terminal < function definition list > by the single production

   $< \text{function definition list} > \rightarrow \epsilon$

2. replacing the two productions defining the non-terminal < statement list > by the single production

   $< \text{statement list} > \rightarrow \epsilon$

3. removing all productions that become useless after the preceding changes are made.

You should begin by creating a new directory for this phase of the project (While you will not use the parser you produce for this assignment later in the project, you will use the code you produced for phase 2. So, you should avoid modifying phase 2 in any way while working with YACC and Lex.) Copy the files `Makefile`, `parser.y` and `scanner.l` from the directory `~tom/shared/434/Cdim/yaccing` into the directory you have created for your parser. There are several other files in the yaccing directory, but you should only copy these three!

Remember that in order to allow `make` to properly account for the references your source file makes to '.h' files you must execute the command:

```
make depend
```

after first making these copies and whenever you add or change any `#includes`.

The `parser.y` file contains the beginnings of the parser you need to create. In particular it contains `%token` declarations for all of the terminal symbols used. It also contains includes for several .h files that describe the syntax trees to be produced and the interfaces of several routines I have provided that you may want to use.

Obviously, you need to add the specification of the grammar itself to this file. You will also need to add `%type` declarations for the non-terminal names you use. Remember that once you start adding semantic actions, YACC will be unable to produce a parser without lots of type errors until you finish the job. So, I strongly urge you to first enter a valid grammar for the language and get YACC to produce a working parser that does nothing other than decide whether an input is or is not valid. Then add your semantic actions.

```
struct lexval {
  int line;
};

struct listhead {
  node *head, *tail;
};

typedef union {
  node *treenode;
  struct lexval lexeme;
  struct listhead list;
} YYSTYPE;

/* Tree building operations provided by syntree.c */

   /* makesingle  -  Make a node with a one child.    */
node *makesingle(nodetype op, node *child, int line);

   /* makepair  -  Make a node with a two children.    */
node *makepair(nodetype op, node *child1, node *child2, int line);

   /* maketriple  -  Make a node with a three children.    */
node *maketriple(nodetype op, node *child1, node *child2, node *child3,
 int line);
```

Figure 1: Declarations from ~tom/shared/434/Cdim/syntax.h

The `scanner.l` file similarly contains the skeleton of a Lex specification for a scanner. You will need to add regular expressions and actions for the tokens associated with the subset of C° you need to parse. Like the parser file, this file includes several .h files you will need to complete the scanner. It also defines a yyerror function which is used by the generated scanner and parser to print error messages.

One of the .h files that you will be using in this project is named `syntax.h`. This file contains declarations of the types used when associating semantic values with non-terminals during the parse. The most important declarations from this file are shown in figure 1. The most important declaration is that of the union type `YYSTYPE`. As explained in class, this is the type that describes to YACC the values that will be associated with tokens and non-terminals. The union type this declaration associates with the name `YYSTYPE` includes three members. The `treenode` member of the union is used when a syntax tree node is to be associated with a non-terminal or terminal. The `lexeme` member of the union is used for most tokens returned by the scanner. It simply provides a way that the scanner can associate a line number with each token it returns to the parser. The scanner should associate Nident and Nconst nodes with identifier and constants when they are returned as tokens. A lookup function is provided that will keep a hash table of all identifier seen and build a unique Nident node for each identifier. Note: The scanner is also expected to return an Nident node for the keyword integer even though it is treated as a distinct token type.

The `list` member of the YYSTYPE union can be used when building any of the various lists found in our syntax trees. If you can't figure out what it is for, remember that I explained in class that right recursive rules are evil if a bottom up parser is being used.

Several functions are provided in the file syntree.o that make the actual construction of syntax tree nodes simple. The functions `makesingle`, `makepair` and `maketriple` can be used to create nodes with 1, 2 or 3 children respectively. In addition to pointers to the nodes which are to become the children of the new node, each of these routines expects a node type argument (`op`) and a line number that should be associated with the new node (`line`). The other functions declared in `syntax.h` — `errornode` and `makelist` — will not be of use in your parser.

You will not need to create a `main.c` file for this phase. The `Makefile` is set up to include the file `~tom/shared/434/Cdim/yaccing/main.o` in the `Cdim` executable it builds using your parser. The function `main` defined in this `main.o` file is basically like the `main` I gave you to start phase 1. It initializes the symbol table, calls `yyparse` and then uses `printree` to display the syntax tree built by the parser. The new `main`, however, has two handy features. If you give it a file name as a parameter it will read its input from that file instead of from its standard input. So, you can type either

```
Cdim sample.c
```

or

```
Cdim < sample.c
```

More importantly, the new version of main recognizes one "option". If you type

```
Cdim -d ...
```

the main program will set a flag that enables debugging features of the parser built by YACC before invoking `yyparse`. When this flag is set, your YACC-built parser will display somewhat readable messages telling you which terminals it has read and which reductions it has applied. You will probably find this very helpful, since the only other feedback YACC will give you if something goes wrong during a parse is the message `syntax error`. That's right, not even a line number!