```
$Id: asg3-parser.mm,v 1.1 2014-08-08 16:10:33-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments/
```

1. Overview

Augment your string table and scanner by adding an `oc` parser to the project. The output of this project will be an abstract syntax tree written to the file *program*`.ast`, in addition to the files generated from the previous two projects. All specifications from the first two projects apply, and in addition, the `-y` flag must turn on `yydebug`.

SYNOPSIS

`oc` [`-ly`] [`-@` *flag* ...] [`-D` *string*] *program*`.oc`

The main function will open a pipe to the C preprocessor as before, but will never call `yylex()`. Instead it makes a single call to `yyparse()`, which will call `yylex()` as needed. When `yyparse()` returns, the main function will call a function to dump the AST. The function `yyparse()` calls `yyerror()` to print an error message whenever it detects a syntax error and has to recover. The `-y` flag must turn on the `yydebug` switch. Generate a file called *program*`.ast` based on the input file name, and also generate all files specified in earlier projects.

[***x***]	Brackets indicate the the symbols are optional.
[***x***]...	Brackets and three dots mean that the symbol(s) occur zero or more times.
x **|** ***y***	A bar indicates alternation between its left and right operands.
'`while`'	Symbols representing themselves and written in `Courier bold` and quoted.
symbol	Nonterminal symbols in the grammar are written in lower case Roman.
TOKEN	Token classes with lexical information are written in *UPPER CASE SMALL ITALIC*.

Figure 1. Metagrammar for `oc`

2. The Metagrammar

When reading the grammar of `oc`, it is important to distinguish between the grammar and the metagrammar. the metagrammar is the grammar that describes the grammar. You must also use your knowledge of C to fill in what is only implied. The metalanguage redundantly uses fonts and typography to represent concepts for the benefit of those reading this document via simple ASCII text. It looks prettier in the Postscript version. Note that the meta-brackets and meta-stick are slightly larger than normal type. The notation used is shown in Figure 1.

3. The Grammar of `oc`

Figure 2 shows the context-free grammar of `oc`. Your task is to translate that descriptive user-grammar into LALR(1) form acceptable to `bison`. You may, of course, take advantage of `bison`'s ability to handle ambiguous grammars via the use of precedence and associativity declarations. The dangling `else` problem should also

```
program    → [ structdef | function | statement ]...
structdef  → 'struct' TYPEID '{' [ fielddecl ';' ]... '}'
fielddecl  → basetype [ '[]' ] FIELD
basetype   → 'void' | 'bool' | 'char' | 'int' | 'string' | TYPEID
function   → identdecl '(' [ identdecl [ ',' identdecl ]... ] ')' block
identdecl  → basetype [ '[]' ] DECLID
block      → '{' [ statement ]... '}' | ';'
statement  → block | vardecl | while | ifelse | return | expr ';'
vardecl    → identdecl '=' expr ';'
while      → 'while' '(' expr ')' statement
ifelse     → 'if' '(' expr ')' statement [ 'else' statement ]
return     → 'return' [ expr ] ';'
expr       → expr BINOP expr | UNOP expr | allocator | call | '(' expr ')'
           | variable | constant
allocator  → 'new' TYPEID '(' ')' | 'new' 'string' '(' expr ')'
           | 'new' basetype '[' expr ']'
call       → IDENT '(' [ expr [ ',' expr ]... ] ')'
variable   → IDENT | expr '[' expr ']' | expr '.' FIELD
constant   → INTCON | CHARCON | STRINGCON | 'false' | 'true' | 'null'
```

Figure 2. Grammar of `oc`

Precedence	Associativity	Arity	Fixity	Operators
lowest	right	binary/ternary	matchfix	`if else`
	right	binary	infix	`=`
	left	binary	infix	`== != < <= > >=`
•	left	binary	infix	`+ -`
•	left	binary	infix	`* / %`
•	right	unary	prefix	`+ - ! ord chr`
	left	binary/variadic	postfix	*e*`[`*e*`]` *e*.*i* *f*`(...)`
	—	unary/binary	prefix	`new`
highest	—	unary	matchfix	`(`*e*`)`

Figure 3. Operator precedence in `oc`

be handled in that way.

You will not be able to feed the grammar above to `bison`, because it will not be able to handle *BINOP* and *UNOP* as you might expect. You will need to explicitly enumerate all possible rules with operators in them. However, using `bison`'s operator precedence declarations, the number of necessary rules will be reduced. Figure 3 shows operator precedence and associativity.

There is actually more information there than that which will be useful in `%left` and `%right` declarations. In addition, it is necessary to eliminate the metagrammar's optional and repetitive brackets, a feature that `bison` does not have.

4. Constructing the Abstract Syntax Tree

The abstract syntax tree (AST) is constructed in such a way that all operators and operator-like tokens are the parents of their operands, which are adopted as their children. The children may be leaf nodes (identifiers or constants) or other expressions. Constants and identifiers are always leaf nodes. In general, interior nodes may have an arbitrarily large number of children. This is the case wherever the grammar shows "..." indicating repetition.

There are also several situations where the scanner can not distinguish between tokens which have the same lexical structure, but become different syntactically. For example, the operators `+` and `-` may be either unary or binary, and a sequence of characters like `-123` must be scanned as two separate tokens.

Also, the difference between an identifier and a type id can only be determined by the parser, not the scanner, as can the overloaded use of `=` as either a variable declaration initializer or an assignment operator.

4.1 The Root Token

At the root of the entire AST is the root token, with code `TOK_ROOT`. This is a token synthesized by the parser, since there are no characters at the beginning of the program for the scanner to make with this. The parser creates this root token as its first action, when reducing the empty right hand side to a `program`.

4.2 Identifiers

There are multiple different tokens all of which have the appearance of being identifiers. The scanner distinguishes and returns special token codes for reserved words, but returns all others as `TOK_IDENT`. The parser must then substitute `TOK_IDENT` depending on the context. This makes the symbol table and type checker easier to implement.

(a) In a field declaration, substitute `TOK_FIELD`.

(b) In a variable or function declaration, substitute `TOK_DECLID`. `TOK_DECLID` A `TOK_DECLID` and a `TOK_IDENT` are exactly the same, except that the first appears in a declaration and the second in a variable or function reference.

(c) In a structure type declaration, or a use of a structure type in a variable declaration or function return type, substitute `TOK_TYPEID`.

4.3 Structure Definitions

A structure defines a new data type and may only appear as a global statement. The `struct` keyword can serve as the root, with the braces and internal semicolons being discarded. The `TYPEID` name is the first child, similar to a call statement, with each of the field definitions following as the other children in their declared order. The scanner will return the `TYPEID` name as a `TOK_IDENT`, so change it to `TOK_TYPEID`.

4.4 Operators

Operators are interior nodes whose children are their operands.

4.4.1 Binary operators

The binary operators each have two children, the left being the first, and the right being the second.

(a) The field selector (`.`) behaves syntactically as a binary operator, but its right operand is a field selector, not an identifier, so change the right operand's token code to `TOK_FIELD`.

(b) The array indexing operator (`[ ]`) is syntactically a postfix matchfix operator. Use the left bracket as the operator and change its token code to `TOK_INDEX`. Discard the right bracket. The left child is the left operand and the right child is the expression between the brackets.

4.4.2 Unary operators

The unary operators have one child each, but since additon (`+`) and subtraction (`-`) are overloaded, at the time the parser has these operators adopt the child, The token codes should be changed to `TOK_POS` and `TOK_NEG`, respectively.

4.4.3 Variadic operators

Variadic operators take a variable number of children as arguments. They can take a variable number of arguments. The only one in `oc` is the `TOK_CALL` operator, whose first child is always an identifier, and whose later children are the arguments found between the parentheses of the argument list. Use the left parenthesis as the operator itself and change its token code to `TOK_CALL` before adopting the children. Discard the right parenthesis.

4.4.4 Allocator operator `new`

The allocator operator `new` is used to allocate a structure object or an array. Since constructors are not supported, `string` is treated as a special case.

(a) If the word following `new` is an `IDENT`, change the token code to `TYPEID`, and have `TOK_NEW` adopt it as its only child.

(b) In the case of allocating an array, change the token code to `TOK_NEWARRAY` as a binary operator and make the two children the `TYPEID` and the dimension expression.

(c) If a `string` is being allocated, change the token code to `TOK_NEWSTRING` as a unary operator with the expression as its child.

4.4.5 Grouping parentheses

Parentheses which are used to override precedence are discarded from the abstrct syntax tree, since the tree structure itself shows the order of evaluation. The AST for the parenthesized expression is just the expression. Both parentheses are discarded.

4.5 Statement Keywords as Operators

The statement keywords `while`, `if`, and `return` have "operands" which will be adopted as their children. These are not really operators in the sense of returning results, but syntactically they function in the same way. Blocks and variable declarations are also statements which must have some kind of operator at the root.

4.5.1 The `while` statement

The `while` statement always has two "operands" and so should adopt two children, the first one being the root of the expression, and the second being the root of the statement following it. Discard the parentheses around the expression.

4.5.2 The `if-else` statement

The `if` statement might have two or three children, depending on whether or not the optional `else` is present. The first child is the expression and the second child is the statement following it. Discard the parentheses around the expression.

(a) If an `else` is present, the `else` token is discarded and the `if` token adopts the statement following the else as the third child. Change the token code to `TOK_IFELSE` as a ternary operator.

(b) If there is no `else`, then the two children are the expression and the statement.

4.5.3 The `return` statement

The `return` keyword may or may not have an operand. It is marked optional in the grammar, because the parser has no way to determine whether or not an operand to `return` is required or prohibited. The latter decision is made by later semantic routines. The semicolon is discarded.

(a) If `return` has an expression, it is adopted as its child.

(b) If not, change the token code to `TOK_RETURNVOID`.

4.5.4 The block statement

A block is a series of statements enclosed between braces. The left brace has its token code changed to `TOK_BLOCK`, and adopts the roots of each of the interior statements. The right brace is discarded.

4.5.5 The semicolons statement

A semicolon by itself is just a vacuous block, and the semicolon itself is the AST. This also distinguishes a function definition from a prototype.

4.5.6 The expression statement

When an expression is used as a statement, the root of the expression becomes the root of the statement. The semicolon is discarded.

4.5.7 The variable declaration statement

A variable declaration has as its root the equal symbol which links the actual declaration with its initial value. The left child is an identifier declaration and the right child is the expression. To avoid confusion with the assignment operator, the token code of the equal symbol here is changed to `TOK_VARDECL`.

4.6 Function Definitions

The most complicated part of parsing global units is the function. It has a prototype and either a body or just a semicolon. The prototype consists of an identifier declaration and a parameter list.

It consists of a `TOK_FUNCTION` with three children: the identifier declaration, the parameter list, and the block. For the parameter list, change the open parenthesis token code to `TOK_PARAMLIST` and have it adopt each of the parameters. It will always be there but may possibly have no children. Like `TOK_ROOT`, the `TOK_FUNCTION` node can be spontaneously generated with the serial number of the first token in the function.

If instead of a block, there is only a semicolon, then the parent node in a function is a `TOK_PROTOTYPE`, and it has only two children.

4.7 Miscellaneous

A few miscellaneous ideas do not strictly belong to a unique one of the categories above.

(a) **Synthesized tokens:** There are three tokens that are not identified and returned by the scanner: `TOK_ROOT`, `TOK_FUNCTION`, and `TOK_PROTOTYPE`. Semantic actions that are part of the parser will create these tokens. Use the serial number `0.0.0` for `TOK_ROOT`, and the serial number from the first token of a function for `TOK_FUNCTION`, or prototype for `TOK_PROTOTYPE`. Be sure to insert a valid string table entry into the node as well. Use tokens distinct from what the scanner returns, such as `<<ROOT>>`, `<<FUNCTION>>`, and `<<PROTOTYPE>>`.

(b) **Identifier declarations:** Identifier declarations are part of a declaration, but otherwise can not exist on their own. They are always associated with an initialized variable declaration, a structure field, or a parameter list, or a function definition. The type itself can serve as a root, with the identifier being its child. In the case of an array, which is a generic data type, the array token `TOK_ARRAY` can be the root, with the base type and identifier being children. Note that `TOK_ARRAY`, lexical (`[]`) is a single token, different from the left (`[`) and and right (`]`) bracket tokens.

(c) **Substituted token codes:** In order to make the later phases of the compiler simpler and capable of having individual components chosen by a large `switch` statement, the parser substitutes a token code initialized by the scanner to something else.

(d) **Parser's token codes:** Following are the synthesized and substituted token codes described above which are introduced by the parser, with the sections wherein they were defined:

`TOK_ROOT` (4.1), `TOK_DECLID` (4.2), `TOK_TYPEID` (4.3), `TOK_FIELD` (4.4.1), `TOK_INDEX` (4.4.1), `TOK_POS` (4.4.2), `TOK_NEG` (4.4.2), `TOK_CALL` (4.4.3), `TOK_NEWARRAY` (4.4.4), `TOK_NEWSTRING` (4.4.4), `TOK_IFELSE` (4.5.2), `TOK_RETURNVOID` (4.5.3), `TOK_BLOCK` (4.5.4), `TOK_VARDECL` (4.5.7), `TOK_FUNCTION` (4.6), `TOK_PARAMLIST` (4.6), and `TOK_PROTOTYPE` (4.6).

5. Displaying the AST

After constructing an AST from a file called *program*`.oc`, write a file called *program*`.ast`, containing a representation of the AST in text form, printed using a depth-first pre-order traversal, showing depth via indententation.

Each line is indented to show its distance from the root, with a line upward immediately to its right that points at its parent. After the indentation print the symbolic token code, lexical information in double quotation marks, and the serial number consisting of a file number, a line number, and an offset. During project 4, this will also have type information following it.

All global statements, functions, and declarations will appear under the root, `TOK_ROOT`. If an include file is present, the contents of that file will also be printed under

```
 1      int fac (int n) {
 2         int f = 1;
 3         while (n > 1) {
 4            f = f * n;
 5            n = n - 1;
 6         }
 7         return f;
 8      }
 9      int n = 1;
10      while (n <= 5) {
11         puti (fac (n));
12      }
```

Figure 4. Example program

`TOK_ROOT` as well, with only the file numbers leaving a clue as to where the code originated. The parser has no direct idea about the names of the files that the tokens came from.

The program shown in Figure 4 would be printed as an AST as shown in Figure 5. To avoid clutter, the prefix **`TOK_`** has been omitted from the printed tree. The prefix was used in the C code to avoid possible name clashes. The following, using pointer arithmetic, can be used to eliminate the prefix:

```
char *tname = get_yytname (symbol);
if (strstr (tname, "TOK_") == tname) tname += 4;
```

6. The Parser

Start out with a **`parser.y`** which will generate a header file and a C source file. Develop it incrementally, possibly using routines from the example **`expr-smc`**, bearing in mind that that code does not exactly fit this project. Use the declarations shown in Figure 6 in section 1 of the grammar to configure your parser.

Your program will need to perform some syntax error checking and recovery, although not much sophistication is required. Use the code shown in Figure 7 at the beginning of your grammar section to set up the root of the AST and return it back to main. When there is a syntax error, an attempt to recover will be done by searching forward for a closing brace or a semicolon.

All other AST nodes are adopted by the root, which is a synthetically manufactured token (not created by the scanner). The parser needs a way to communicate with the main function, but has no communication results or parameters, so the global variable **`yyparse_astree`** will be used for that purpose.

All actions in the parser should be simple. Use function calls when that is not the case. Actions should be of one of the following two forms:

`{ $$ =` *simple expression*`;` **`}`**
`{ $$ =` *fncall (args...)*`;` **`}`**

```
<<ROOT>> "" 0.0.0
|  <<FUNCTION>> "" 0.1.0
|  |  INT "int" 0.1.0
|  |  |  DECLID "fac" 0.1.4
|  |  PARAM "(" 0.1.8
|  |  |  INT "int" 0.1.9
|  |  |  |  DECLID "n" 0.1.13
|  |  BLOCK "{" 0.1.15
|  |  |  VARDECL "=" 0.2.9
|  |  |  |  INT "int" 0.2.3
|  |  |  |  |  DECLID "f" 0.2.7
|  |  |  |  INTCON "1" 0.2.9
|  |  |  WHILE "while" 0.3.3
|  |  |  |  GT ">" 0.3.10
|  |  |  |  |  IDENT "n" 0.3.8
|  |  |  |  |  INTCON "1" 0.3.12
|  |  |  |  BLOCK "{" 0.3.14
|  |  |  |  |  '=' "=" 0.4.8
|  |  |  |  |  |  IDENT "f" 0.4.6
|  |  |  |  |  |  '*' "*" 0.4.12
|  |  |  |  |  |  |  IDENT "f" 0.4.10
|  |  |  |  |  |  |  IDENT "n" 0.4.12
|  |  |  |  |  '=' "=" 0.5.8
|  |  |  |  |  |  IDENT "n" 0.5.6
|  |  |  |  |  |  '-' "-" 0.5.12
|  |  |  |  |  |  |  IDENT "n" 0.5.10
|  |  |  |  |  |  |  INTCON "1" 0.5.12
|  |  |  RETURN "return" 0.7.3
|  |  |  |  IDENT "f" 0.7.10
|  VARDECL "=" 0.9.6
|  |  INT "int" 0.9.0
|  |  |  DECLID "n" 0.9.8
|  |  INTCON "1" 0.9.4
|  WHILE "while" 0.10.0
|  |  LE "<=" 0.10.9
|  |  |  IDENT "n" 0.10.7
|  |  |  INTCON "5" 0.10.12
|  |  BLOCK "{" 0.10.15
|  |  |  CALL "(" 0.11.3
|  |  |  |  IDENT "puti" 0.11.9
|  |  |  |  CALL "(" 0.11.13
|  |  |  |  |  IDENT "fac" 0.11.11
|  |  |  |  |  IDENT "n" 0.11.15
```

Figure 5. Example AST

```
%debug
%defines
%error-verbose
%token-table
%verbose

%start start
```

Figure 6. Parser configuration options

```
start    : program            { yyparse_astree = $1; }
         ;
program  : program structdef  { $$ = adopt1 ($1, $2); }
         | program function   { $$ = adopt1 ($1, $2); }
         | program statement  { $$ = adopt1 ($1, $2); }
         | program error '}'  { $$ = $1; }
         | program error ';'  { $$ = $1; }
         |                    { $$ = new_parseroot (); }
         ;
```

Figure 7. Beginning parser code

Neatly line up all of the nonterminals in the first column, all of the colon (`:`), alternation (`|`), and action (`{`) characters so that the grammar is easy to read. You should use **`valgrind`** to detect memory errors, although you can ignore memory leak. A good set of options to use when compiling C code is

```
gcc -g -O0 -Wall -Wextra -std=gnu99
```

However, you may prefer to avoid **`-Wall`** and **`-Wextra`** with code generated by **`flex`** and **`bison`**.