```
$Id: asg2-scanner.mm,v 1.10 2015-04-09 17:45:07-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments/
```

1. Overview

Augment your string table manager from the previous project by adding to it a scanner written in **flex**. Continue to use the module **auxlib**. Include token creation routines in the modules **astree**, **auxlib**, and **lyutils** from **Assignments/util-code**.

SYNOPSIS

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

For this project, given an input file called *program*.**oc**, you will generate output files called *program*.**str** (as before) and also *program*.**tok**. All specifications from project 1 apply to this project. In addition, the **-l** flag must turn on **yy_flex_debug**.

2. Tokens in the oc language

The oc language has the following tokens:

(a) Special symbols:

```
[] ( ) [ ] { } ; , . = == != < <= > >= + - * / % !
```

Single-character tokens may be represented by their ASCII values, but multiple-character tokens must be represented by a **bison**-generated set of names. Note the hack that `[]` is a single token, added to the language to eliminate a difficult shift/reduce conflict in project 3.

(b) Reserved words:

```
void bool char int string struct if else while return
false true null ord chr new
```

Reserved words may be just added to the scanner as patterns, but must precede recognition of identifiers.

(c) Identifiers are any sequence of upper- or lower-case ASCII (not Unicode) letters, digits, and underscores, but may not begin with a digit.

(d) Integer constants which consist of any sequence of decimal digits. Octal and hexadecimal constants are not supported. There is no floating point.

(e) Character constants consist of a pair of single quote marks with a single character or escape between them:

```
('([^\\'\n]|\\[\\'"0nt])')
```

(f) String constants consist of a pair of double quote marks with zero or more characters or escapes between them:

```
(\"([^\\"\n]|\\[\\'"0nt])*\")
```

Backslash, single quote, and newline may not appear in a character or string constants unless escaped.

(g) Comments and white space are consistent with the C preprocessor, which removes comments from the input stream. All C preprocessor statements are handled by **cpp**.

(h) Output directives from cpp of the form
 # *line* "*filename*"
must be scanned explicitly and used to indicate coordinates for printing error messages from source code.

(i) Also recognize invalid identifiers (beginning with a digit), and invalid character and string constants (missing a final quote or a character following an escape). Make sure the scanner report does not show any jamming states.

3. The scanner

Create a file `scanner.l` which is used to generate `yylex.cc`.

(a) The only C code that should appear in the `%{...%}` at the start of your scanner should be `#include` and `#define` preprocessor statements. In the first part of the scanner, use the following options :

```
%option 8bit
%option debug
%option nodefault
%option nounput
%option noyywrap
%option verbose
%option warn
```

(b) Retrofit your first project so that the external variable `FILE *yyin` is used to read the pipe from cpp. Every time `yylex()` is called, it reads from that external variable. Your main function will repeatedly call `yylex()` until it returnsa value of `YYEOF`.

(c) The file `misc-code/parser.y` contains a dummy parser which will not be called from this project, but which must be included so that the internal names of tokens can be printed. Copy that file and be sure that your `Makefile` uses it to build `yyparse.h` and `yyparse.cc`. The function `get_yytname`, given an integer symbol, will return a string representation of that symbol.

(d) In the parser provided, the first group of token definitions will be used by the scanner to return codes that are not represented by a single character. The second group of tokens are not recognized by the scanner, but are used in project 3 to edit the AST in order to prepare it for the later projects.

4. A sample compiler

Look in the directory
 `/afs/cats.ucsc.edu/courses/cmps104a-wm/Examples/e08.expr-smc`
for a sample compiler for a simple language. You will want to copy code from that directory, especially the modules `auxlib`, `astree`, and `lyutils`. Copy the `Makefile` as well and edit it as appropriate.

(a) Module `auxlib`, which you are already using for project 1 has several useful additions to the standard library, and macros for generating debugging information.

(b) Module **`astree`** has code useful for creating the abstract syntax tree, which you will need for this project, even though no AST will actually be assembled. The scanner creates ASTs for each token that it finds.

(c) Since an AST is a n-way tree with some nodes having an arbigrary number of children, it is easiest to represent the children by a C++ **`vector<astree*>`** field, for which **`push_back`** can be used to add a new rightmost child.

(d) Module **`lyutils`** contains useful declarations and functions for interfacing with code generated by **`flex`** and **`bison`**. Do not include C code (except function calls) in your scanner. Instead, make calls to functions in this module.

5. Output format

Your program will produce output similar to that shown in here :

```
# 16 "foobar.oc"
   2  16.003  264  TOK_KW_RETURN   (return)
   2  16.010   61  '='             (=)
   2  20.008  258  TOK_IDENT       (hello)
   2  20.010  271  TOK_LIT_INT     (1234)
   2  25.002  123  '{'             ({)
   2  26.008  272  TOK_LIT_STRING  ("beep")
```

(a) It models the information in the **`struct astree_rep`** constucted by the scanner, ignoring the pointers to other AST nodes, which have not yet been determined. Output will be printed to a file ending with the suffix **`.tok`**.

(b) Everytime a file directive is found, it is printed to the output token file, and also scanned to update the coordinate information.

(c) Each token is also printed to the output file in neatly aligned columns :
   (i) Index into filename vector, incremented for each **`#`**-directive.
   (ii) The line number within the given file where the token was found.
   (iii) The character offset of the first character of the token within that line.
   (iv) The integer token code stored in the AST node.
   (v) The name of the token as determined by **`get_yytname`**.
   (vi) The lexical information associated with the token.

6. Fragments of a `Makefile`

Some of the macro definitions in the **`Makefile`** might be :

```
LSOURCES  = scanner.l
YSOURCES  = parser.y
CLGEN     = yylex.cc
HYGEN     = yyparse.h
CYGEN     = yyparse.cc
LREPORT   = yylex.output
YREPORT   = yyparse.output
```

Then we may use **flex** and **bison** to build the scanner and parser with the following recipes :

```
${CLGEN} : ${LSOURCES}
        flex --outfile=${CLGEN} ${LSOURCES} 2>${LREPORT}
        - grep -v '^  ' ${LREPORT}

${CYGEN} ${HYGEN} : ${YSOURCES}
        bison --defines=${HYGEN} --output=${CYGEN} ${YSOURCES}
```

7. Some utility code

The subdirectory **code/utility-code/** contains some files that should be incorporated into your compiler.

(a) **astree.{h,cpp}** contains the definition and implemenetation of an n-way abstract syntax tree.

(b) **auxlib.{h,cpp}** are useful auxiliary functions.

(c) **lyutils.{h,cpp}** are definitions of functions and variables exported by **yylex** and **yyparse**, and some useful lexer functions. **stringset.{h,cpp}** is the string set from project 1.

(d) **yyparse.h** and **parser.y** is a dummy header file and dummy parser to be used with project 2 and discarded with project 3.

8. What to Submit

Submit **README**, **Makefile**, **scanner.l**, **parser.y** (for this project, the dummy parser), and all of the header and C++ implementation files. ***Do not*** submit the file generated by **flex**.