CS143 Handout #5
Autumn 2000

PP1: Lexical analysis[1]

Due midnight Mon Oct 9th
(can be submitted at most 3 days late)

The goal

In the first programming project, you will get your compiler off to a great start by implementing the lexical analysis phase, the first task of the front-end. You will use `lex` to create a scanner for the SOOP programming language. The scanner will read SOOP source programs and output a list of tokens and lexemes. In addition, the scanner will place all identifier names in a symbol table and keep track of how many times each identifier appears in the source.

This is a rather straightforward assignment and most students don't find it too time-consuming., however we are giving you a full week to work on it. Don't let that lull you into procrastinating on getting started. You will need to spend some time figuring out how to use lex and you should also plan for enough time to thoroughly test your scanner on lots of cases until you're sure you have all the bases covered.

Lexical structure of SOOP

Handout #4 gives you a taste of the SOOP language. It shares many similarities with C/C++/Java but keep in mind that not all features exactly match. In this first project, you are only concerned with being able to recognize and categorize the valid tokens from the input. Here is a summary of the token types in SOOP:

Identifiers: An identifier name can begin with an upper or lowercase letter or an underscore, and can be followed by upper or lowercase letters, digits, or underscores. An underscore alone is a valid identifier name.

Constants:

Integer Integer constants can either be specified in decimal (base 10) or hexadecimal (base 16). For decimal numbers, the valid digits are `0-9`. Hexadecimal adds `a` through `f` (both upper and lowercase are acceptable). A decimal integer is a sequence of decimal digits. A hexadecimal integer must begin with `0X` or `0x` (that is a zero, not an oh) and is followed by a sequence of hexadecimal digits. Examples of valid integers: `8, 012, 0x0, 0X12aE`

Double Doubles are any sequence of digits, a period, followed by any sequence of digits, maybe none. Thus, `.12` is not a valid double but both `0.12` and `12.` are valid. A double can also have an optional exponent, e.g., `12.2E+2` For a double in this sort of scientific notation, the decimal point is required, the sign of the exponent is required, and the E can be lower or upper case. As above, `.12E+2` is invalid, but `12.E+2` is valid. Leading zeroes on the mantissa and exponent are allowed.

String Strings are enclosed with double quotes and can contain any character except a double quote (there is no escape character to allow an embedded double-quote). Strings cannot be split over several lines, e.g.,

```
"this is not a
  valid string constant"
```

[1] *Maggie Johnson and Steve Freund authored the original 143 projects that ours are based on.*

Boolean The reserved words `true` and `false` are used as boolean constants.

Operators and other punctuation:

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

Two-char `&& || <= >= == !=`

Comments:

Single-line
```
// from here to the end of the line is a comment
```

Multi-line
```
/* everything inside here is a comment
   and it can span multiple lines */
/* Unlike C, SOOP comments can
   /* nest /* inside */ */
one another */
```

Note that the following are comments: `/*//// */` and `/******/`. Basically, any symbol is allowed in a comment except the sequence `*/` which always marks the end of the comment or the sequence `/*` which starts a nested comment within.

Reserved words: The following words are reserved words in SOOP. Like C and Java, SOOP is case-sensitive.

```
void  int  double  bool  class
while  if  else  return  public  private
New  NewArray  Print  ReadInteger
true  false  NULL
```

Starter files

We have a directory on leland **/usr/class/cs143/** where we will place course materials. The starting files for this project are in the subdirectory **assignments/pp1**. You can access them directly on the leland filesystem or from the class Web site **http://www.stanford.edu/class/cs143/**.

Various support files will be provided to you for each programming project. The first thing to do is review all of the given files. We provide helpful functions and type and constant definitions that you should use rather than re-implement. Always, always, always read through the support files first to understand what we give you and save yourself time and effort. The files should be commented with reasonable information about the implementation and use of the provided features. If you have questions, be sure to ask!

The starting pp1 project contains the following files (the boldface entries are the ones you will need to modify):

`Makefile`	builds project
`main.cc`	`main()` and some helper functions
`soop.h`	defines prototypes for scanner functions
`soop.l`	skeleton of a lex program
`tokens.h`	defines token constants and types
`declaration.h/.cc`	interface/implementation for Declaration class
`utility.h/.cc`	interface/implementation of various utility functions
`samples/`	directory of test input files

Copy the entire directory to your home directory. Run `make depend` to set up dependency information and then use `make` to build the project. The makefile provided will produce a lexical

analyzer called `pp1`. It reads input from stdin and you can use standard UNIX file redirection to read from a file:

```
% pp1 < samples/program.soop
```

(You will need to specify the full path as `./pp1` if you don't have . in your path.)

You should **not** modify `tokens.h`, `soop.h`, or `main.cc`, since our grading scripts depend on your output matching our defined constants and behavior. You can (but are not likely to need to) modify `utility.h/.cc`. You will definitely need to modify `declaration.h/.cc` and `soop.l`.

We provide starter code that will compile but is very incomplete. It will not run properly if given sample files as input, so don't bother trying to run it against these files as given.

Symbol table implementation

To handle identifiers, you will need two things: an object for managing a declaration and a table to map identifiers to a declaration. For the first, you are to complete the Declaration class. For the second, you must write a hashtable that allows for initialization, insertion and lookup. You may re-use a hashtable you implemented for a previous course or project, but you must have written the code yourself (i.e. your hashtable from 107 would be fine).

The hashtable can be implemented as a set of C functions, an ADT, a C++ class, a template C++ class, whatever you like. Do not worry about dynamically resizing your hashtable or anything fancy. Any reasonably efficient implementation will do. You should put your hashtable implementation in its own file. You will need to modify the makefile to include that file in the list of `SRCS` compiled into the executable.

Scanner implementation

The `soop.l` lex input file in the starter project contains a skeleton you must complete. The `yylval` global variable declared in `tokens.h` is designed to be filled for each token scanned. The action for each pattern will assign any necessary fields in the global variable and then return the appropriate token value. Your goal is to modify `soop.l` so the scanner will:

- skip over white space
- skip over all comments (including nested ones)
- recognize all reserved words and return the correct value from `tokens.h`
- recognize punctuation and single-char operators and return the ASCII value for the symbol
- recognize two-character operators and return the correct token value
- recognize int, double, and bool constants, return the correct code and set appropriate field of `yylval`
- recognize string constants, return the correct code and set appropriate field of `yylval`
- recognize identifiers, return the correct code and set appropriate fields of `yylval`. In addition, if the identifier has not been seen before, create a declaration record in the symbol table; if already seen, increment a count for that declaration.
- print specific errors for strings with embedded newlines and unterminated comments and provide a generic error message for any other invalid character in the input

We recommend adding token types one at a time to `soop.l`, testing after each addition. Do the reserved words first since they are simple. Next add the single-character operators as one big character class (a single pattern) and entries for the two-character operators. Be careful with punctuation that has special meaning such as `*` and `-` (see lex manual for how/when to suppress

special-ness). Next work out the patterns for integers, doubles, and strings. These require some careful testing to make sure you have all the cases covered. Add patterns for the identifiers and handling of the symbol table of declarations to update and report the seen counts.

Now add handling for comments. Single-line comments are straightforward, but multi-line and nested comments are a little tricky. Even without the nesting, a regular expression that correctly matches all multi-line comments is difficult to write and get entirely correct (try it!). But furthermore, balanced-parentheses-type languages are not regular (you should be able to prove this), so you cannot write a regular expression to match a comment nested to an arbitrary depth. There are various ways to track the comment state and nesting depth yourself. You will probably need a global variable or two. Lex states can come in handy here. One particularly tidy approach is to use the state stack features of flex and you won't need any globals at all. Read the online manual (links in Other Materials of course web site) for information about lex states and the state stack.

Lastly you need to be sure that your scanner correctly reports errors encountered in the input. The action for an error case should call the `yyerror()` function (provided by us as a utility, it needs to be named this for the parser to find it) passing an informative message. If the action does not have a return statement, the effect is to ignore the just-scanned input and continue on from there.

For a newline inside a string constant, the appropriate message is `Illegal newline in string constant "blah blah"`. If the file ends with an open comment, the message should be `Input ends with unterminated comment`. (See the flex manual for info on the `<<EOF>>` pattern that can be helpful here.) For each character that cannot be matched to any token pattern, the message is `Unrecognized char: '~'`. Your scanner should not halt on errors, it discards the problem input, and continues, either reporting more errors for additional incorrect input or recognizing correct tokens that follow.

You should be able to recognize everything you need using regular expressions and the "lookahead" feature of lex. You should not use `yyless`, `yymore`, `input`, `unput`, or reprocess the string in `yytext`. You may add any necessary initialization steps to the `Inityylex()` function that is called once before any scanning at the start of execution.

Keep in mind that lex is not all that user-friendly. For example, if you put a space or newline in the wrong place, it will often print "syntax error" with no line number or hint of what the true problem is. It may take some delving into the manual, a little experimentation, and some patience to learn its quirks. Here are a few suggestions to help keep you sane:

- Be careful about spaces within patterns (it's easy to accidentally allow a space to be interpreted as part of the pattern or signal the end of pattern prematurely if you aren't attenttive).
- Never put newlines between a pattern and an action.
- When in doubt, parenthesize with the pattern to ensure you are getting the precedence you intend.
- Enclose each action in curly braces (although not required for a single-line action, better safe than sorry).
- Use the definitions section to define pattern substitutions (names like Digit, StartComment, etc.). It makes for much more readable rules that are easier to modify, build upon, and debug.
- Always put parens around the body of a definition to ensure the correct precedence is maintained when it is substituted.
- You must put curly braces around the definition name when you are using it in another definition or a pattern, without them it will only match the literal name.

Testing your scanner

In the starting project, there is a `samples` directory containing various input files and some ".out" files which represent the expected output. This output will give you a good idea of how your scanner should behave. Be sure to look through these files since they provide helpful clues.

However, be careful, the provided test files do **not** test every possible case! Examine the test files and think about what cases aren't covered. Construct some of your own input files to further test your scanner. What formations look like numbers but aren't? What sequences might confuse your processing of comments or string constants?. This is exactly the sort of thought-process a compiler writer must go through. Any sort of input is fair game to be fed to a compiler and you'll want to be sure yours can handle anything that comes its way, correctly tokenizing it if possible or reporting some reasonable error if not.

Remember that lexical analysis is only responsible for correctly breaking up the input stream and correctly categorizing each token by type. The scanner will accept semantically-incorrect sequences such as:

```
int if stdio + 4.5 [bool}
```

General requirements

Some ground rules that describe our expectations regarding all projects:

- Your source files should be easily readable on UNIX— i.e. end-of-line characters should be proper, lines shouldn't wrap in obnoxious places, etc. This is of particular importance to those of you moving your files to Solaris from elsewhere.
- The submitted project should include all necessary files. We should be able to issue the command `make` and all files should compile cleanly (i.e. with no warnings).
- Your executable should be named `pp1` and should require no command-line arguments. It should process input from stdin. For example, we should be able to run your scanner with the command `pp1 <samples/program.soop`.
- You will probably need a global variable or two, but think carefully before promoting a variable to global scope and be sure it is truly necessary.
- Compilers are notorious for leaking memory. Given that they run once and exit, this is not considered much of a problem. We will not examine your programs for leaks and not expect them to free dynamically allocated memory.
- We have included a Purify target in the starting makefile. Build using `make pure` and then run the result as `pp1.purify`. By default, it will place its output in the text file `purify.log`. You can learn all about Purify from its man page. It's a great tool for debugging memory errors if you happen to make any. We only included this target for your use, we will not run your program under purify during grading and as noted above, you do not have to worry about leaks.

Grading

This project is worth 70 points. All 70 points are for correctness. Be aware your projects in this course will be graded primarily on their output. We will run your program through the given test files from the samples directory as well as several others of our own. We will compare (`diff`) your output to the output of our solution. Your best bet is to thoroughly test your program on the provided test files and make up others of your own until you feel confident that your program can handle all situations. For this project, we will also look at your hashtable code. You will receive full credit for a moderately efficient, reasonably implemented hashtable.

Note that no specific points are allocated for style or design. It is not that we consider these unimportant, but by now all of you have taken a course like 107 and we hope you have internalized and practice good coding practices as a matter of habit. Thoughtful design and coding will improve your chances of having a correctly working program, too. Also note that we will not be poking around trying to fix broken submissions that don't compile or seg fault on our tests, only limited partial credit will be given in such situations.

Portability

Our files are designed for the versions of the `make`, `g++`, `flex` and `bison` tools available on our Solaris workstations. You can try to move the project to another platform, but be aware it may take tweaking to get it working. If you run into problems, you will need to move back to our supported environment.

No matter where you do your development work, when done, you must be sure your project will compile and run on the leland workstations. All assignments will be electronically submitted there and that is where we will compile and test your project.

Electronic submission

You will not submit paper printouts, but instead use a submit script to electronically deliver your entire project directory. All students (SITN, local, remote, or otherwise) use the same process to submit assignments. The submit instructions are detailed on web site at **http://www.stanford.edu/class/cs143/policies/**.

Some advice— leave enough time before the due time to run the submit script and deal with any submission problems. This goes triple for those doing your work on another platform. You will need time to move the files over, recompile them on Solaris, re-test the code, remove object files, and submit. You may want to do a trial run of compiling and submitting on Solaris in advance to become familiar with the process and avoid last minute panic.

Late days

Refer to handout #1 or the web site policy page for the course policy on late work. If you are choosing to use one of your self-granted extension days, you do not need to confirm with us, just submit your work using the same process and it will be time-stamped accordingly. We will not penalize for use of up to three self-granted extension days across the quarter, but past that, late work is discounted approximately 20% per 24-hour day late. No projects will be accepted more than three days after the due date (may be shorter for some projects).

Partners

You are encouraged to work in pairs on the programming projects. Although working with a partner won't exactly cut the work in half, it will provide some relief and companionship during the process and we believe you might both come away with a better understanding of the material. A win-win situation!

You can work with only one partner on any one assignment, but you can switch partners between assignments or do some assignments with a partner and others alone. If you work with a partner, only one of you should e-submit the project, but include information about both partners in the README file.

Honor code

Programming projects must be implemented "from scratch". It is forbidden to derive solutions from existing code, especially from previous instances of this course. A failure to abide by this condition constitutes a violation of the Stanford Honor Code. If you are re-using your own work from a previous unsuccessful attempt at the course, please note so in your README.

Although you may want to occasionally talk through general issues with others in the class, we expect that you will not discuss your work in detail or be sharing your code with others or reading other's code.

If you are in doubt about how our policies might apply in a situation, please ask. Dishonorable behavior hurts all students, not just the one making the wrong choice, and it is immensely disappointing to me in a personal sense. Our staff is here as a resource for you and our goal is to help you succeed on your own with integrity and we want that to be your goal as well.

"We have enough youth, how about a fountain of smart?" —seen on a bumper recently