

Programming Project 1: Lexer

Handout written by Julie Zelenski. Updated by David Underhill.

The Goal

In the first programming project, you will get your compiler off to a great start by implementing the lexical analyzer. For the first task of the front-end, you will use `flex` to create a scanner for the Decaf programming language. Your scanner will run through the source program, recognizing Decaf tokens in the order in which they are read, until end-of-file (EOF) is reached. For each token, your scanner will set its attributes appropriately (this will eventually be used by other components of your compiler) so that information about each token will be properly printed.

This is a fairly straightforward assignment and most students don't find it too time-consuming. Most of the work comes in figuring out how `flex` works.

Due: Wed 02 July 2008 at 4:00 p.m.

Decaf: Lexical Structure

Here is a summary of the token types in Decaf.

The following are keywords. They are all reserved:

```
void int double bool string class interface null this  
extends implements for while if else return break New NewArray
```

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Decaf is case-sensitive, e.g., `if` is a keyword, but `IF` is an identifier; `binky` and `Binky` are two distinct identifiers. Identifiers can be at most 31 characters long.

Whitespace (i.e. spaces, tabs, and newlines) serves to separate tokens, but it's otherwise ignored. Keywords and identifiers must be separated by whitespace, or by a token that is neither a keyword nor an identifier. `ifintthis` is a single identifier, not three keywords. `if(23this` scans as four tokens.

Decaf adopts the two types of comments available in C++. A single-line comment is started by `//` and extends to the end of the line. Multi-line comments start with `/*` and end with the first subsequent `*/`. Any symbol is allowed in a comment except the sequence `*/` which ends the current comment. Multi-line comments do not nest. Your scanner should consume any comments from the input stream and ignore them. If a file ends with an unterminated comment, the scanner should report an error.

A Boolean constant is either `true` or `false`.

An integer constant can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits (0-9). A hexadecimal integer must begin with `0x` or `0X` (that is a zero, not the letter oh) and is followed by a sequence of hexadecimal digits. Hexadecimal digits include the decimal digits and the letters `a` through `f` (either upper or lowercase). Examples of valid integers: `8`, `012`, `0x0`, `0X12aE`. Type `'man strtol'` for information about how to convert decimal and hexadecimal strings to `longs`, and note the minor difference between what we want and what we get with `strtol`. You may assume integers can be represented with 32 bits.

A double constant is a sequence of digits, a period, followed by any sequence of digits, maybe none. Thus, `.12` is not a valid double but both `12.` and `0.12` are. 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 optional (if not specified, `+` is assumed), 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. You can assume that it is safe to use `strtod` to convert double constants.

A string constant is a sequence of characters enclosed in double quotes. Strings can contain any character except a newline or double quote. String constants do not allow C-style escape sequences. For instance, `"\"` is a perfectly valid string in Decaf, even though it would be an open string constant in C or C++. A string must start and end on a single line, it cannot be split over multiple lines:

```
"this string is missing its close quote
  this is not a part of the string above
```

Operators and punctuation characters used by the language includes:

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

Note that `[`, `]`, and `[]` are three different tokens and that for the `[]` operator, as well as the other two-character operators, there must not be any space between the characters.

Using flex

Inspect the CS143 web site. All the way to the right you'll find links to various `flex` resources. We will not cover all the details of the tools in lecture, so you will need to read the online docs to learn what you need to know.

`flex` is not exactly user-friendly. 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 shortcomings. 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 if you aren't careful).
- Never put newlines between a pattern and an action.
- When in doubt, parenthesize within 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`, `Exponent`, etc.). It makes for much more readable rules that are easier to modify, extend, and debug.
- Always put parentheses around the body of a definition to ensure the correct precedence is maintained when it is substituted.
- You must put curly braces around a definition name when using it in another definition or a pattern; without them it will only match the literal name.

Starter files

The starting files can be checked out with `svn` by issuing the following command:

```
from myth: svn co file:///usr/class/cs143/assignments/svnrepos/pp_base
```

Or remotely: `svn co svn+ssh://USER@myth.stanford.edu/usr/class/cs143/assignments/svnrepos/pp_base`

If you have permissions issues, please contact the course staff.

The starting `pp1` project has the following structure

<code>Makefile</code>	rules to build, test, and submit the scanner
<code>Makefile.common</code>	definitions of tools, flags, libraries, and other variables
<code>README</code>	fill in the information about your project
<code>scripts/</code>	scripts you won't need to modify or directly use (test aids)
<code>src/</code>	directory of source code for the project
<code>debug.h/.c</code>	simple debug printing and checking methods
<code>errors.h/.cpp</code>	error messages you are to use
<code>list.h</code>	<code>List</code> implementation with C++ templates
<code>location.h</code>	defines some helper functions for working with <code>yy1loc</code>
<code>main.cpp</code>	initializes the scanner and generates the output requested
<code>Makefile</code>	rules to build the scanner
<code>manual_token.h/.c</code>	defines token values for you to use (yacc will later generate these); also provides methods for printing tokens
<code>options.h/.cpp</code>	parses command-line args and sets options appropriately (default [print tokens] is what we want for now)
<code>scanner.h</code>	prototype declarations for scanner
<u><code>scanner.l</code></u>	starting scanner skeleton
<code>tests/</code>	directory for test framework
<code>private_self_tests</code>	put tests you create for your own team here
<code>samples-pp1</code>	sample tests for <code>pp1</code>

`shared_student_tests` tests contributed by students and shared by all
 Your first order of business is to read through all the files to learn the lay of the land as well as absorb the helpful hints contained in the files. You probably will not need to modify anything except `scanner.1`.

We have provided you with a sample `Makefile` as a start to build the project. The `Makefile` has a target to build the `dcc` executable (just type “make” or “make dcc”). `dcc` must read input from `stdin`; therefore, you can use standard UNIX file redirection to read from a file. For example, to invoke your compiler (well, your scanner) on a particular input file, you would use:

```
% dcc < sample.decaf
```

Scanner implementation

The `scanner.1` lex input file in the starter project is where you’ll do your work. The `yy1val` global variable is used to record the value for each lexeme scanned and the `yy1loc` global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code. Your goal is to modify `scanner.1` to:

- skip over white space
- recognize all keywords and return the correct token from `manual_token.h`
- recognize punctuation and single-char operators and return the ASCII value as the token
- recognize two-character operators and return the correct token
- recognize `int`, `double`, `bool`, and `string` constants, return the correct token and set appropriate field of `yy1val`
- recognize identifiers, return the correct token and set appropriate fields of `yy1val`
- record the line number and first and last column in `yy1loc` for all tokens
- report lexical errors for improper strings, lengthy identifiers, and invalid characters

We recommend adding token types one at a time to `scanner.1`, testing after each addition. Be careful with characters that have special meaning to `lex` such as `*` and `-` (see docs for how/when to suppress special-ness). The patterns for integers, doubles, and strings will require careful testing to make sure all cases are covered (see man page for `strtol/strtod` for converting strings to numbers).

Recording the position of each lexeme requires you to track the current line and column numbers (you will need global variables) and update them as the scanner reads the file, mostly likely incrementing the line count on each newline and the column on each token. A tab character always accounts for exactly 8 columns – there is no notion of tab stops. There is code in the starter file that installs a function to be automatically included with each action (that’s much nicer than repeating the call everywhere!)

Lastly you need to be sure that your scanner reports the various lexical errors. The action for an error case should call our `ReportError` class with one of the standard error messages provided in `errors.h`. For each character that cannot be matched to any token pattern, report it and continue parsing with the next character. If a string erroneously contains a `newline`, report an error and continue at the beginning of the next line. If an identifier is longer than the Decaf maximum, then report the error, truncate the identifier to the maximum number of characters (discarding the rest), and continue.

Take care that comment characters inside string literals don't get misidentified as comments. If a file ends with an unclosed multi-line comment, an error is reported via a call to one of the methods in the `ReportError` class. In order to match our output exactly, please use the standard error messages provided in `errors.h`.

Testing your work

In the starting project, there is a `tests` directory containing various input files and matching `.out` files which represent the expected output. You should `diff` your output against ours as a first step in testing. Now examine the test files and think about what cases aren't covered. Construct some of your own input files to test your scanner even more. What lexemes look like numbers but aren't? What sequences might confuse your processing of comments? This is precisely the sort of thought process a compiler writer must go through. Any sort of input is fair game 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.

Note that lexical analysis is responsible only for correctly breaking up the input stream and categorizing each token by type. The scanner will accept syntactically bogus sequences such as: `int if array + 4.5 [bool}`

Testing framework

We also provide you with a simple test framework in the `tests` folder. **First, get the tests by going to the `tests` folder and type "make up". If you're not running on a `leland/myth` machine, then instead do "make co-ssh" to get the tests over the Internet via SSH.** To run your scanner against all the tests, type "make test" from the root folder, or just "make" from the `tests` folder. This executes your scanner on every test's input and check that it matches the expected output. When a test fails, you can go to its folder and look at the `.diff` file to get an idea of what went wrong. Initially, the provided scanner source just echoes all input, so it will fail everything.

The tests we provide are only a sample of what we will test your code with - we have many more tests which we will use to grade your submission. We encourage you to create your own tests - you are permitted to share your test cases with other groups too! To encourage you work together, we've tried to make it easy to share your tests. The

`tests/shared_student_tests` folder is a SVN repository which *everyone* can modify. To get the latest tests, go to the `tests` folder and type “make up”. To add a test:

1. Create a new subfolder - its name is the name of your test (ex: “badbool”).
2. Create your test in the subfolder with the same name of your subfolder. The extension should be `.decaf` if the code has no errors, or `.frag` if the compiler should detect errors.
3. Create the reference output file with the name test, followed by the test type (“token” for now since we’re just testing the lexer), and then `.ref`. (ex: “badbool.token.ref”).
 - Note: If your scanner is working, you can type “make test-init” to have the reference file generated from your input file automatically.
4. Place a copy of the Makefile from the example test we provided (“ChessMeister”) in your subfolder. Open the new Makefile and update the variables `TEST` and `IN_EXT` with the name of your test and the input file’s extension, respectively.
5. Copy the `README` file from the example test and adapt it to your test.
6. Add your test to the list of tests in `tests/shared_student_tests/Makefile`.
7. All done! Now try it out and once you’re satisfied add the test to the repository so others can use it too.

The root Makefile is also capable of generating coverage data for your project with “make cov”. This may be more useful in later projects, but it will generate a series of web pages in a subfolder named “lcov-html” which will graphically show you which lines were hit by the tests. This might give you some insight into what other aspects of your code might need to be tested. In order to use this, you will need to have the `lcov` tool (located in `/usr/class/cs143/bin`) in your `PATH` variable. You can add it to your path in `tcsh` with this command: “`setenv PATH /usr/class/cs143/bin:$PATH`” or in `bash` or `sh` with this command: “`export PATH=/usr/class/cs143/bin:$PATH`”.

Grading

Most of the grade will be allocated for correctness (85%). There will also be some consideration (15%) for design and implementation (see the “Coding Guidelines” page on the website). We will run your program with the given tests as well as others of our own. We will use `diff -w` to compare your output to the reference solutions.

Deliverables

You are to electronically submit your entire project using an electronic submission process (see the “Submission Instructions” page on the website). Be sure to include your `README` file, which is your chance to explain your design decisions, how you handled tricky cases, what interesting problems you encountered, and any feedback you might have for us.