

Programming Project I

CS 322 Compiler Construction
Winter Quarter 2006

Due: Friday January 13 at 11:59pm

Description

You will create a lexical analyzer (scanner) for a version of the language Pascal as described below. Your lexical analyzer will scan an input sequence of characters and convert it into a stream of tokens. To specify the tokens you will use FLEX. Read the manual that has been provided. There are many useful examples in it.

In addition, you will implement part of the symbol table where all identifiers are stored.

IMPORTANT: Read the whole handout before you start writing code.

Tokens

The tokens to be recognized are:

Reserved words:

| | | | | | | |
|--------|---------|----------|------------|----------|---------|---------|
| TARRAY | TBEGIN | TCASE | TCONST | TDO | TDOWNTO | TELSE |
| TEND | TFOR | TFORWARD | TFUNCTION | TGOTO | TIF | TLABEL |
| TNIL | TNOT | TOF | TPROCEDURE | TPROGRAM | TRECORD | TREPEAT |
| TTHEN | TTO | TTYPE | TUNTIL | TVAR | TWHILE | TNEW |
| TREAD | TREADLN | TWRITE | TWRITELN | TTRUE | TFALSE | |

Operators

| | | | |
|---------|---------------------|---------|-----------------|
| TRELOP | <, >, <=, >=, =, <> | TADDOP | +, -, <i>or</i> |
| TMULOP | *, /, <i>and</i> | TSEMI | ; |
| TCOLON | : | TASGN | := |
| TCARET | ^ | TDOT | . |
| TDOTDOT | .. | TCOMMA | , |
| TLPAREN | (| TRPAREN |) |
| TLBRACK | [| TRBRACK |] |

Constants

| | |
|---------|--|
| TUINT | Unsigned integer |
| TUREAL | Unsigned real (with optional exponent) |
| TSTRING | Double-quoted string |

other

TIDENT Identifier
TERROR erroneous symbol, none of the above

The symbol table

The symbol table is a dictionary structure that associates *attributes* with *names*. It is used to keep track of all identifiers along with information such as their scope, type, parameter types (for function names) etc.

The symbol table must provide the following functions:

- **Find**: look up an identifier in the symbol table to see if it has been declared.
- **Insert**: insert an identifier in the symbol table.
- **Delete**: when an identifier goes out of scope, delete all associated information. This will be implemented at a later stage.

Our symbol table is implemented as a hash table with chaining. There is exactly one symbol table entry for each identifier.

All the identifiers and all the strings that are encountered in the input are stored in `char * lexemeArray`. Each symbol table entry contains the index of the location in the `lexemeArray` where the corresponding identifier is stored.

Look in `syntab.h` for the definition of the symbol table class. You will need to implement the following functions:

- **Find**
Given a lexeme, search the symbol table to see if it has already been stored. If so, return its index in the `lexemeArray`, otherwise return -1.
- **Insert**
Given a lexeme, look for it in the symbol table. If it is not there, create a new symbol table entry for it and add the lexeme to the `lexemeArray`. If inserting the element will cause the load factor to exceed 1.5, then rehash. This function returns the index in the `lexemes` array where the lexeme begins.
- **Rehash**
Doubles the size of the hash table and rehashes all elements.

Recognizing comments

Comments in Pascal are delimited by (`*` and `*`) and may span several lines. A `*` may not appear inside a comment. The shortest possible comment is `(**)`. Your scanner should recognize comments and ignore them. In addition, include pattern-action pair(s) to print the error message

End of file encountered while still in comment.

if a comment is not terminated prior to end-of-file.

Hint: Use start conditions in conjunction with `<<EOF>>`

Recognizing strings

The lexeme for a double-quoted string constant should be entered into the lexemes array. You should store only the string itself without the surrounding quotes.

Warning: Do not do this by manipulating the buffer pointed to by *yytext*. Read the section titled “How the input is matched” (in the flex manual) and in particular the part about *yytext* as an array vs. a pointer. In this assignment, *yytext* is a pointer.

Strings are delimited by double quotes and may include any printable character except a newline. Two consecutive double quotes are used to include a double quote character in the string. For example, “abc%” corresponds to the string abc% and “ab””cd””””” corresponds to the string ab”cd”

Your scanner must be able to handle erroneous double quoted strings (i.e. strings which fail to have a terminating double quote prior to end-of-line). For such a string, your scanner should print the error message:

```
Missing double quote (") supplied.
```

and then should behave as if the string were correctly terminated. In other words, it should handle the string as if there had been a terminating double-quote just prior to the newline, and should return the token TSTRING.

The maximum string length allowed in Pascal is 255 characters. Your scanner should check for that and if it finds a longer string, it should print out the error message:

```
String is longer than 255 characters. Extra characters discarded.
```

Then, it should behave as if the string was the correct size (extra characters should be ignored).

As mentioned before, you should ensure that *yytext* is intact upon return from *yylex*.

Hint: Use start conditions.

Recognizing identifiers and numbers

The lexeme for TIDENT should be placed in the Symbol Table and *yylval* should be set equal to the appropriate index into the lexemes array. Then, your scanner should return the token TIDENT. TIDENTs are considered case sensitive, so you need not worry about converting upper case to/from lower case.

The lexeme for a TUINT should be gathered and the value of the unsigned integer should be placed in *yylval*. Your scanner should then return the token TUINT. You need not insert the lexeme in any table.

Reserved words are not treated as special identifiers which are entered into the symbol table, but must be recognized separately by your scanner.

Our compiler will not implement type `real`, but your scanner should recognize real numbers, set *yylval* = 0 and return the token TREAL. Reals are defined as follows:

```
<real> -> <int> <exponent> | <int>.<int> <exponent> | <int>.<int>  
<exponent> -> E+<int> | E-<int> | E <int>
```

Recognizing operators

For the lexemes `<`, `>`, `=`, `<=`, `>=`, `<>`, set *yylval* to an appropriate value to distinguish the actual lexemes, (i.e. VLT, VGT, VEQ, VLE, VGE, VNE), and return the token TRELOP.

Similarly, return TADDOP for `+`, `-` and *or*, and TMULOP for `*`, `/` and *and* and set *yylval* to an appropriate value (see `globals.h`) to distinguish actual lexemes.

Output

Your scanner should skip whitespace. It must echo the input with line numbers as it is read, including comments. Look at the test cases (`test.?.std`) to see what we expect.

HINT: Avoid putting couts or ECHO in every action. Instead, any debugging output should be printed by `scan()` (see `flex.l`). For error reporting, use `yyerror()`. Error messages should be sent to standard output, not `stderr`.

IMPORTANT NOTE

The purpose of this assignment is to build a scanner using the capabilities of FLEX. Therefore, something like the following is not acceptable:

```
"(*)"      { C++ code using get() to read characters until a *) is found ... }
```

Instead use appropriate flex patterns and tools like `yymore`.

Files

The following files are provided for you :

- `syntab.h` : interface to the symbol table class.
- `syntab.cpp` : implementation of the symbol table class. You will need to modify it.
- `globals.h` : constants, `#includes`, etc. Do not modify it.
- `tokennums.h` : values of various tokens. Do not modify it.
- `flex.l` : a “skeleton” flex file that you should use as a starting point to create your flex file.

A Makefile has also been provided. The provided files compile but the resulting scanner doesn't do anything.

Type `make` to compile your code.

Type `make test.?.s` to run your program on test case `test.?` where `?` is a number between 1 and 8. Your output will be in file `test.?.s`.

Type `make tests` to run your program on all test cases.

Test cases

Test files are also provided. The expected output for test file `test.?` is shown in `test.?.std`. We will be using `diff` to compare your output to ours. Matching the test cases does not guarantee full credit. Make sure you submit well-designed code.

Submitting your code

Submit `flex.l`, `syntab.h` and `syntab.cpp` ONLY. Email them to

`c22@cs.nwu.edu`

Include your name and email in a comment at the beginning of the files you are submitting.

If you want to submit an updated version of your program, just email it again with subject RESUBMIT.

We are not responsible for mistakes in emailing the assignment. It is a good idea to cc the email to yourself to verify that the code was attached correctly.

Getting help

You may post questions (but NOT code) to the newsgroup and discuss the assignment with other students. Check the newsgroup often for tips and hints. You may NOT share code with anyone.

Grading

Your code will be graded for correctness and style. Follow the guidelines posted above. Also, keep in mind that items such as recognition of strings will be worth more than, say, recognition of integer literals.