CS 280
Programming Language Concepts

About Assignment 2

Notes for Assignment 2

- Be sure to read and understand the assignment!
- Make a list of the information that you will need to keep track of in order to do the assignment
  - How should you save the data?
- What algorithm should you use for lexical analyzer?

Outline

- Set up to run (check arguments, open files, etc)
- Repeatedly call getNextToken
  - Print trace if –v is enabled
  - Keep statistics on results
- Print results
- Write the pseudocode for this!

Tokens to Recognize

- The language has identifiers, which are defined to be a letter followed by zero or more letters or numbers. This will be the Token IDENT.
- The language has integer constants, which are defined to be one or more digits with an optional leading sign. This will be the Token ICONST.
- The language has string constants, which are a double-quoted sequence of characters, all on the same line. This will be the Token SCONST.
- A string constant can include escape sequences: a backslash followed by a character. The sequence \n should be interpreted as a newline. The sequence \\ should be interpreted as a backslash. All other escapes should simply be interpreted as the character after the backslash.
- The language has reserved the keywords print, println, repeat, begin, end. They will be Tokens PRINT PRINTLN REPEAT BEGIN END.
- The language has several operators. They are + - * = / ( ) which will be Tokens PLUS MINUS STAR SLASH EQ LPAREN RPAREN
- The language recognizes a semicolon as the token SC
- A comment is all characters from // to the end of the line; it is ignored and is not returned as a token. NOTE that a // in the middle of a SCONST is NOT a comment!
- Whitespace between tokens can be used for readability, and can serve as one way to delimit tokens.
- An error will be denoted by the ERR token.
- End of file will be denoted by the DONE token.

getNextToken questions

- What patterns need to be recognized?
- Is there a different approach for multi-character tokens and single character tokens?
- What are the states?
- How do I represent states?

Possible state list

- INIDENT, INSTRING, ININT, INCOMMENT
  - That is, I have seen a character to transition me into a state where I am collecting characters for an identifier, a string constant, an integer constant, or a comment
- Each state has different rules *inside* the state
  - Zero or more letters or numbers for identifier
  - All characters to a double quote for a quoted string (note, newline in string is an error, and also note there are escapes inside the string)

Naming states

```
enum TokState {
   BEGIN,
   INID,
   INSTRING,
   ININT,
   INCOMMENT
};
TokState lexstate = BEGIN;
```

What is an enum?

- An enumerated type
- All possible values are symbolic names that are specified in the declaration
- A variable of an enumerated type can only take on one of the specified values
- The compiler assigns the values
- The values are integers but you cannot do math on them

State machine pseudocode

- Each state has its own unique symbolic name (one of the members of the enum)
- You can keep track of the current state in a variable of type “TokState” (or whatever you decided to name the enum)
- The “rules” for characters in a given state appear with the code for that state

Pseudocode

```
if( lexState == BEGIN ) {
   // BEGIN code
}
else if( lexState == INID ) {
   // INID code
}
…
```

```
switch( lexState ) {
case BEGIN:
   // BEGIN code
   break;
case INID:
   // INIT code
   break;
}
```


getNextToken outline

- Set initial state
- Loop, reading character at a time
- Select block of code to execute based on the current state
- Might change state based on character (ex: a letter in the BEGIN state indicates the beginning of an identifier, so change to INID)
- Return when token is recognized

getNextToken approach

- Draw a DFA for the lexical analyzer
  - This will tell you what states you need in your implementation
- Write pseudocode for the function
- Implement a state at a time

Assignment 2 pieces

- The lex.h header file is given
- You should implement the lexical analyzer function in one source file
- You should implement a test main program in another source file

- Vocareum will compile everything together
- DO NOT #include a .cpp file!

- The lexical analyzer function will have the following calling signature:

```
Lex getNextToken(istream& in, int& linenumber);
```

- The first argument to getNextToken is a reference to an istream that the function should read from. The second argument to getNextToken is a reference to an integer that contains the current line number. getNextToken will update this integer every time it reads a new line.
- getNextToken returns a Lex. A Lex is a class that contains a Token, a string for the lexeme, and the line number that the token was found on.
- A header file, lex.h, will be provided for you. You MUST use the provided header file. You may NOT change it.

Lex.h

- Definition for all of the possible token types
- Class definition
- Some function prototypes

NJIT New Jersey's Science & Technology University
COLLEGE OF COMPUTING SCIENCES

```
enum Token {
        // keywords
    PRINT, PRINTLN, REPEAT, BEGIN, END,

        // an identifier
    IDENT,

        // an integer and string constant
    ICONST, SCONST,

        // the operators, parens, semicolon
    PLUS, MINUS, STAR, SLASH, EQ, LPAREN, RPAREN, SC,

        // any error returns this token
    ERR,

        // when completed (EOF), return this token
    DONE
};
```


Lex class

```
class Tok {
        Token      token;
        string     lexeme;
        int        lnum;

public:
        Tok() {
                token = ERR;
                lnum = -1;
        }
        Tok(Token token, string lexeme, int line) {
                this->token = token;
                this->lexeme = lexeme;
                this->lnum = line;
        }

        bool operator==(const Token token) const { return this->token == token; }
        bool operator!=(const Token token) const { return this->token != token; }

        Token      GetToken() const { return token; }
        string     GetLexeme() const { return lexeme; }
        int        GetLinenum() const { return lnum; }
};

extern ostream& operator<<(ostream& out, const Tok& tok);

extern Tok getNextToken(istream& in, int& linenum);
```

constructors

overloaded operators

getters

NJIT New Jersey's Science & Technology University
COLLEGE OF COMPUTING SCIENCES

Some notes

- We will cover C++ class format, etc. in the next lecture
- The “overloaded operators” define how to compare a Lex to a Token
  - This allows you to write code like this:

```
Lex t;
t = getNextToken(…);
if( t == DONE || t == ERR ) { … }
```


External definitions

```
extern ostream& operator<<(ostream& out, const Lex& tok);

extern Lex getNextToken(istream& in, int& linenum);
```

- “extern” tells the compiler that someone will provide functions with these signatures
- *you* are the someone
- The operator<< function is an overload that lets you print a Lex to a stream (more next week)

Lexical Rules

- The lexical rules represent the patterns your lexical analyzer must recognize
- You should understand the patterns and build a DFA representing all of the patterns
- Assignment requires writing code to implement the DFA

Pseudocode for getNextToken

- Keep track of your state
- Each state has a set of valid, expected characters in that state
- For each input character
  - Consider your state
  - Decide if the character is valid for that state
  - Process the character (changing state if necessary)
- A switch statement based on state is one reasonable way to implement this
- Create something to represent all the states you need (an enum works well for this)

A small piece of getNextToken

```
Lex
getNextToken(istream& in, int& linenum)
{
        enum LexState { BEGIN, INID, /* others */ };
        LexState lexstate = BEGIN;
        string lexeme;
        char ch;

        while(in.get(ch)) {

                switch( lexstate ) {
                case BEGIN:
                        if( ch == '\n' ) {
                                linenum++;
                        }

                        if( isspace(ch) )
                                continue;

                        lexeme = ch;

                        if( isalpha(ch) ) {
                                lexstate = INID;
                        }
                        else /* more … */
```


Peek And Putback

- Your getNextToken function might need to look at the next character from input to decide if the token is finished or not
- Method 1: use the peek() method, examine the next character, and only read it if it belongs to the token
- Method 2: if you read a character that does not belong to the token, use the putback() method to put it back, so that you get() it next time

Main test program

- Process arguments
- Decide where input comes from
- Call getNextToken until it returns DONE or ERR
- Keep statistics
- Print statistics

Reading from cin or a file

- The first argument to getNextToken is an istream&
- An istream& (a reference to an istream) can refer to cin, or it can refer to an ifstream
  - cin is an istream. Therefore you can pass cin as the first argument if you want to read from standard input
  - ifstream inherits from istream, so it "is a" istream. Therefore you can pass the stream as the first argument if you want to read from a file
- Keep it simple: create an istream* representing the input. Initialize it to either &cin or &the file you opened. Then just pass *that variable to getNextToken

Main loop

```
istream *in;
int lineNumber = 0;
…

Token tok;
while( (tok = getNextToken(*in, lineNumber)) != DONE && tok != ERR )
{
        // handle verbose mode
        // keep statistics
}
```

- getNextToken routine will read things a character at a time
- the main program will process the input a token at a time
- counts and statistics are kept in main

Required statistics

- How many lines in the input?
- How many tokens in the input?
- What strings are in the input?
- What integers are in the input?
- What identifiers are in the input?


NJIT
THE EDGE IN KNOWLEDGE