Lexical and Syntax Analysis

Chapter 2 Part 2

Traditional Compilation Process

- Lexical analysis – Used to find the names and numeric and operator literals (tokens)
  - Use a scanner
- Syntax analysis –
  - Determine if the program is syntactically correct
  - Find the parse tree for the program.
  - Use a parser
- Usually separate processes

Lexical Analysis

Identify the tokens in these programs

C

```
sum = 0;
prod = 1;
for( int j = 1; j<= 10 ; j++)
{
            sum += j;
            prod *= j;
}
```

Pascal

```
sum := 0;
prod := 1;
for j := 1 to 10 do
begin
   sum    := sum + j;
    prod  := prod * j
end
```

Calculator Language Extended BNF (Handout)

<progr> → <stmt-lst> .

<stmt-lst> → <stmt> { ; <stmt>}*

<stmt> → <read-stmt> | <write_stmt> | <assign_stmt>

<read-stmt> → read <id>

<write-stmt> → write <id>

<assign-stmt> → <id > = <expr>

<expr> → <expr> { <op> <expr>}* | (<expr>) |

<number> | <id>

<op> → + | - | * | /

<id> → <letter> { <letter > | <digit> }*

<number> → <digit> { <digit> }* [. { <digit> }*]

Calculator Language Programs

```
height = 67.0;
width = 3.4;
area = height * width;
write area .
--------------------
read x;
y = 5 *  ( x + 10);
write y .
```

Tokens for Calculator Language

Token	Pattern
read	read
write	write
id	letter { letter | digit }*
number	digit {digit}* [. {digit}*]
assign	=
op	+ | - | * | /

more tokens on next page

Tokens for Calculator Language -2

Token	Pattern
lparen	(
rparen	)
semicolon	;
period	.

Scanner to recognize tokens

- Scan text of program, looking for tokens
- Skip over white space (blanks, newlines,…)
- As soon as you find a token match, report token type found and continue scanning
- Match the longest substring possible
  - "3.1415 " is the number 3.1415 , not a 3 and . and 1415
  - "reader" is the identifier *reader* , not *read* and *er*

Pseudocode for scanner

- Distribute copies in class

Pseudocode Scanner for Calculator Language

```
set nextchar to first char in program
while not at end of program
{
        skip over white space
        if nextchar is  in set { ( , ) , +, -, * , / ,  ; , =,  . }
                return appropriate single char token
        else if nextchar is a digit
                read additional digits
                if nextchar is a  .
                        read any additional digits
                        return number token with number
                else
                        return number token with number
        else if nextchar is a letter
                read any additional letters and digits
                check to see if resulting string is read or write
                if so, return corresponding token
                else return id token with id
        else
                return error
}
```


Programming Project #1 Part A

- Write a Java or C++ program to implement the scanner for the Calculator Language.
- Testing: Test on lexically “correct” and “incorrect” sample programs.

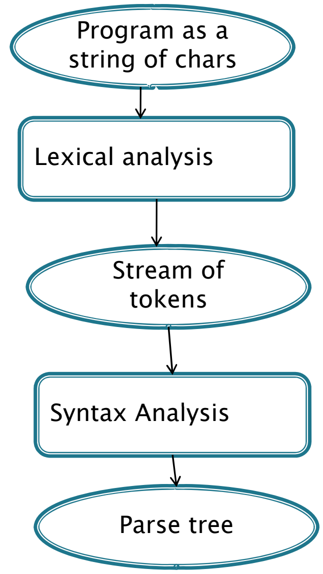
See posted assignment for more details.
Due: Sept 26, 2013

Syntax Analysis

Parsing

- Given a BNF grammar description of the language
  - Determine if a program is a legal program
  - Create an (implicit) parse tree for the program
  - If program is not a legal program, return a helpful error message
- There are software tools to create a syntax analyzer given a BNF description (eg. yacc)

Lexical + Syntax Analysis Steps



Two Basics Kinds of Parsers

- Top down parser : Create a parse tree for a string from the top of the tree down
  - Easier to code
  - Serious Grammar Limitations
- Bottom-up parsers: Create a parse tree for a string from the bottom of the tree up
  - Complex to code but can be more efficient
  - Can handle more grammar types than top down parsing

Illustrate top down parsing vs. bottom up parsing with abstract grammar:

S → A B | B S B
A → a
B → b

Here the capital letters are the nonterminals and lowercase letters are the terminals.

See board work.

Recursive Descent Parsing

- Top-down parser
- Collection of recursive subprograms
  - One for each nonterminal
- Grammar should not have any left recursive rules
  - <st_list> → <st_list> ;< stmt> | <stmt> BAD
  - <st_list> → <stmt> ; <st_list> | <stmt> OK

Write a recursive descent parser for the following grammar.

Assignment Language

<prog> → <stmtList>
<stmtList> → <stmt> | <stmt> ;<stmtList>
<stmt> → <id> = <num>
<id> → x | y | z
<num> → 0|1|2|3|4|5|6|7|8|9

Token List: { x,y,z,0,1,2,3,4,5,6,7,8,9, ; , =}

Parser should report "successful parse" or "syntax error".
Project does not require you to create a syntax tree.

Input: string of characters representing list of tokens.
Output: "successful parse" or "syntax error" as appropriate

Examples:

tokens	Parser returns
"x=6;y=5$"	"successful parse"
"x=78$"	"syntax error"
"y = 0; z = 8$"	"syntax error" (white space)
"y=z=4$"	"syntax error"
"x=5;y=2"	"syntax error"
" x =8;$"	"syntax error"

Distribute Recursive Descent Parser for Assignment Language

Go over it in class

Pseudocode for recursive descent parser

```
//global variables
char nextchar;  String  tokens

void  program() {  }

void stmtList()  {  }

void stmt() {  }

void id() {}

void num() { }

void getChar()  {   //gets next char from tokenList }
```

Note: tokens string will contain only chars from token list and will end with a “$’ to indicate end of input.

```
void main() {
    //read in tokens
    tokens = "x=5;y=8$"

    getChar();

    program( );

    print( " Successful parse");
 }
```



```
void program() {

      stmtList();

      if ( nextchar == '$')
        print "success"
      else
        error();
}
```

Grammar Rule
<prog> → <stmtList>

```
void stmtList()  {
        stmt();
         if (nextchar == ';') {
            getChar();
         stmtList();
        }
       else
         return;
}
```

Grammar Rule
<stmtList> → <stmt>
| <stmt> ;<stmtList>


```
void stmt()  {
    id();

     if (nextchar == '=' )
     {
         getchar();
         num();
     }
     else
         error();

}
```

Grammar Rule
<stmt> → <id> = <num>

```
void id()
{
        if (nextchar == 'x'  || nextchar == 'y'
                || nextchar == 'z')
        {
                getchar();
                 return;
         }
         else
           error();
{
```

Grammar Rule
<id> → x | y | z


```
void num()
{
        if (isdigit(nextchar) )
        {
                getchar();
                 return;
         }
         else
                error();
}
```

Grammar Rule
<num> → 0|1|2|3|4|5|6|7|8|9

Programming Project #1 Part B

- Modify the recursive descent parser for the Assignment Language so that it prints helpful error messages. Hint: Modify the error method so that it accepts a string as a parameter. This will be the error message.
- Thoroughly test your program to show all of the errors it can detect and report.
- Download parser from course Moodle site. See Moodle for complete description of Project 1 Part B

Due Sept 26, 2013