Parsing and Evaluation

Lecture 10

- The next two lectures cover parsing.
- To parse a sentence in a formal language is to break it down into its syntactic components.
- Parsing is one of the most basic functions every compiler carries out, as well has having many other uses.
- In this lecture, we'll be looking at how to define a formal language, and we'll study an example of parsing and evaluating an arithmetic expression.

Backus-Naur Form

- Parsing is an operation that's carried out on *formally defined* languages. So how do we formally define a language?
- We do it using another formally defined language, called a ***meta-language***, such as **Backus-Naur** Form (or BNF).
- Here's an example of BNF. It defines what a "product" is.

<product> ::= <number> | <product> * <number>

- This says that a product is either a number or a number followed by "*" followed by another product. Note that it's a recursive definition.
- The "|" symbol is part of BNF, and it means "or". (The "*" symbol is not part of BNF, but is part of the language being defined.)

Arithmetic Expressions in BNF

- The following BNF definitions describe the syntax of an arithmetic expression.

<expression>	::=	<term> |
		<expression> + <term> |
		<expression> – <term>
<term>	::=	<factor> |
		<term> * <factor> |
		<term> / <factor>
<factor>	::=	<number> | (<expression>)

Examples of Arithmetic Expressions

- From the preceding definitions, it should be clear that the following are all syntactically correct expressions:

$$5$$
$$5 + 8$$
$$(5 + 8)$$
$$5 * 2 + 1$$
$$5 * (2 + 1)$$

- while the following are not:

$$+$$
$$5 +$$
$$((5 + 8)$$
$$5 * 2 + + 1$$

Lexical Analysis I

- Before we parse a sentence in a formal language, it's convenient to break it down into a list of lexemes.
- A **lexeme** (sometimes also called **token**) is the smallest syntactic unit in a language. The following are typical kinds of lexemes:
  - Numbers (ie: numeric strings)
  - Names (ie alphanumeric strings)
  - Operators (including brackets) (eg: *, +, ^, etc)
- For example, the sentence "(64 + 7) * 128" is broken down into a list of seven lexemes: "(", "64", "+", "7", ")" , "*", and "128".

Lexical Analysis II

- In what follows, we'll assume that the expression is a string pointed to by a pointer "expression".
- We'll also assume we have the following access functions are already written:

```
bool notEmpty (const string& expression);
//returns true if expression is not empty string

char nextChar (const string& expression);
//returns next character in expression, nothing consumed

char getNextChar (string& expression);
//returns next character in expression, char consumed

int getNum (string& expression);
//returns value of next integer in expression, consume integer
```

- Don't worry about the data type string and how these access functions are written for now. We will deal with them later.

Recursive versus Iterative Processing

- Here's the BNF for an expression again.

```
<expression> ::=   <term> |
                   <expression> + <term> |
                   <expression> – <term>
```

- This definition is **left-recursive** — the recursive part is the leftmost item on the right-hand-side on the definition. If we translated this directly into a recursive procedure, it would go into an infinite loop.
- Here's a **right-recursive** version which defines the same thing.

```
<expression> ::=   <term> |
                   <term> + <expression> |
                   <term> – <expression>
```

- The right-recursive version can be translated directly into a recursive parsing procedure, but processes expressions in the reverse order to what we require.
- So we use the left-recursive version, and translate it into an **iterative procedure**.

Evaluating an Expression I

- Here's the BNF again.

```
<expression> ::=    <term> |
                    <expression> + <term> |
                    <expression> – <term>
```

- To ***evaluate*** an expression E iteratively, this is what you do.
  - Evaluate the next term in E, and call the result X.
  - While the next character in E is an operator ("+" or "–"),
    - Read past the operator.
    - Evaluate the next term in E, calling the result Y.
    - Let X be X + Y or X – Y, depending on the operator.
- The value of E is the final value of X.

Evaluating an Expression II

- Here’s the C++ code. It should be self-explanatory.

```
void evalExpression (string& expression, int& result) {
   int temp;
   char op;

   evalTerm (expression, result);
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='+') ||
          (nextChar(expression)=='-'))){
       op = getNextChar(expression);
       evalTerm(expression, temp);
       if (op=='+')
           result = result + temp;
       else
           result = result - temp;
   }
}
```

Evaluating a Term I

- Evaluating a term is almost identical. Here's the BNF.

```
<term> ::=   <factor> |
             <term> * <factor> |
             <term> / <factor>
```

- To evaluate a term T, this is what you do.
  - Evaluate the next factor in T, and call the result X.
  - While the next character in T is an operator ("*" or "/"),
    - Read past the operator.
    - Evaluate the next factor in T, calling the result Y.
    - Let X be X * Y or X / Y, depending on the operator.
- The value of T is the final value of X.

Evaluating a Term II

- And here's the C++ code.

```
void evalTerm (string& expression, int& result) {
   int temp;
   char op;

   evalFactor (expression, result);
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='*') ||
          (nextChar(expression)=='/'))){
      op = getNextChar(expression);
      evalFactor(expression, temp);
      if (op=='*')
         result = result * temp;
      else
         result = result / temp;
   }
}
```

Evaluating a Factor I

- Evaluating a factor is a little different, and involves a recursive call. Here's the BNF again.

<factor> ::= <number> | (<expression>)

- To evaluate a factor F, this is what you do.
  - If the next character in F isn't "(" then let X be the first number in F.
  - Otherwise evaluate the next expression in F, and call the result X. Then read past the ")".
- The value of F is X.

Evaluating a Factor II

- Here's the C++ code.

```
void evalFactor (string& expression, int& result) {

  if (notEmpty(expression) &&
        (nextChar(expression)!='('))
     result = getNum(expression);
  else {
     getNextChar(expression);          // skip '('
     evalExpression(expression, result);
     getNextChar(expression);          // skip ')'
  }
}
```

Evaluating a Factor III

- Note that evalExpression(), evalTerm() and evalFactor() are **mutually recursive**. That is:

evalExpression() → calls → evalTerm() → calls → evalFactor()

calls

- We need to use function prototype to specify functions arguments before they are used. Here are the function prototype for all three functions:

```
void evalExpression (string& expression, int& result);
void evalTerm (string& expression, int& result);
void evalFactor (string& expression, int& result);
```

Access Routines I - strings

- Borland C++ understand a data type called **string**. Here are some examples of access functions for string data type. (They are actually membership functions, something we will consider in a later lecture.)

```
string s1;
string s2 = "initial value";
s3.insert (3, "123");        // s3 = "ini123tial value"
s3.erase (4, 2);             // s3 = "ini3tial value"
s3.replace (4, 2, "pqr");    // s3 = "intpqrial value"

char d[16];
strcpy(d, s3.c_str());        // copy s3 into array d
cout << s3.length() << endl; // returns no of char in s3
s1 = "mississippi";
cout << s1.find("ss") << endl;      // returns 2
cout << s1.find("ss", 3) << endl;   // returns 5
if (s2.empty())
  cout << "string is empty" << endl;
```

Access Routines II

- Now we can write our access routines for this program:

```
//------------- Access Routines ---------------

bool isNum (char c) {
//returns true if c is 0-9
  return ((c >= '0') && (c <= '9'));
}

bool notEmpty (const string& expression) {
//returns true if expression is not empty string
  return (!expression.empty());
}

char nextChar (const string& expression) {
//returns next character in expression, nothing consumed
  return expression[0];
}
```

Access Routines III

```
char getNextChar (string& expression) {
//returns next character in expression, char consumed
   char c = '\0';
   if (notEmpty(expression)) {
      c = expression[0];
      expression.erase(0,1);     //eat one character
      }
   return c;
}
```

Access Routines IV

```
int getNum (string& expression) {
//returns value of next integer in expression, consume integer
  string numToken;
  int i=0;
  int maxString = expression.length();
  while (isNum(expression[i]) && i<=maxString)
      i++;
  numToken = expression.substr(0, i);     //extract the number string
  expression = expression.erase(0, i);    //remove it from expression
  return atoi(numToken.c_str());          //convert string to integer
}
// End of Access Routines
```

Example

expression → 7 * (5 + 4)

```
void evalExpression (string& expression, int& result) {
   int temp;
   char op;

   evalTerm (expression, result);
   while ((notEmpty(expression)) &&
          ((nextChar(expression)=='+') ||
           (nextChar(expression)=='-'))){
       op = getNextChar(expression);
       evalTerm(expression, temp);
       if (op=='+')
           result = result + temp;
       else
           result = result - temp;
   }
}
```

Example

expression → 7 * (5 + 4)

```
void evalExpression (string& expression, int& result) {
```

```
void evalTerm (string& expression, int& result) {
   int temp;
   char op;

   evalFactor (expression, result);
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='*') ||
          (nextChar(expression)=='/'))){
      op = getNextChar(expression);
      evalFactor(expression, temp);
      if (op=='*')
          result = result * temp;
      else
          result = result / temp;
   }
}
```

Example

expression → 7 * (5 + 4)

Returns result = 7
expression string = " *(5+4)"

```
void evalExpression (string& expression, int& result) {
```

```
void evalTerm (string& expression, int& result) {
```

```
void evalFactor (string& expression, int& result) {

  if (notEmpty(expression) &&
       (nextChar(expression)!='('))
     result = getNum(expression);
  else {
     getNextChar(expression);          // skip '('
     evalExpression(expression, result);
     getNextChar(expression);          // skip ')'
  }
}
```

Example

op = *

expression → (5 + 4)

result = 7

```
void evalExpression (string& expression, int& result) {
```

```
void evalTerm (string& expression, int& result) {
   int temp;
   char op;

   evalFactor (expression, result);
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='*') ||
          (nextChar(expression)=='/'))){
      op = getNextChar(expression);
      evalFactor(expression, temp);
      if (op=='*')
          result = result * temp;
      else
          result = result / temp;
   }
}
```

Example

expression → 5 + 4)

nextChar() = (

```
void evalExpression (string& expression, int& result) {
```

```
void evalTerm (string& expression, int& result) {
```

```
void evalFactor (string& expression, int& result) {

  if (notEmpty(expression) &&
       (nextChar(expression)!='('))
     result = getNum(expression);
  else {
     getNextChar(expression);          // skip '('
     evalExpression(expression, result);
     getNextChar(expression);          // skip ')'
  }
}
```

Example

expression → 5 + 4)

nextChar() = (

```
void evalExpression (string& expression, int& result) {
   int temp;
   char op;

   evalTerm (expression, result);
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='+') ||
          (nextChar(expression)=='-'))){
       op = getNextChar(expression);
       evalTerm(expression, temp);
       if (op=='+')
           result = result + temp;
       else
           result = result - temp;
   }
}
```

Example

expression → 5 + 4)

Result = 5

```
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
   int temp;
   char op;

   evalFactor (expression, result);
   while ((notEmpty(expression)) &&
          ((nextChar(expression)=='*') ||
           (nextChar(expression)=='/'))){
        op = getNextChar(expression);
        evalFactor(expression, temp);
        if (op=='*')
            result = result * temp;
        else
            result = result / temp;
   }
}
```

Example

op = +

expression → + 4)

Returns temp = 4

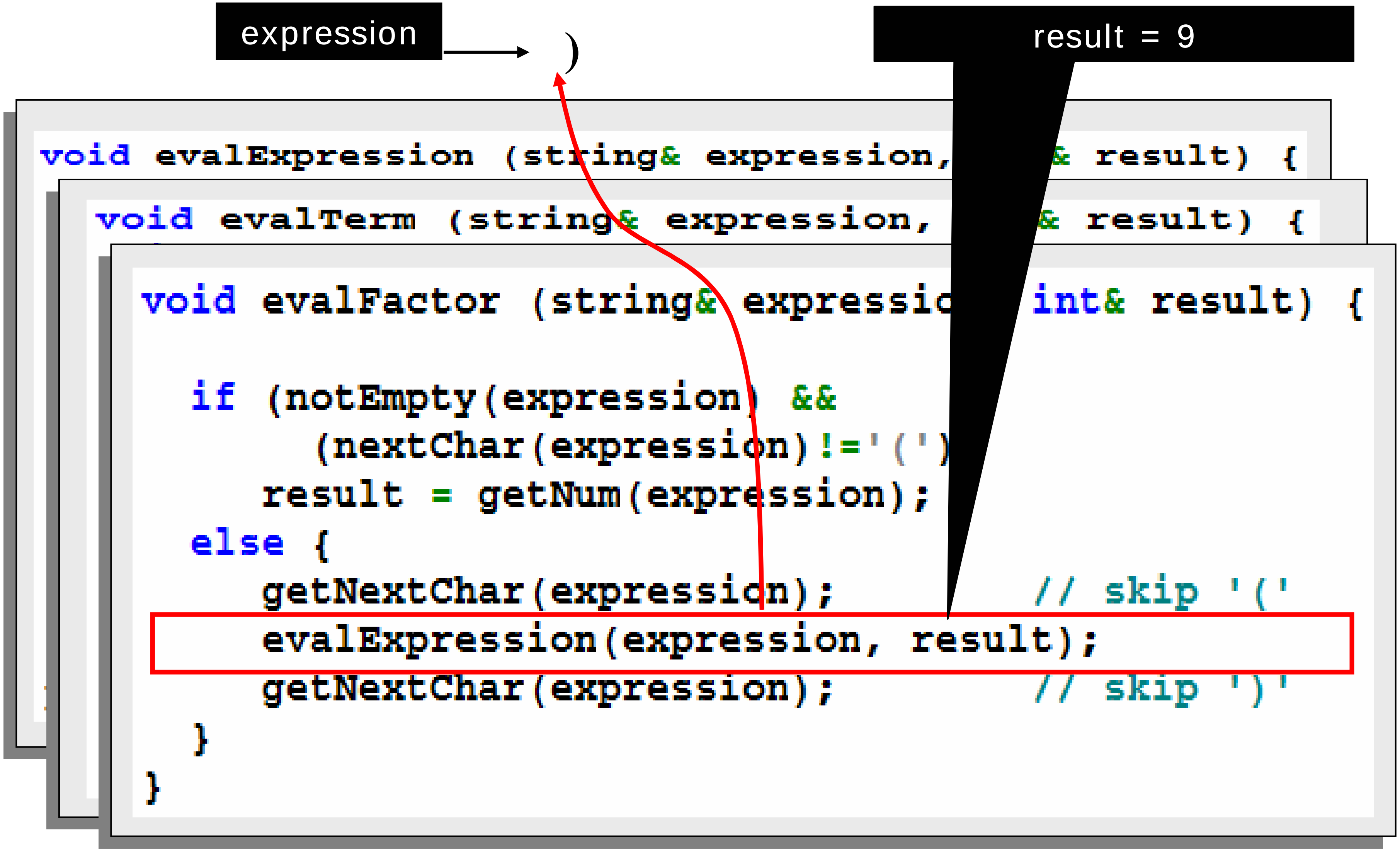
```
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
```

```
void evalExpression (string& expression, int& result) {
  int temp;
  char op;

  evalTerm (expression, result);
  while ((notEmpty(expression)) &&
        ((nextChar(expression)=='+') ||
         (nextChar(expression)=='-'))){
     op = getNextChar(expression);
     evalTerm(expression, temp);
     if (op=='+')
        result = result + temp;
     else
        result = result - temp;
  }
}
```

result = 9

Example


expression
)
result = 9
void evalExpression (string& expression, & result) {
void evalTerm (string& expression, & result) {
void evalFactor (string& expressio int& result) {
if (notEmpty(expression) &&
(nextChar(expression)!='(')
result = getNum(expression);
else {
getNextChar(expression); // skip '('
evalExpression(expression, result);
getNextChar(expression); // skip ')'
}
}

Example

expression

result = 7

temp = 9

```
void evalExpression (string& expression, int& result) {
```

```
void evalTerm (string& expression, int& result) {
   int temp;
   char op;

   evalFactor (expression, result)
   while ((notEmpty(expression)) &&
         ((nextChar(expression)=='*') ||
          (nextChar(expression)=='/'))){
      op = getNextChar(expression);
      evalFactor(expression, temp);
      if (op=='*')
          result = result * temp;
      else
          result = result / temp;
   }
}
```

Example

expression

final result = 63

```
void evalExpression (string& expression, int& result) {
   int temp;
   char op;

   evalTerm (expression, result);
   while ((notEmpty(expression)) &&
          ((nextChar(expression)=='+') ||
           (nextChar(expression)=='-'))){
      op = getNextChar(expression);
      evalTerm(expression, temp);
      if (op=='+')
         result = result + temp;
      else
         result = result - temp;
   }
}
```