Software Engineering II
Algorithms and Data Structures

Parsing and Expression Evaluation

Dr. Christos Bouganis
Imperial College London

Objectives

- Learn how to parse and evaluate an expression

In other words:
How does the computer evaluate the following expression?

*5*2+3*2+50/10*

- Learning Outcomes
  - Familiarize students with Backus-Naur Form
  - Familiarize students with expression evaluation
  - Study how to map expression evaluation to an algorithm

What is 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.
  - Production of a **correct** machine language representation

3

Backus-Naur Form

- Computer languages are represented by formal grammars.
- These grammars are usually represented by another formally defined language, such as **Backus-Naur Form** (or BNF), or syntax diagrams.
- 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 an another product followed by "*" followed by a number. 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.)

4

Arithmetic Expressions

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

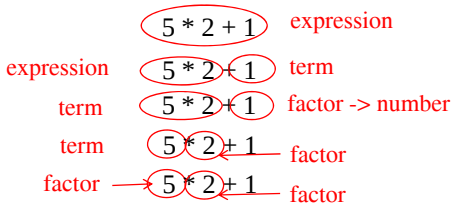
```
<expression> ::=   <term> |
                       <expression> + <term> |
                       <expression> – <term>

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

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

5

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
```

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

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

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

6

Recursive versus Iterative Processing

- Here's the BNF for an expression again.

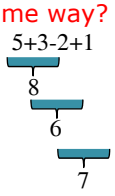
```
<expression> ::= <term> |
                 <expression> + <term> |
                 <expression> – <term>
```

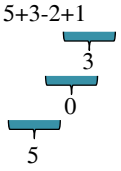
- This definition is **left-recursive** — the recursive part is the leftmost item on the right-hand-side on the definition.
- Here's a **right-recursive** version which defines the same thing.

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

Do they evaluate an expression in the same way?

5+3-2+1





- We will use iterative processing.

7

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 (i.e. numeric strings)
  - Names (i.e. alphanumeric strings)
  - Operators (including brackets) (e.g. *, +, ^, etc)
- For example, the sentence "(64 + 7) * 128" is broken down into a list of seven lexemes: "(", "64", "+", "7", ")" , "*", and "128".

8

Lexical Analysis II

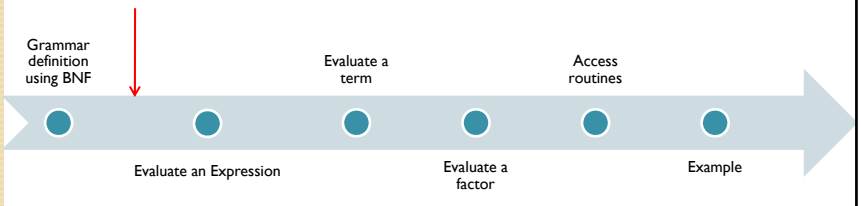
- In what follows, we'll assume that the expression is a string pointed to by a pointer "expression".
- We'll also assume that we have available the following access functions:

```
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.

9

Outline



10

Evaluating an Expressions I

- Here's the BNF again.

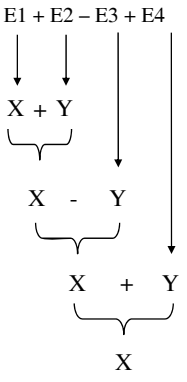
```
<expression> ::= <term> |
                 <expression> + <term> |
                 <expression> – <term>
```

- 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.

11

Example

- 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.



12

Evaluating an Expressions 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;
  }
}
```

13

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.

14

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;
  }
}
```

15

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.

16

Evaluating a Factor II

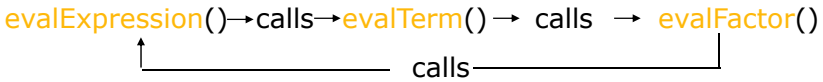
- Here's the C++ code.

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

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

17

Evaluating a Factor III

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



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

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

18

Access Routines I

- 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];
}
```

19

Access Routines II

```
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;
}
```

20

Access Routines III

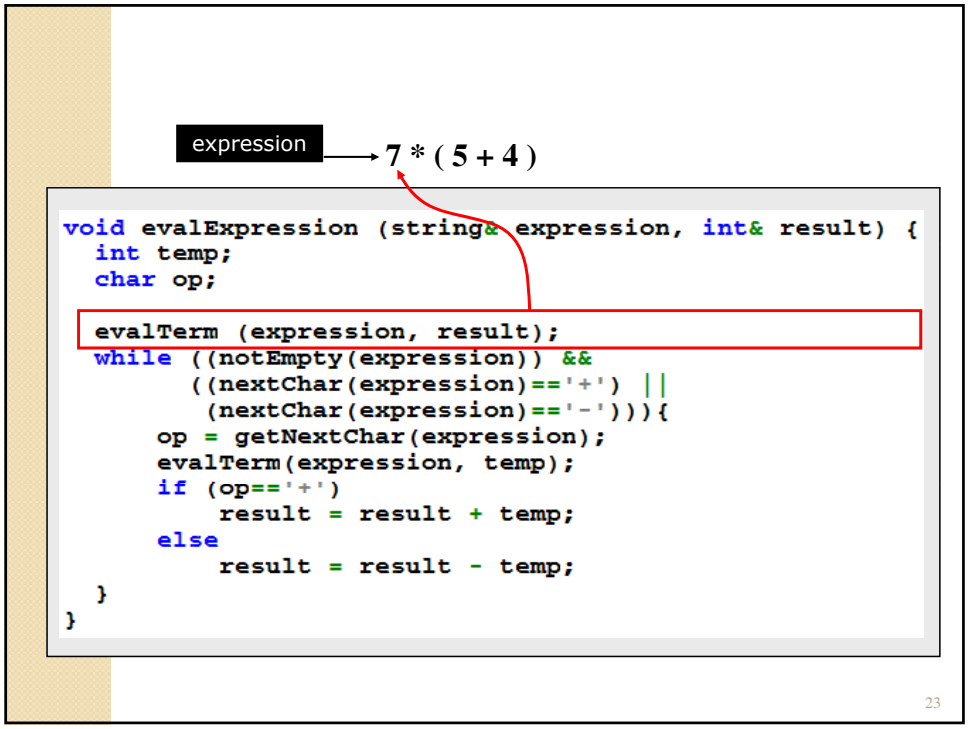
```
int getNum (string& expression) {
//returns value of next integer in expression, consume integer
  string numToken;
  int i=0;
  int maxString = expression.length();
  while (isNum(expression[i]) && i<=maxString)
      i++;
  numToken = expression.substr(0, i);     //extract the number string
  expression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

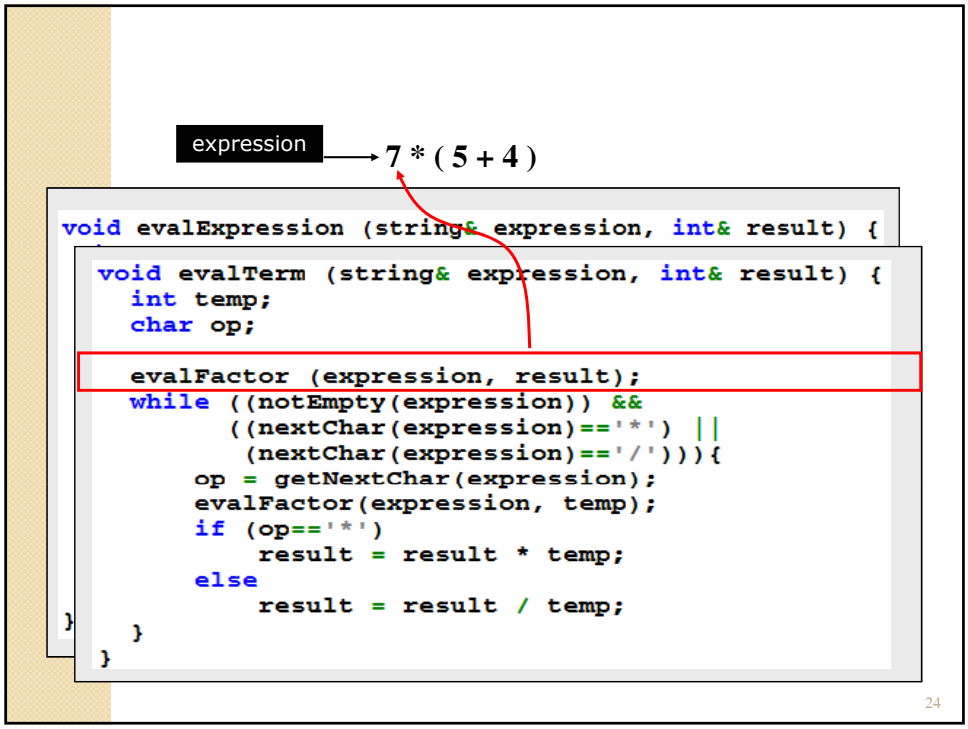
21

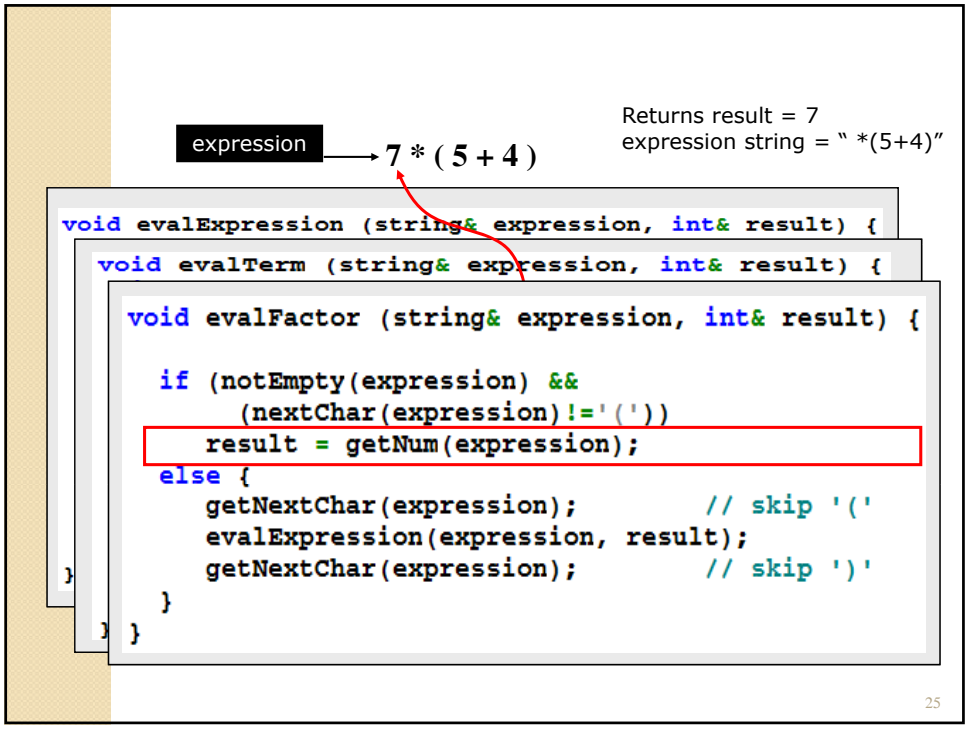
AN EXAMPLE

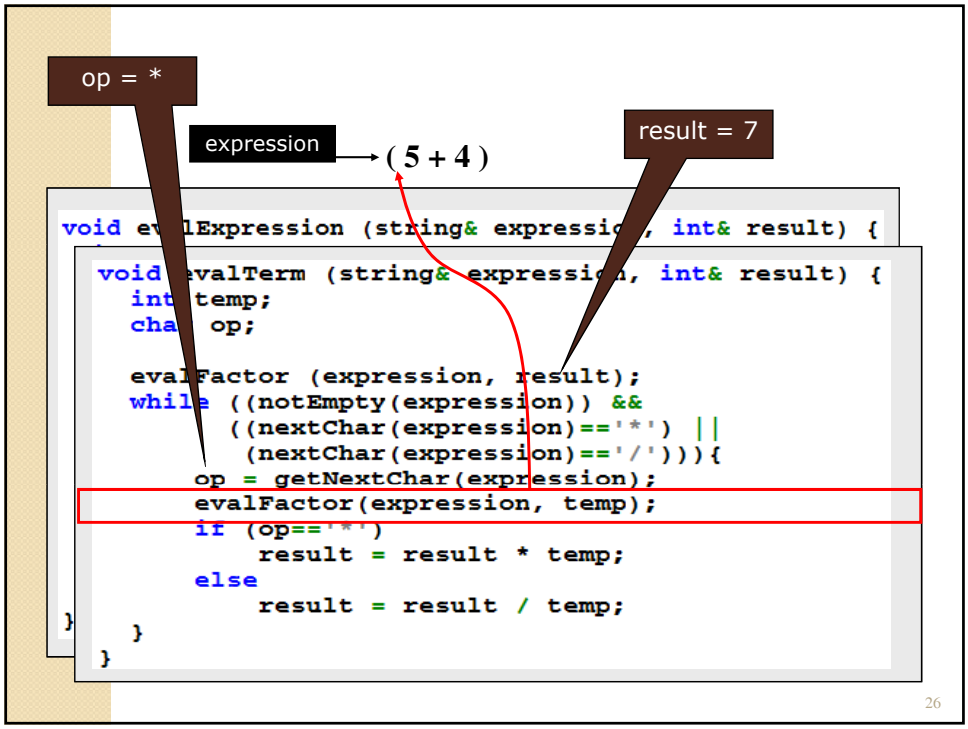
Evaluate 7*(5+4)

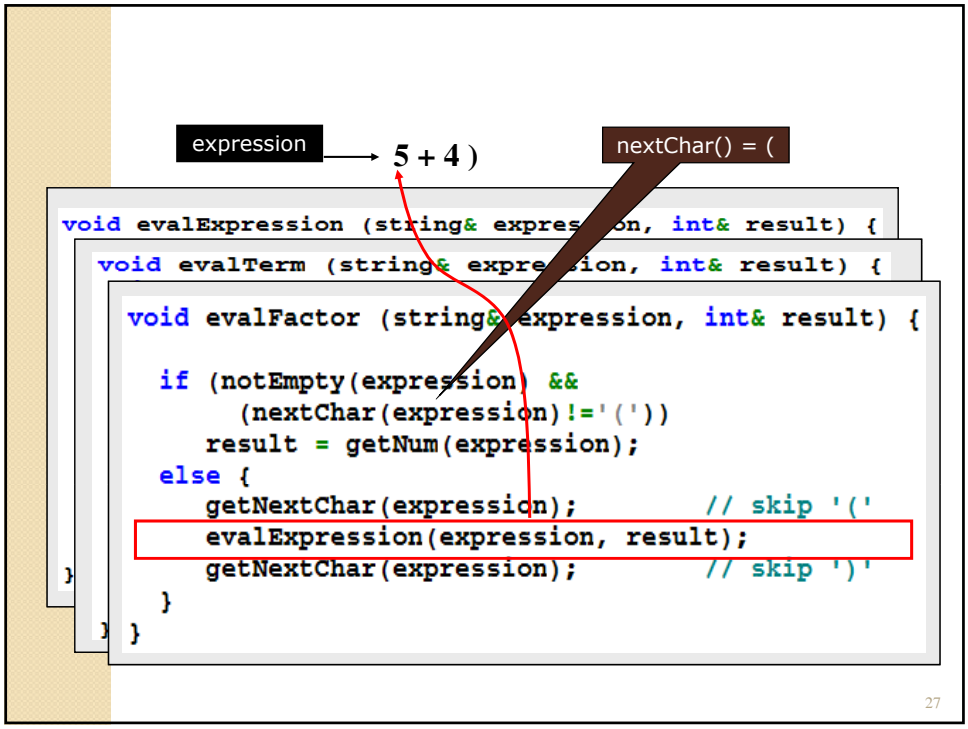
22

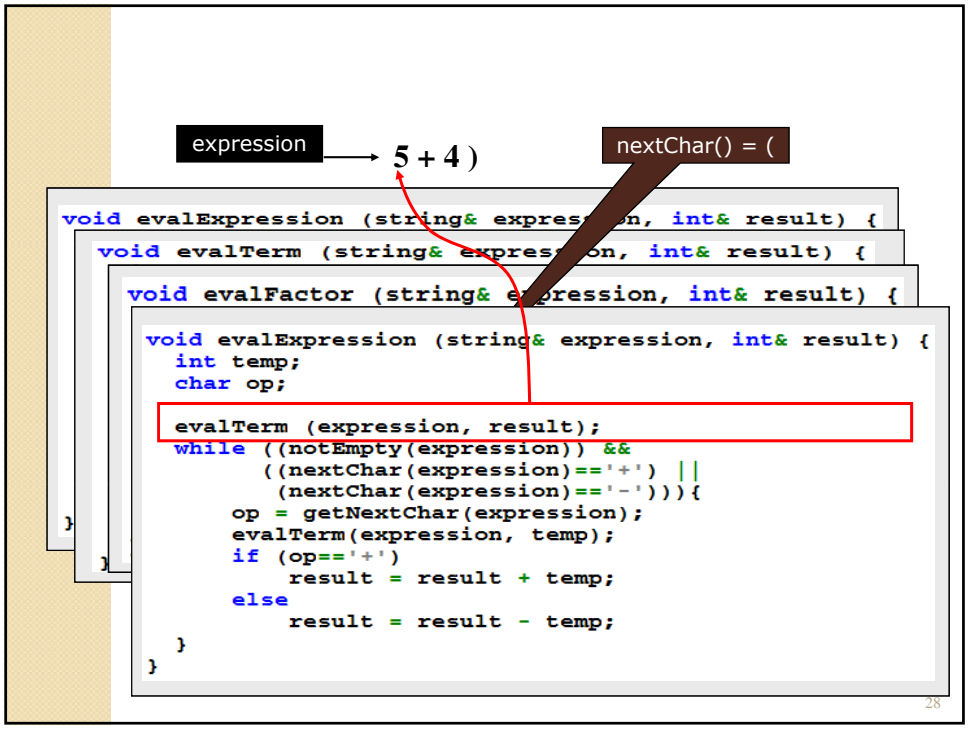

expression
7 * (5 + 4)
void evalExpression (string& expression, int& result) {
int temp;
char op;
evalTerm (expression, result);
while ((notEmpty(expression)) &&
((nextChar(expression)=='+') ||
(nextChar(expression)=='-'))){
op = getNextChar(expression);
evalTerm(expression, temp);
if (op=='+')
result = result + temp;
else
result = result - temp;
}
}
23

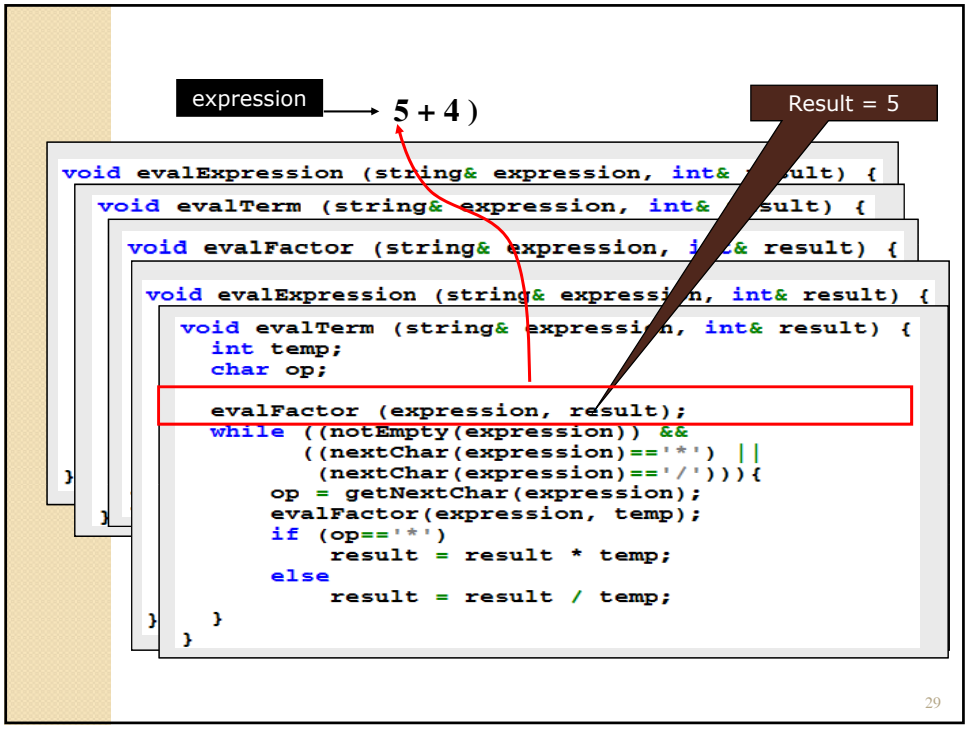

expression
7 * (5 + 4)
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
int temp;
char op;
evalFactor (expression, result);
while ((notEmpty(expression)) &&
((nextChar(expression)=='*') ||
(nextChar(expression)=='/'))){
op = getNextChar(expression);
evalFactor(expression, temp);
if (op=='*')
result = result * temp;
else
result = result / temp;
}
}
24

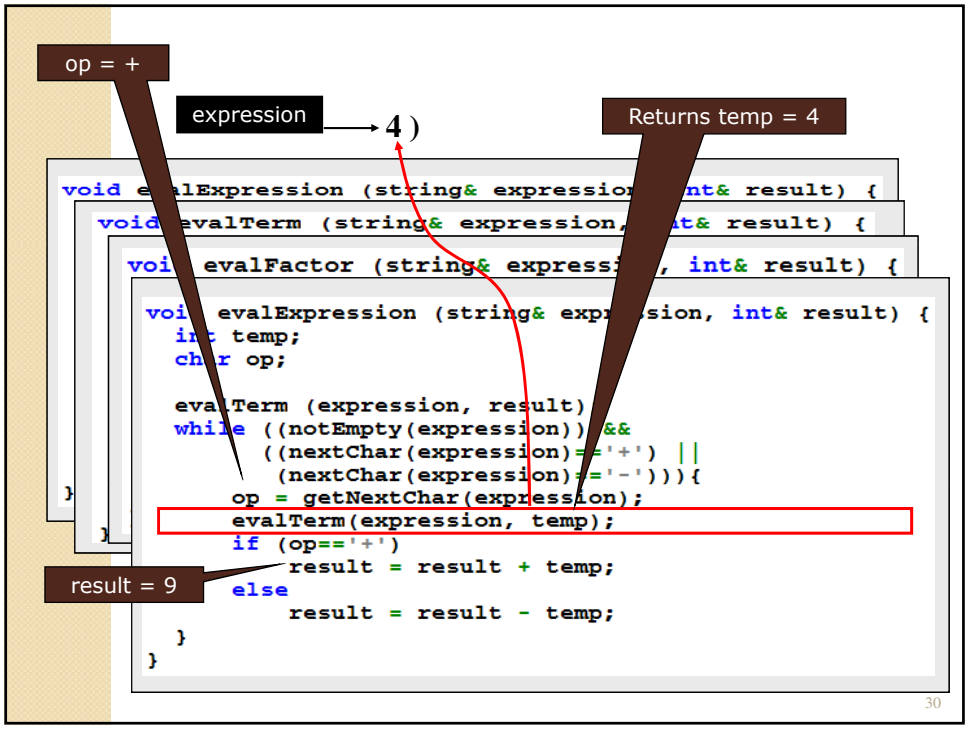

expression
7 * (5 + 4)
Returns result = 7
expression string = " *(5+4)"
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
if (notEmpty(expression) &&
(nextChar(expression)!='('))
result = getNum(expression);
else {
getNextChar(expression); // skip '('
evalExpression(expression, result);
getNextChar(expression); // skip ')'
}
}
}
25

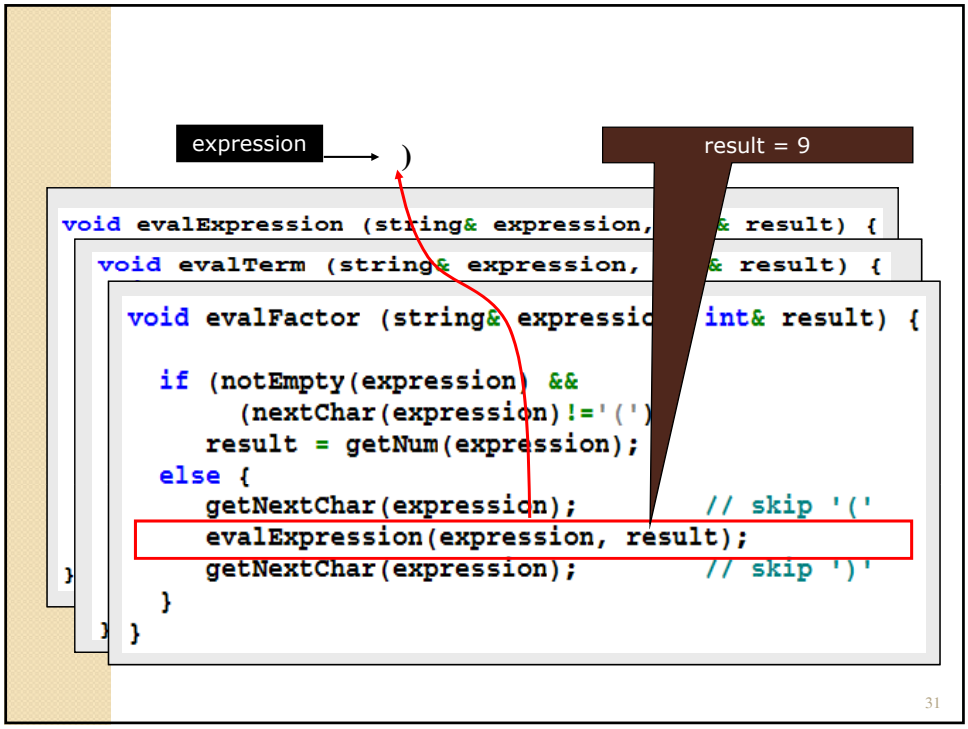

op = *
expression
(5 + 4)
result = 7
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
int temp;
char op;
evalFactor (expression, result);
while ((notEmpty(expression)) &&
((nextChar(expression)=='*') ||
(nextChar(expression)=='/'))){
op = getNextChar(expression);
evalFactor(expression, temp);
if (op=='*')
result = result * temp;
else
result = result / temp;
}
}
26

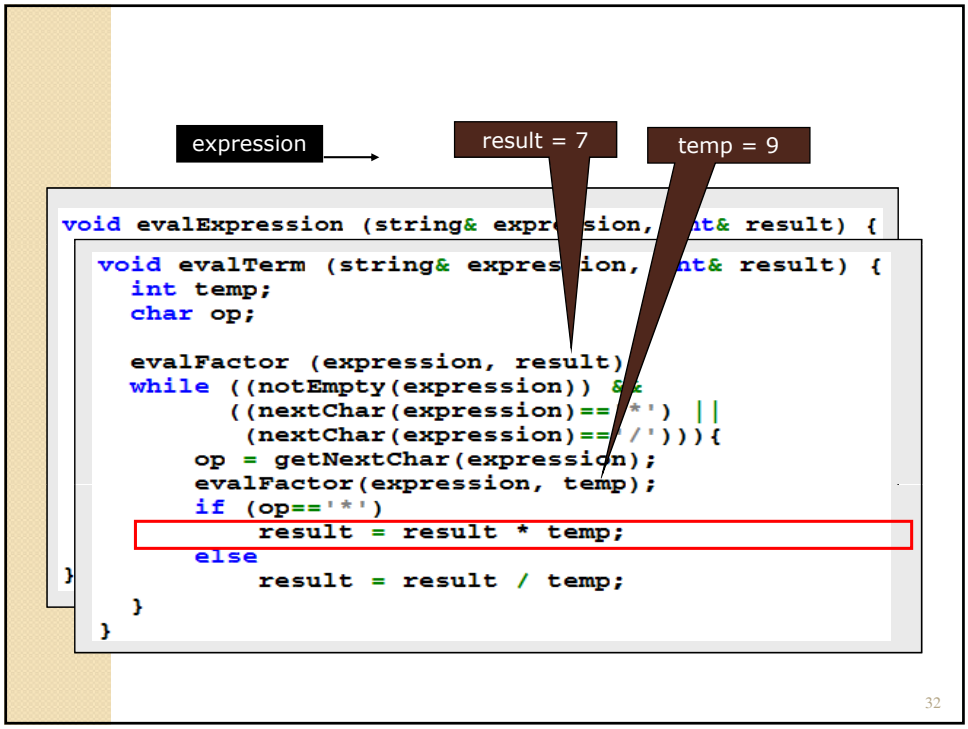

expression
5 + 4)
nextChar() = (
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
if (notEmpty(expression) &&
(nextChar(expression)!='('))
result = getNum(expression);
else {
getNextChar(expression); // skip '('
evalExpression(expression, result);
getNextChar(expression); // skip ')'
}
}
27

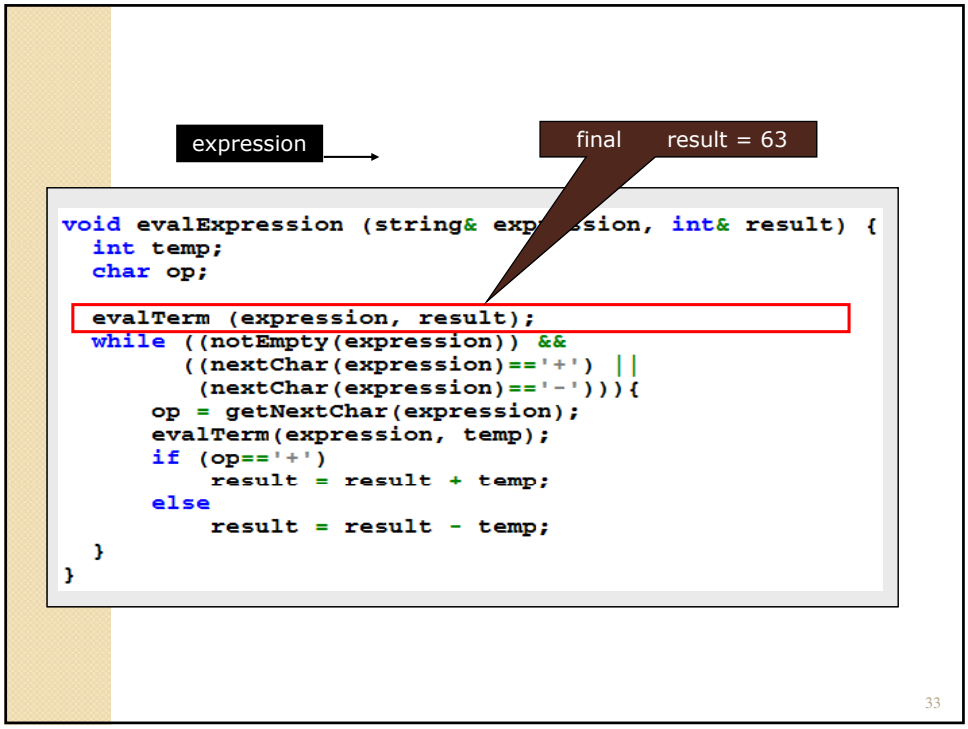

expression
5 + 4)
nextChar() = (
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
void evalExpression (string& expression, int& result) {
int temp;
char op;
evalTerm (expression, result);
while ((notEmpty(expression)) &&
((nextChar(expression)=='+') ||
(nextChar(expression)=='-'))){
op = getNextChar(expression);
evalTerm(expression, temp);
if (op=='+')
result = result + temp;
else
result = result - temp;
}
}
28

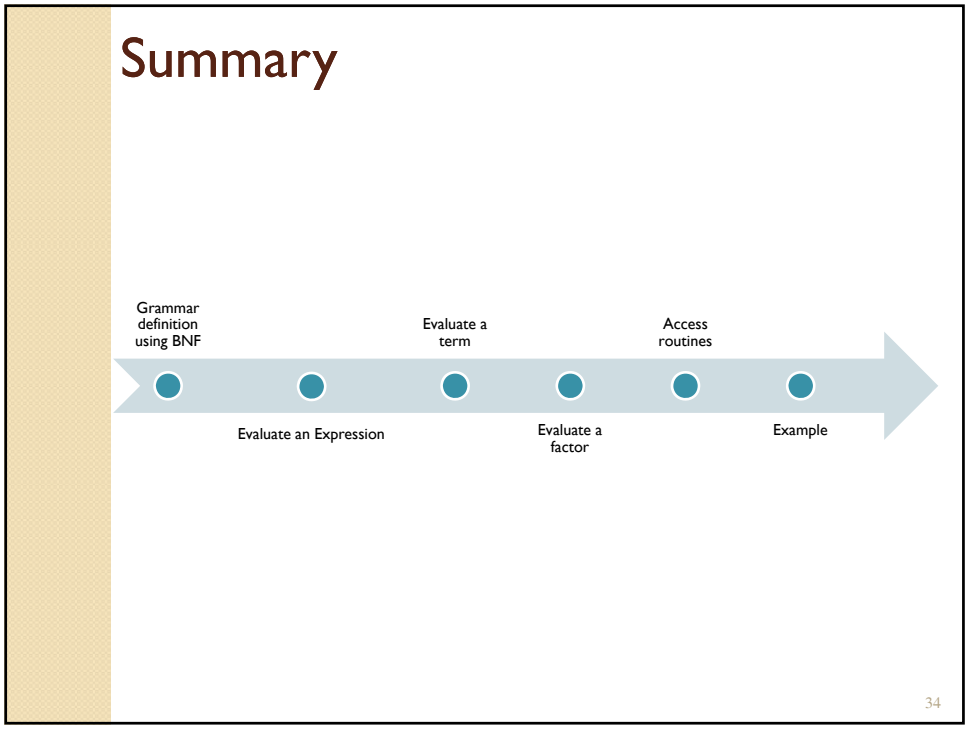

expression
5 + 4)
Result = 5
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
void evalFactor (string& expression, int& result) {
void evalExpression (string& expression, int& result) {
void evalTerm (string& expression, int& result) {
int temp;
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;
}
}
29


op = +
expression
4)
Returns temp = 4
result = 9
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;
}
}
30


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 ')'
}
}
}
31


expression
result = 7
temp = 9
void evalExpression (string& expr sion, t& result) {
void evalTerm (string& expres ion, t& 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;
}
}
32


expression
final
result = 63
void evalExpression (string& exp ssion, 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;
}
}
33


Summary
Grammar definition using BNF
Evaluate an Expression
Evaluate a term
Evaluate a factor
Access routines
Example
34