Expressions & Statements

Michael Nowak

Texas A&M University

Overview

Overview

Introduction

- When writing in English, we
  - put words together into phrases,
  - combine phrases into sentences,
  - compose sentences into paragraphs
- To help you understand programming, we will make analogies between standard English and the components of the C++ programming language
- Such analogies will be not be perfect

Overview

Tokens

- The smallest piece of a programming language that has meaning is called a `token`
- In English, a token is like a word or punctuation mark
- If you change a token in C++, you change its meaning
  - This is similar to breaking up a word
  - can result in something that is no longer a word
    - often without any meaning at all
- Many tokens in C++ are words; others are symbols like punctuation

Overview

Statements

- Let's consider

```
std::cout << "Hello, World!" << std::endl;
```

- In English, putting words together builds sentences
  - A sentence is a grouping that stands on its own in written English
- The equivalence of a sentence in C++ is a `statement`
  - A `statement` is a complete and meaningful command that can be given to a computer
- In C++, a `semicolon` denotes the end of a `statement`
  - In English, we end sentences with a period or some other punctuation mark

Overview

Expressions

- In English, sentences are built from words
- In reality, you build phrases from words and sentences from phrases
- In C++, the equivalent of a phrase is an `expression`
- An `expression` is a group of `tokens` that yields a result when evaluated

Expressions

- In English, some phrases can be made from a single word
- In C++, some `tokens` represent things that have values on their own, and are thus `expression` themselves
  - The simplest form of an `expression` is a single `token` that yields a result when evaluated

Expressions

- In C++, some `tokens` are interpreted as `operands` in an `expression`
- Other `tokens` comprise `operators`
- The simplest form of an `expressions` is thus composed using one or more `operands` that yield a result when evaluated
- More complicated expressions are formed by incorporating an `operator` and one or more `operands`

Overview

Grammars

- Let's consider the English sentence

  I went to the store I got milk and cookies.

- In English, two independent phrases cannot just be joined together without some type of punctuation
- English has a large and complex collection of rules for specifying the syntax of its sentence, known as its `grammar`

Grammars

- Let's consider the C++ `statement`

```
2 2;
```

- This code produces the following error:

```
[cling]$ 2 2;
input_line_3:2:3: error: expected ';' after expression
 2 2;
  ^
  ;
```

- In C++, two independent `operands` cannot just be joined together without an `operator`
- The `statement 2 2;` is invalid in the C++ language
- Programming languages, like the English language, also have `grammars` that dictate which `statements` are valid

Grammars

- `Grammars` define the syntax of our programming language
- To illustrate this concept, let's consider a simple `grammar` for the evaluation of simple arithmetic `statements`

$$
\begin{array}{lll}
\langle \textit{expression} \rangle & ::= & \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '+' } \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '-' } \langle \text{term} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '*' } \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '/' } \langle \text{number} \rangle \\
\\
\langle \textit{number} \rangle & ::= & \text{'floating-point literal'}
\end{array}
$$

Grammars

- An `expression` must be an `term` or end with a `term`

$$
\begin{array}{lll}
\langle \textit{expression} \rangle & ::= & \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '+' } \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '-' } \langle \text{term} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '*' } \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '/' } \langle \text{number} \rangle \\
\\
\langle \textit{number} \rangle & ::= & \text{'floating-point literal'}
\end{array}
$$

Grammars

- A `term` must be a `number` or end with a `number`

$$
\begin{array}{lll}
\langle \textit{expression} \rangle & ::= & \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '+' } \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '-' } \langle \text{term} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '*' } \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '/' } \langle \text{number} \rangle \\
\\
\langle \textit{number} \rangle & ::= & \text{'floating-point literal'}
\end{array}
$$

Grammars

- An `number` must be a `floating-point literal`

$$
\begin{array}{lll}
\langle \textit{expression} \rangle & ::= & \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '+' } \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '-' } \langle \text{term} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '*' } \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '/' } \langle \text{number} \rangle \\
\\
\langle \textit{number} \rangle & ::= & \text{'floating-point literal'}
\end{array}
$$

Grammars

- In the notational convention presented here,
  - The `tokens` are put in single quotes and are called `terminals`
  - The rules are called `non-terminals` or `productions`

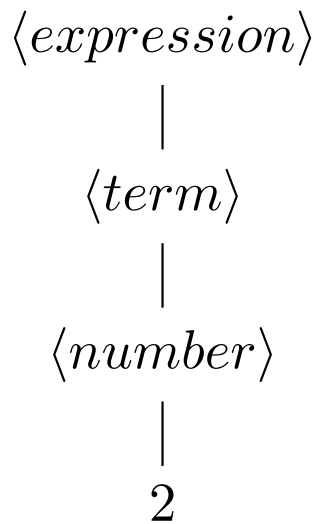
$$
\begin{array}{lll}
\langle \textit{expression} \rangle & ::= & \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '+' } \langle \text{term} \rangle \\
& | & \langle \text{expression} \rangle \text{ '-' } \langle \text{term} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '*' } \langle \text{number} \rangle \\
& | & \langle \text{term} \rangle \text{ '/' } \langle \text{number} \rangle \\
\\
\langle \textit{number} \rangle & ::= & \text{'floating-point literal'}
\end{array}
$$

Grammars

- Given some input, you can read a `grammar` by starting with the "top rule" `expression` and search through the rules to find a match for the `tokens` as they are read
- Reading a sequence of `tokens` according to a `grammar` is known as `parsing`
- A program that does this is often called a `parser` or `syntax analyzer`

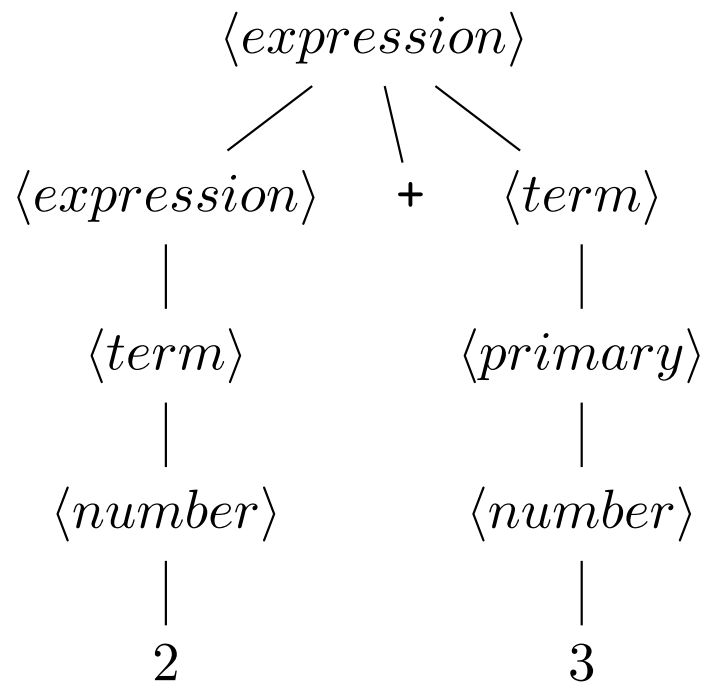
Grammars

- For instance, we would `parse` the number `2` as:



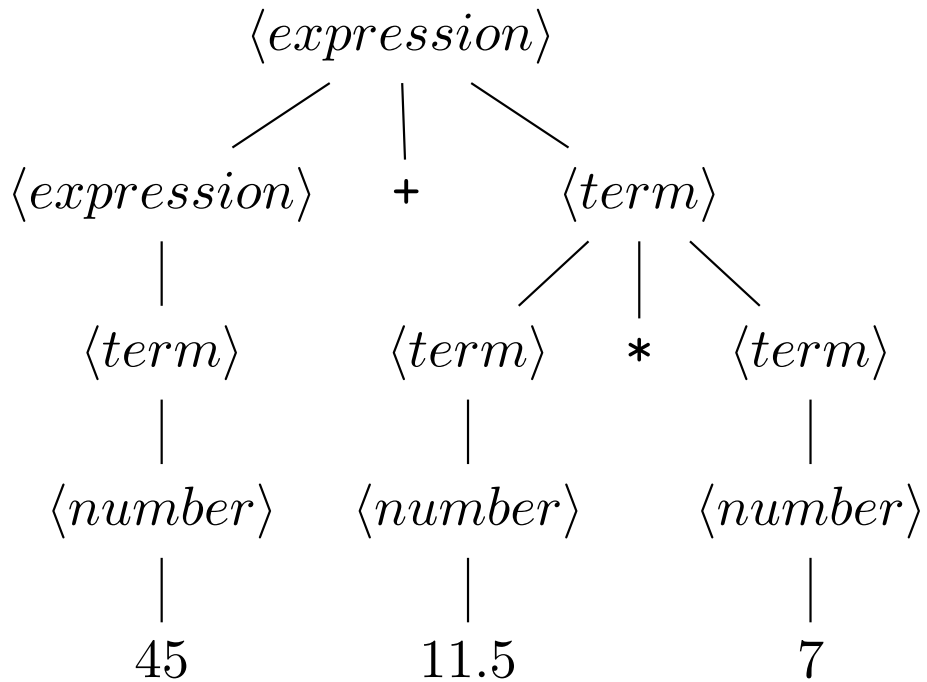
Grammars

- Parsing the expression $2 + 3$ is as easy as:



Grammars

- Parsing the expression $45 + 11.5 * 7$ is as easy as:



Overview

Overview

Overview

Basic concepts

- `Unary operators` act on one `operand`
- `Binary operators` act on two `operands`
- Some tokens are used as both `unary operators` and `binary operators`

Overview

Grouping operators and operands

- An expression with two or more operators is a `compound expression`
- Understanding the evaluation of `compound expressions` requires an understanding of
  - `precedence`
  - `associativity`
  - `order of evaluation`

Overview

Precedence

- Operands of operators with higher precedence group more tightly than those at lower precedence
  - Multiplication and division both have higher precedence than addition and subtraction
  - Multiplication and division group before operands to addition and subtraction

$$3 + 4 * 5 = 23 \text{ not } 35$$

Overview

Associativity

- Associativity determines how operators of the same precedence are grouped
  - Assignment operators are right associative, which means operators at the same precedence group right to left

```
int ival, jval;
ival = jval = 0;
```

  - Arithmetic operators are left associative, which means operators at the same precedence group left to right

$$20 - 15 - 3 = 2 \text{ not } 8$$

Overview

Order of evaluation

- Precedence specifies how the operands are grouped
- Precedence does not specify the order in which the operands are evaluated
- In most cases, the order is largely unspecified
- For example,

$$\text{int i} = \text{f1}() * \text{f2}();$$

- `f1` and `f2` must be called before multiplication can be done
- However, it is unknown whether `f1` will be called before `f2` or vice versa

Overview

Overview

Arithmetic operators (Left Associative)

Operator	Function	Use
`+`	unary plus	`+ expr`
`-`	unary minus	`+ expr`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	remainder	`expr % expr`
`+`	addition	`expr + expr`
`-`	subtraction	`expr - expr`

Overview

Logical and relational operators

Associativity	Operator	Function	Use
Right	`!`	logical `NOT`	`!expr`
Left	`<`	less than	`expr < expr`
Left	`<=`	less than or equal	`expr <= expr`
Left	`>`	greater than	`expr > expr`
Left	`>=`	greater than or equal	`expr >= expr`
Left	`==`	equality	`expr == expr`
Left	`!=`	inequality	`expr != expr`
Left	`&&`	logical and	`expr && expr`
Left	`||`	logical or	`expr || expr`

Overview

Overview

Overview

Simple statements

- Most `statements` in C++ end with a `semicolon`
- An `statements` becomes an `expression statement` when it is followed by a `semicolon`

$$3 + 5;$$

$$std :: cout << (2 + 3);$$

Overview

Null statements

- The simplest `statement` is the `null statement`
- Useful when the language requires a statement, but your logic does not

```
;
```

Overview

Compound statements

- A `compound statement` is usually referred to as a `block`
- It is a (possible empty) sequence of statements and declarations surrounded by a pair of curly braces
- Used when the language requires a single statement, but the logic of our program requires more than one
- `Compound statements` are *not* terminated by a `semicolon`

Overview

Conditional statements

- C++ provides two statements that allow for conditional execution
  - The `if statement`
  - The `switch statement`

Overview

The `if` statement

- An `if statement` conditionally executes another statement based on whether a specified condition is true
- Two forms:
  - `Syntactic` form of the simple `if` is

```
if (condition)
    statement
```

  - An `if else statement` has the form

```
if (condition)
    statement
else
    statement
```

Overview

Iterative statements

- Provide for repeated execution until a condition is true

Overview

`while` statement

- Repeatedly executes a statement as long as a condition is true
- Syntactic form is

```
while (condition)
    statement
```

- In a `while`, the statement (which is often a `block`) is executed as long as `condition` evaluates to `true`
- Usually, the `condition` or the `loop body` must do something to change the value of the expression

`while` statement

- Frequently used when we want to iterate indefinitely, for example
  - While reading input
  - When we need to access the value of the loop `control variable` outside of the loop.

Overview

`for` statement

- Syntactic form is

```
for (init-statement condition; expression)
    statement
```

- The `for` and part inside the parentheses is often referred to as the `for` header
- `init-statement` must be either a declaration statement, an expression statement, or a null statement (each of which end with a `semicolon`)
- The statement (which is often a `block`) is executed as long as `condition` evaluates to `true`
- `expression` is evaluated after each iteration of the loop

`for` statement

- Provided the following `for` loop,

```
for (decltype(s.size()) index = 0; index != s.size();
++index)
    s.at(index) = toupper(s.at(index));
```

  1. `init-statement` is executed once at the start of the loop
  2. Next, the `condition` is evaluated.
     - If it is true, the `loop body` is executed
     - otherwise, the loop terminates
  3. If the `condition` was true, the `statement` is executed
  4. The `expression` is evaluated and we continue from step 2

Overview

do while statement

- Syntactic form is

```
do
    statement
while(condition);
```

- The `do while statement` is like a `while statement`, but has its `condition` tested after the `statement` completes
- Regardless of the value of the condition, the `loop body` is executed at least once
- If `condition` evaluates to false, then the loop terminates; otherwise, the loop is repeated

Overview

References

- Lewis, M. C. (2015). *Introduction to the art of programming using Scala*. CRC Press.
- Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson Education.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Pearson Education.