Concepts of Programming Languages

Lecture 11 - Expressions

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Programming #2 : due 03.21
Homework #3 : due 03.31

Reading:

Chapter 7

Ishmael: Surely all this is not without meaning.

Moby Dick by Herman Melville

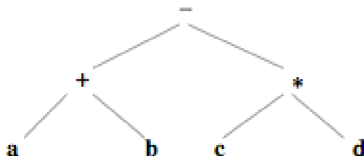
Expressions

Expressions are the fundamental means of specifying computations in a programming language

To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation

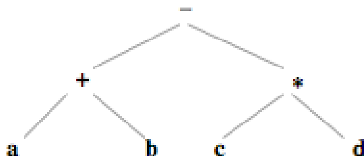
Essence of imperative languages is dominant role of assignment statements

Expression Semantics



How do we represent this expression tree?

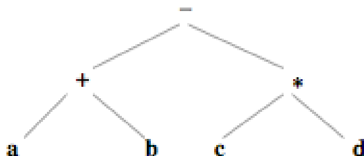
Expression Semantics



How do we represent this expression tree?

- Associative: (a + b) - (c * d)

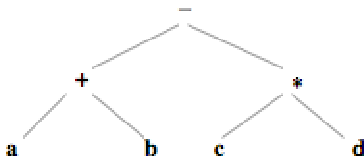
Expression Semantics



How do we represent this expression tree?

- Associative: (a + b) - (c * d)
- Polish Prefix: - + a b * c d

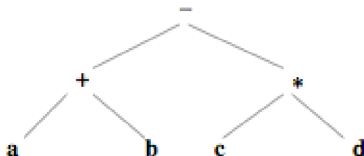
Expression Semantics



How do we represent this expression tree?

- Associative: (a + b) - (c * d)
- Polish Prefix: - + a b * c d
- Polish Postfix: a b + c d * -

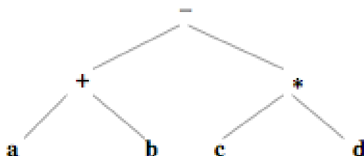
Expression Semantics



How do we represent this expression tree?

- Associative: (a + b) - (c * d)
- Polish Prefix: - + a b * c d
- Polish Postfix: a b + c d * -
- Cambridge Polish: (- (+ a b) (* c d))

Expression Semantics



How do we represent this expression tree?

- Associative: (a + b) - (c * d)
- Polish Prefix: - + a b * c d
- Polish Postfix: a b + c d * -
- Cambridge Polish: (- (+ a b) (* c d))

Arithmetic Expressions

Arithmetic evaluation was one of the motivations for the development of the first programming languages

Arithmetic expressions consist of operators, operands, parentheses, and function calls

Design issues for arithmetic expressions

- Operator precedence rules?
- Operator associativity rules?
- Order of operand evaluation?
- Operand evaluation side effects?
- Operator overloading?
- Type mixing in expressions?

Operator Precedence Rules

Definition

The ***operator precedence rules*** for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated.

Typical precedence levels:

- parentheses
- unary operators
- ** (if the language supports it)
- *, /
- +, -

Precedence of Operators

Operators	C-like	Ada	Fortran
Unary -	7	3	3
**		5	5
* /	6	4	4
+ -	5	3	3
== !=	4	2	2
< <= ...	3	2	2
not	7	2	2

Operator Associativity Rule

Definition

The ***operator associativity rules*** for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.

Typical associativity rules:

- Left to right, except **, which is right to left

Sometimes unary operators associate right to left (e.g., in FORTRAN)

APL is different; all operators have equal precedence and all operators associate right to left

Precedence and associativity rules can be overriden with parentheses

Associativity of Operators

Language	+ - * /	Unary -	**	== != < . . .
C-like	L	R		L
Ada	L	non	non	non
Fortran	L	R	R	L

What is the meaning of a < b < c?

Expressions in Ruby and Scheme

Ruby:

- All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods
- One result of this is that these operators can all be overriden by application programs

Scheme (and Common LISP):

- All arithmetic and logic operations are by explicitly called subprograms
- `a + b * c` is coded as `(+ a (* b c))`

Conditional Expressions

C-based languages (e.g., C, C++)

An example:

```
average = (count == 0)? 0 : sum / count
```

Evaluates as if written as follows:

```
if (count == 0)
    average = 0
else
    average = sum /count
```

Operand Evaluation Order

Operand evaluation order:

1. Variables: fetch the value from memory
2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
3. Parenthesized expressions: evaluate all operands and operators first
4. The most interesting case is when an operand is a function call

Program State

Definition

The ***state*** of a program is the collection of all active objects and their current values.

Maps:

1. The pairing of active objects with specific memory locations,
2. and the pairing of active memory locations with their current values.

Program State

The current statement (portion of an abstract syntax tree) to be executed in a program is interpreted relative to the current state.

The individual steps that occur during a program run can be viewed as a series of state transformations.

For the purposes of this chapter, use only a map from a variable to its value; like a debugger watch window, tied to a particular statement.

Example

```
// compute the factorial of n
1    void main ( ) {
2        int n, i, f;
3        n = 3;
4        i = 1;
5        f = 1;
6        while (i < n) {
7            i = i + 1;
8            f = f * i;
9        }
10   }
```

Example

```
// compute the factorial of n
1   void main ( ) {
2      int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7         i = i + 1;
8         f = f * i;
9      }
10  }
```

n	i	f

Example

```
// compute the factorial of n
1    void main ( ) {
2       int n, i, f;
3       n = 3;
4       i = 1;
5       f = 1;
6       while (i < n) {
7          i = i + 1;
8          f = f * i;
9       }
10   }
```

n	i	f
undef	undef	undef
3	undef	undef

Example

```
// compute the factorial of n
1   void main ( ) {
2      int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7         i = i + 1;
8         f = f * i;
9      }
10  }
```

n	i	f
3	undef	undef
3	1	undef

Example

```
// compute the factorial of n
1    void main ( ) {
2       int n, i, f;
3       n = 3;
4       i = 1;
5       f = 1;
6       while (i < n) {
7          i = i + 1;
8          f = f * i;
9       }
10   }
```

n	i	f
3	1	undef
3	1	1

Example

```
// compute the factorial of n
1   void main ( ) {
2      int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7         i = i + 1;
8         f = f * i;
9      }
10  }
```

n	i	f
3	1	1
3	1	1

Example

```
// compute the factorial of n
1   void main ( ) {
2      int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7         i = i + 1;
8         f = f * i;
9      }
10  }
```

n	i	f
3	1	1
3	2	1

Example

```
// compute the factorial of n
1   void main ( ) {
2      int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7         i = i + 1;
8         f = f * i;
9      }
10  }
```

n	i	f
3	2	1

Example

// compute the factorial of n	n	i	f
1 void main ( ) {			
2 int n, i, f;			
3 n = 3;			
4 i = 1;			
5 f = 1;			
6 while (i < n) {	3	2	2
7 i = i + 1;	3	2	2
8 f = f * i;	3	3	2
9 }	3	3	6
10 }			

Example

	n	i	f
`// compute the factorial of n`			
1 `void main ( ) {`			
2 `int n, i, f;`			
3 `n = 3;`			
4 `i = 1;`			
5 `f = 1;`			
6 `while (i < n) {`			
7 `i = i + 1;`			
8 `f = f * i;`			
9 `}`			
10 `}`	3	3	6

Side Effects

Definition

A ***side effect*** is a change to any non-local variable or I/O.

What is the value of:

```
i = 2; b = 2; c = 5;

a = b * i++ + c * i;
```

Side Effects

A change to any non-local variable or I/O.

What is the value of:

```
i = 2; b = 2; c = 5;

a = b * i++ + c * i;
```

Potentials for Side Effects

Definition

Functional side effects are when a function changes a two-way parameter or a non-local variable.

Problem with functional side effects:

- When a function referenced in an expression alters another operand of the expression;
- e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

Functional Side Effects

Two possible solutions to the problem:

1. Write the language definition to disallow functional side effects
   - No two-way parameters in functions
   - No non-local references in functions
   - **Advantage**: it works!
   - **Disadvantage**: inflexibility of one-way parameters and lack of non-local references
2. Write the language definition to demand that operand evaluation order be fixed
   - **Disadvantage**: limits some compiler optimizations
   - Java requires that operands appear to be evaluated in left-to-right order

Referential Transparency

Definition

A program has the property of ***referential transparency*** if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If fun has no side effects, `result1 = result2`

Otherwise, not, and referential transparency is violated

Referential Transparency

Advantage of referential transparency

- Semantics of a program is much easier to understand if it has referential transparency

Because they do not have variables, programs in pure functional languages are referentially transparent

- Functions cannot have state, which would be stored in local variables
- If a function uses an outside value, it must be a constant (there are no variables). So, the value of a function depends only on its parameters

Overloaded Operators

Use of an operator for more than one purpose is called operator overloading

Some are common (e.g., + for `int` and `float`)

Some are potential trouble (e.g., * in C and C++)

- Loss of compiler error detection (omission of an operand should be a detectable error)
- Some loss of readability

Overloaded Operators

C++, C#, and F# allow user-defined overloaded operators

When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)

Potential problems:

- Users can define nonsense operations
- Readability may suffer, even when the operators make sense

Type Conversions

Definition

A ***narrowing conversion*** is one that converts an object to a type that cannot include all of the values of the original type e.g., `float` to `int`.

Definition

A ***widening conversion*** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`.

Type Conversions: Mixed Mode

Definition

A ***mixed-mode expression*** is one that has operands of different types.

Definition

A ***coercion*** is an implicit type conversion

Disadvantage of coercions:

- They decrease in the type error de1tection ability of the compiler

In most languages, all numeric types are coerced in expressions, using widening conversions

In Ada, there are virtually no coercions in expressions

In ML and F#, there are no coercions in expressions

Explicit Type Conversions

Definition

Explicit type conversions are called ***castings*** in C-based languages.

Examples:

- C: `(int)angle`
- F#: `float(sum)`

Note that F#'s syntax is similar to that of function calls

Errors in Expressions

Causes:

- Inherent limitations of arithmetic — e.g., division by zero
- Limitations of computer arithmetic — e.g. overflow

Often ignored by the run-time system

Relational and Boolean Expressions

Relational Expressions:

- Use relational operators and operands of various types
- Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, ~=, .NE., <>, #)

JavaScript and PHP have two additional relational operator, === and !==

- Similar to their cousins, == and !=, except that they do not coerce their operands
- Ruby uses == for equality relation operator that uses coercions and `eql?` for those that do not

Relational and Boolean Expressions

Boolean Expressions

- Operands are Boolean and the result is Boolean

C89 has no Boolean type–it uses int type with 0 for false and nonzero for true

One odd characteristic of C's expressions: `a < b < c` is a legal expression, but the result is not what you might expect:

- Left operator is evaluated, producing 0 or 1
- The evaluation result is then compared with the third operand (i.e., c)

Short Circuit Evaluation

Definition

Short Circuit Evaluation is an expression in which the result is determined without evaluating all of the operands and/or operators

Example

```
(13 * a) * (b / 13 - 1)
```

If a is zero, there is no need to evaluate `(b /13 - 1)`

Short Circuit Evaluation

`a and b` evaluated as:

```
if a then b else false
```

`a or b` evaluated as:

```
if a then true else b
```

Short Circuit Evaluation Example I

```
Node p = head;
while (p != null && p.info != key)
        p = p.next;
if (p == null) // not in list
        ...
else // found it
        ...
```

Short Circuit Evaluation Example II

```
boolean found = false;
while (p != null && ! found) {
        if (p.info == key)
                found = true;
        else
                p = p.next;
}
```

Short Circuit Evaluation

Problem with non-short-circuit evaluation:

```
index = 0;
while (index <= length) && (LIST[index] != value)
      index++;
```

When `index=length`, `LIST[index]` will cause an indexing problem (assuming `LIST` is `length - 1` long)

Short Circuit Evaluation

C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (&& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)

All logic operators in Ruby, Perl, ML, F#, and Python are short-circuit evaluated

Ada: programmer can specify either (short-circuit is specified with and then and or else)

Short-circuit evaluation exposes the potential problem of side effects in expressions

e.g. `(a > b) || (b++ / 3)`