Concepts of Programming Languages

4th Week

Expressions and Assignments

Prof. Dr. Peter Heusch

Introduction

- Expressions are the fundamental means to specify computations in programs
  - Return a result after evaluation
  - Do not alter machine state (e.g., memory)

- Assignments are executed for their side effects
  - In *imperative programming languages* they alter machine state (think data storage)
  - In *functional programming languages* they define shorthand names for function results

Outline of Class

- Expressions
  - With arithmetic operators
  - With relational and Boolean operators
- Assignments

Arithmetic Expressions

- In the history of programming languages, the formulation of arithmetic expressions was one of the motivations for programming languages
- Arithmetic expressions contain:
  - Variables and constants
  - Operators
  - Function calls
  - Parentheses
- Expression evaluation may cause errors
  - Type, admissibility, representation

Operator Characteristics

The result of arithmetic expressions can differ due to many different criteria:

- Operator precedence
- Operator associativity
- Order of operand evaluation
- Operator side effects
- Operator overloading
- Operator arity (unary, binary, ternary, ...)

Operator Precedence

- Some languages (OCCAM) have no precedence rules at all and require that all expressions be explicitly parenthesized
- Other languages (Assembler) evaluate strictly from left to right
- Using wrong precedence can considerably complicate expressions (Pascal):

  `4 < 5 or 6 < 7` is evaluated as

  `4 < (5 or 6) < 7`

Typical Operator Precedence

- Operator precedence rules define the order in which adjacent operators with different precedent levels are evaluated
  - Adjacent means „separated by one operand“
  -
- Typical precedence levels (as in mathematics)
  - Parentheses
  - Unary operators (sign -, --, ++)
  - ** (if the language supports it)
  - *, /
  - +, -

Operator Characteristics

The result of arithmetic expressions can differ due to many different criteria:

- Operator precedence
- **Operator associativity**
- Order of operand evaluation
- Operator side effects
- Operator overloading
- Operator arity (unary, binary, ternary, ...)

Operator Associativity

- Associativity rules define the order in which adjacent operators **with the same precedence level** are evaluated
- Typical associativity rule: Left to right
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
  - APL: All operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses

Operator Characteristics

The result of arithmetic expressions can differ due to many different criteria:

- Operator precedence
- Operator associativity
- **Order of operand evaluation**
- Operator side effects
- Operator overloading
- Operator arity (unary, binary, ternary, ...)

Operand Evaluation Order

- Parentheses influence the *operator* evaluation order, but not the *operand* evaluation order
- Operand evaluation means determining the value of the operands
  - Fetch variables from memory
  - Evaluate function calls
- If operand evaluation has side effects (e.g., variable values change), evaluation order becomes important
  - No issue for functional languages, since no side effects!

Side Effects

- Simple example:

  `a*b()+c`

  If the call of `b()` changes the value of `c`, the change can take place after the evaluation of `c` – results may be unexpected

- An example in C++:

```
int a = 1;
int f(int &x) {  return x=2*x; }
int g() { return a=22; };
int main(char *args[]) { a=a+f(a)+g();}
```

Avoiding Unexpected Side Effects

- Change the language definition such that side effects are avoided:
  - Functions may not persistently alter parameter or global variables
  - Advantage: It works
  - Disadvantage: It makes programming clumsy
- Change the language definition such that parameter evaluation order is fixed:
  - Advantage: Makes life easier
  - Disadvantage: Hinders some compiler optimizations

Operator Characteristics

The result of arithmetic expressions can differ due to many different criteria:

- Operator precedence
- Operator associativity
- Order of operand evaluation
- Operator side effects
- **Operator overloading**
- Operator arity (unary, binary, ternary, ...)

Operator Overloading

- Use of an operator for more than one purpose is called operator overloading
  - Common for some operators (e.g., + for int, float, String)
- Overloading „-“ may cause trouble:
  - Loss of compiler error detection due to arity
  - `a-b` vs. `-b`
  - Can be avoided by introduction of new symbols (e.g., Pascal used `div` for integer division, Perl uses `eq` for string equality)

Operator Overloading (contd.)

- C++ and Ada allow user-defined overloaded operators
- In Ruby, everything is an object (even literals) – therefore, arithmetic operations are methods that can be overridden
- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Operator Definition

- Eiffel allows the creation of new operators
- Potential problems:
  - Complicates Scanner and Parser for Eiffel
  - Affects readability:
    What is the meaning of |@|#?

Operator Characteristics

The result of arithmetic expressions can differ due to many different criteria:

- Operator precedence
- Operator associativity
- Order of operand evaluation
- Operator side effects
- Operator overloading
- **Operator arity (unary, binary, ternary, ...)**

Ternary Operator: Conditional Expressions

- In C, Perl and Java:

```
A = B ? C : D;
if ( B ) A = C else A = D;
```

- Non-ternary alternative in Python:

```
A = (B and C) or D
```

This will fail if , C is false! (which in Python means "", 0, 0.0, None, () or [])

- Similarly, Bash and Perl use && and || :

```
rm /home/somefile || echo "rm failed"
```

Relational and Boolean Operators

- All relational operators compare two values and yield a Boolean value
- Some operators are standard (<, >), some are not (!=, /=, <>, .NE., #)
- Some languages allow using = for assignment and comparison, others use different symbols:
  - Pascal, Ada: := and =
  - C, C++, Java: = and ==
- Some languages prefer symbols (==, !=), others prefer names (and, or)

Boolean Type
Here: In C

- C traditionally had no Boolean type:
  - 0, 0.0 and (void*)0 were interpreted as `false`
  - Everything else was interpreted as `true`
- Therefore, `a < b < c` is legal, since:
  - First `a < b` is evaluated (either 0 or 1)
  - Then the result is compared against `c`
- This was very dangerous especially for floats and doubles, where only the value `0.0` was treated as false

Short Circuit Evaluation

- Sometimes the result of a Boolean expression is known without evaluating all parts
- The expression a && b is false, if a is false, irrespective of the value of b
- Mutatis mutandis the same holds for a || b
- Languages that only evaluate the necessary parts use short circuit (or lazy) evaluation
- Some languages allow both evaluations:
  - Java: a & b vs. a && b
  - Ada: a AND b vs. a AND THEN b

Ternary Operator: Conditional Expressions

- In C, Perl and Java:

```
A = B ? C : D;
if ( B ) A = C else A = D;
```

- Non-ternary alternative in Python:

```
A = (B and C) or D
```

This will fail if , C is false! (which in Python means "", 0, 0.0, None, () or [])

- Similarly, Bash and Perl use && and || :

```
rm /home/somefile || echo "rm failed"
```

Arithmetic Expressions

- In the history of programming languages, the formulation of arithmetic expressions was one of the motivations for programming languages
- Arithmetic expressions contain:
  - Variables and constants
  - Operators
  - Function calls
  - Parentheses
- Expression evaluation may cause errors
  - Type, admissibility, representation

Type Conversions

- When performing arithmetic with different types, values must be converted:
  - 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
  - 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
- Most language require explicit narrowing and provide automatic widening

Overloading and Mixed Mode Expressions

- Mixed-mode expressions have operands of different types
- This requires explicit or implicit (coercion) type conversion because binary operations are never mixed mode
- Automatic narrowing coercions decrease the type error detection ability of the compiler
- Explicit conversions can be implemented:
  - As a feature of the language (C, C++, Ada)
  - As a predefined function reverting to assembler language (Pascal)

Explicit Type Conversion

- Explicit type conversion by call to a predefined function needs no compiler support
- Two variants for type conversion exist in modern languages:
  - Type casting: the target type is written in () in front of the expression (C, C++):
    ```
    float f = (float) 1.0;
    ```
  - Type construction: the conversion syntactically looks like a constructor call (C++, Ada):
    ```
    float f = float(1.0);
    ```

Detecting Arithmetic Errors

- Few programming languages allow the detection of all arithmetic errors:
  - Division by zero – not admissible (nearly all)
  - Overflow, Carry – representation fails (nearly none)

Example: Python Checks for Overflow

- Python uses the „natural“ machine integer type for most arithmetic computations

- All operations with the natural integer type are checked for overflow:
  - Addition needs 3 lines of C-code
  - Multiplication needs 100 lines of C-code

Outline of Class

- Expressions
  - With arithmetic operators
  - With relational and Boolean operators
- Assignments

Assignment

- General syntax:

  ```
  variable assignmentOperator value
  ```

- Which assignment operator?
  - = FORTRAN, BASIC, PL/I, C, C++, Java
  - := ALGOLs, Pascal, Ada. Reason: = can be bad when it is overloaded for the relational operator for equality
- In theory, C++ would allow to overload the = operator for equality testing!

Assignment: Multiple Assignments

- Python allows to assign multiple variables in a single assignment atomically:

```
(a,b) = (c,d)
```

- Eliminates the need for explicit temporary variables:

```
def ggT(a,b):
  while a!=0:
    (a,b) = (b%a,a)
  return b
```

Assignment as an Expression

- In some languages (e.g., Java, Perl and JavaScript), assignment returns the assigned value as a result (like an expression)
- This result can then be used in a comparison
- In combination with using specific data values as a replacement for a Boolean type, errors become harder to detect

  `if (x = y)` vs `if (x==y)`