# Foundations of Programming Languages

## Expressions


Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik


22. Oktober 2014

# Common Expressions

Some typical expressions:

- **Literals**: `1`, `"string"`, `1.3e-7`, ...
- **Names**: `i`, `a_name`, `camelCaseName`, `lisp-name`, ...
- **Composite expressions**:
  - **subprogram calls**: `factorial(7)`, `print(32767 + 1)`, ...
  - **Algebraic expressions** (functional programming)
  - **Lambda expressions** (functional programming): `fn x => x * x`
  - **Tuples**: `(1, 2)`, `(1, "mixed types", 3.14)`, ...
  - **Operator applications**: `1 + 2 * 17 / -x`
  - Type conversions
- Expressions in parentheses: `(1 + 2) * (17 / -x)`

**Availability varies by language**

# Operators

- Most languages provide binary *arithmetic* operators:
  - `+`, `*`, `/`, `-`
- *Arity*: Number of parameters
- *Fixity*:
  - **Prefix:** `-x` (unary negation)
  - **Infix:** `1+3` (binary addition)
  - **Suffix:** `8!` (unary factorial)
- *Associativity*
- *Precedence*

# Precedence

$$1 + 2 * 3 + 4$$

$$1 + (2 * 3) + 4$$

**Multiplication has higher precedence than addition**

- Languages provide precedence tables for binary infix operators. Examples from C:

| | |
|---|---|
| highest | `(`*`expression`*`)` `[...]` `->` `.` |
| | `*` `/` `%` |
| | `+` `-` |
| | `<<` `>>` |
| | `<` `<=` `>=` `>` |
| | `==` `!=` |
| lower | ... |

**Resolves ambiguity across different operators**

# Associativity

$$1 - 2 - 3 - 4$$

- Left-associative:

$$((1 - 2) - 3) - 4$$

- Right-associative:

$$1 - (2 - (3 - 4))$$

**Resolves ambiguity among operators of same precedence**

# Referential Transparency

```
x = (a + 5) + b;
y = (a + 5) + c;
```

Can we simplify this by computing `a + 5` only once?

- Depends on language semantics
  C++: + might be overloaded and print something
- Characterised by *referential transparency*:

  > An expression is *referentially transparent* if we can substitute the expression's evaluation result for the expression without changing the meaning of the program

- *pure functions*: subroutines whose calls are referentially transparent

**Referential transparency is fundamental in functional programming**

# Relational and Boolean Expressions

When is an expression *true*?

- Boolean literals:
  **true**, **false**
- Boolean expressions:
  - Numeric comparisons:
    less-than, greater-than, less-than-or-equal, greater-than-or-equal
  - Combination of boolean expressions:
    and, or, exclusive-or
  - Negation of boolean expression
  - Equality comparison
- Automatic promotion of non-boolean expressions:
  - e.g. C: `NULL` pointer **false**, all other pointers **true**
  - e.g. Python: empty container objects are **false**, nonempty ones **true**

# Equality

```
String s = new String("foo");
return s == "foo";
```

This is *false* in Java and many other languages!

**Equality is a difficult concept!**

- *reference equality*: point to same memory address
- *value equality*: structural match
  - Easy for integers, strings
  - User-defined data structures: user-defined equality
  - Floating points: is $0.000000001 = 0.0$?
- Equality is *usually*
  - symmetric ($a = b$ whenever $b = a$)
  - ransitive ($a = b$ and $b = c$ implies $a = c$)
  - Known exceptions:
    - User-defined equality may be buggy
    - Javascript's == is intransitive by definition

# Short-Circuit evaluation

Consider:

```
(x < 1)  or  isPrimeNumber(x)
```

If `x` $<$ 1, should we call `isPrimeNumber(x)`?

- Result is already known (**true**)
- `isPrimeNumber(x)` might not be referentially transparent:
  Calling vs. not calling makes observable difference
- Language design choice:
  - *Short-circuit boolean operators*:
    If left operand to **and**/**or** determines outcome: Don't evaluate right operand
  - *Non-short-circuit boolean operators*:
    Always evaluate both operands

# Conditional Expression

- Expression that allows choice between two options:
  - *condition* (boolean expression)
  - *then*-branch (picked if *condition* is `true`)
  - *else*-branch (picked if *condition* is `false`)
- Examples:
  - SML:
    ```
    val x = if 2 > 3 then "weird" else "ok"
    ```
  - Python:
    ```
    x = 'weird' if 2 > 3 else 'ok'
    ```
  - C/C++/Java:
    ```
    var x = (2 > 3) ? "weird" : "ok"
    ```
- Also known as *ternary expression* (C family), *functional if*

**Should we evaluate both branches?**

# Summary

- Expressions are a rich part of most languages:
  - Literals, names
  - Composite expressions: subprogram calls, binary operators, conditional expressions, ...
- Binary infix operators are often provided:
  - *precedence* determines which operators bind more tightly
  - *associativity* determines evaluation order at same precedence level
- *Referencial transparency* describes that an expression can be substituted by its own evaluation result without altering observable behaviour
- Boolean expressions allow computing a notion of 'truth'
- Short-circuit operators combine truth values efficiently, skipping evaluation of right-hand-side operand when possible
  - Will affect behaviour if right-hand-side is not referentially transparent