Imperative programming

Expressions

Tamás Kozsik et al

Eötvös Loránd University

October 15, 2022

Table of contents

Examples

```
n + 1

3.14 * r * r

3 * v[0]

x < 3.14

3 * (r1 + r2) == factorial(x)
```

Lexical elements

- Literals
- Operators
- Identifiers
- Braces
- Other signs, e.g.: comma

Syntax of function call

```
<function-call> ::= <identifier>()
                  | <identifier>(<argument-list>)

<argument-list> ::= <expression>
                  | <expression>, <argument-list>
```

```
pi()
factorial(n + m)
min(0, x + y)
```

Operator arity

- Unary, e.g.: `-x`, `c++`
- Binary, e.g.: `x - y`
- Ternary, e.g.: `x < 0 ? 0 : x`

Operator fixity

- Prefix, e.g.: `++c`
- Postfix, e.g.: `c++`
- Infix, e.g.: `x + y`
- Mixfix, e.g.: `x < 0 ? 0 : x`

Meaningful expressions

- Contained identifiers are declared
  (`n + 1` is not meaningful)
- Well typed
  (Make sense in C: `3 + 3.14`, `"hello" + 1`,
  Doesn't make sense in C: `"hello" * 42`)
- Well typed, but still makes no sense
  (e.g.: `1 / 0`)

Pure and impure languages

- Impure (e.g. C)
  - Determining an expression's value
  - Side-effect
- Pure (e.g. Haskell)
  - Determining an expression's value
  - Referential transparency

Rules of evaluation

- Fully bracketed expression
  `3 + ((12 - 3) * 4)`
- Precedence: `*` binds stronger than +
  12 - 3 * 4
- Left and right associativity
  - In case of operators with the same precedence
  - `3 * n / 2` means `(3 * n) / 2` (left assoc. op.)
  - `n = m = 1` means `n = (m = 1)` (right assoc. op.)

Assignment

Assignment statement

`n = 1;`

Expression with side-effect

`n = 1`

Value of expression with side-effect

`(n = 1)` the value is 1

Value propagation

`m = (n = 1)`

Side-effects

```
printf("%d", n)
n = 1
i *= j
i++
++i
```

Increment/decrement operators

```
int n = 5;
int i;
i = ++n;

n == 6
i == 6
```

```
int m = 5;
int j;
j = m++;

m == 6
j == 5
```

Meaning of expressions

- “Normal” value
- Runtime error, e.g.: `5 / 0` in many languages
- Infinite computation

Lazyness, greedyness

- Greedy: expression in form of `A + B`
- Lazy: expression in form of `A && B` (furthermore: `||`, and `?:`)

```
int f() { printf("f"); return 1; }
int g() { printf("g"); return 2; }

int  i  = f() + g();
/* Output: fg, Result: i == 3 */

bool b1 = (f() == 0) && (g() == 2);
/* Output: f, Result: b1 == false */

bool b2 = (f() == 1) || (g() == 3);
/* Output: f, Value: b2 == true */
```

Result of lazy and greedy "and" expression

Let $\uparrow$, $\downarrow$, $\perp$ and ∞ denote the four possible results for the evalueation of a logical expression: true, false, exception, non-terminating computation. The value of expression $\alpha \wedge \beta$ depending on the values of α and β:

$\alpha \wedge_{lazy} \beta$	$\beta =\uparrow$	$\beta =\downarrow$	$\beta = \perp$	$\beta = \infty$
$\alpha =\uparrow$	$\uparrow$	$\downarrow$	$\perp$	∞
$\alpha =\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$\alpha = \perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\alpha = \infty$	∞	∞	∞	∞

$\alpha \wedge_{greedy} \beta$	$\beta =\uparrow$	$\beta =\downarrow$	$\beta = \perp$	$\beta = \infty$
$\alpha =\uparrow$	$\uparrow$	$\downarrow$	$\perp$	∞
$\alpha =\downarrow$	$\downarrow$	$\downarrow$	$\perp$	∞
$\alpha = \perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\alpha = \infty$	∞	∞	∞	∞

Example

Find the index of the first negative element of an array

```
/* Right */
for (i = 0; i < LENGTH && array[i] >= 0; ++i);
/* Wrong */
for (i = 0; array[i] >= 0 && i < LENGTH; ++i);
```

Side-effect in the operands of a lazy operator

```
(n = 1) +  (m = 1)
(n = 1) || (m = 1)
```

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

Evaluation order of function parameters and operands

Evaluation order is not specified

```
int f() { printf("f"); return 1; }
int g() { printf("g"); return 2; }

void printSum(int a, int b) { printf("%d\n", a + b); }

printf("%d\n", f() + g());      /* Output: fg3 or gf3 */
printSum(f(), g());             /* Output: fg3 or gf3 */
```

Don't do this!

```
int i = 2;
int j = i-- - --i;
printf("%d\n", j);              /* Output: 2 or 0      */
```

Sequence point

- At the end of the whole expression
- At the end of the evaluation of a function's actual parameters
- After the first parameter of a lazy operator
- At a comma operator

Comma operator

```
<expression> ::= ...
               | <expression>, <expression>
```

- Its result is the same as the last parameter's value
- Low precedence

```
int f() { printf("f"); return 1; }
int g() { printf("g"); return 2; }
int h() { printf("h"); return 3; }

printf("%d\n", (f(), g(), h()));      /* Output: fgh3 */
```