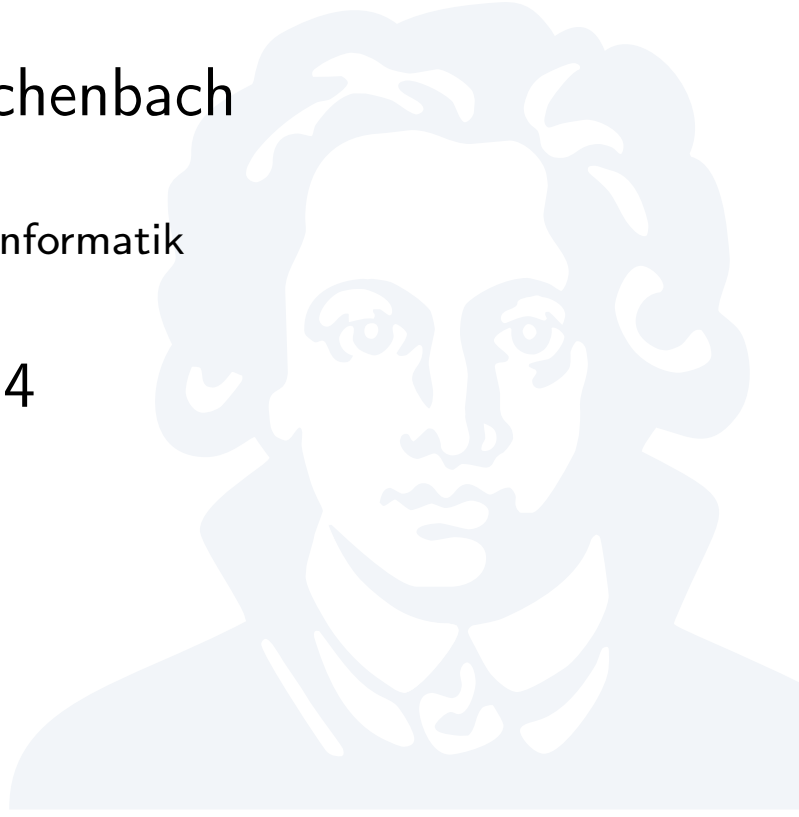
Foundations of Programming Languages

Subprograms

Prof. Dr. Christoph Reichenbach

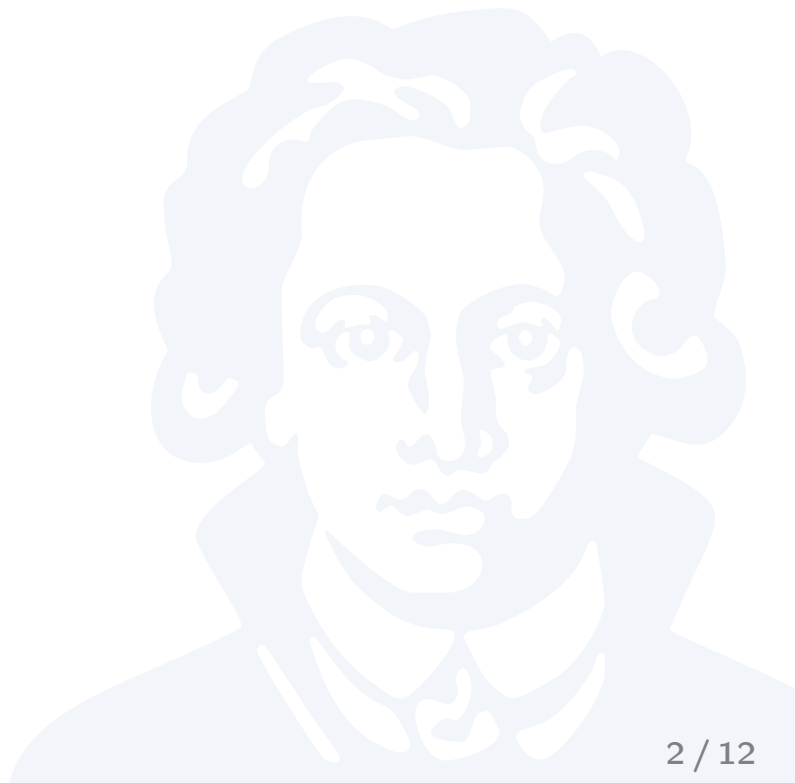
Fachbereich 12 / Institut für Informatik

29. Oktober 2014



Subprograms

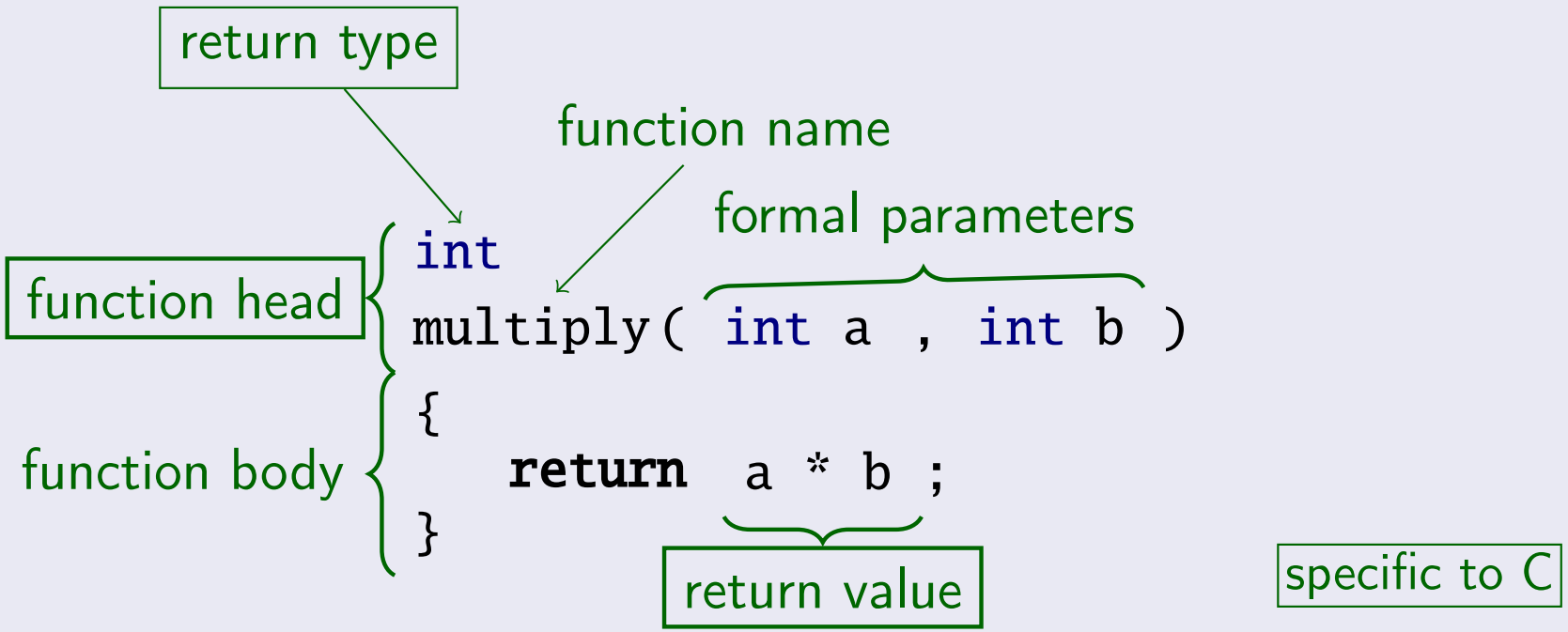
- *Subprograms* make code re-usable
  - Equivalent to subroutines in assembly
- *Examples:*
  - *Functions* (C, Python, Haskell, . . .)
  - *Procedures* (Pascal, SQL, . . .)
  - *Methods* (Smalltalk, Java, . . .)
- Subprograms can be:
  - *defined*
  - *declared*
  - *invoked*



Subprogram Definition

- Subprogram definition binds subprogram *name* to:
  - *body*: Code to execute
  - *formal parameters*: Variables to communicate between callers and bodies
    - have their own bindings, attributes
  - *types* (in some languages)

C



Subprogram Declaration

- Subprogram declaration binds function name to *some* attributes:
  - *number of formal parameters*
  - *types* (in some languages)

C

function head
```
int
multiply( int a , int b );
```

Subprogram Invocation

- Invocation structure:
  - *name*
  - *actual parameters*
- Subprogram invocations can be:
  - *statements*
  - *expressions*, if they return a value

C

```
int  v = multiply  (  100, 32768  );
```

function name

actual parameters

Subprograms and Referential Transparency

C

```c
int multiply(int a, int b)
{
    return a * b;
}
```

C

```c
static int c = 0;
int next()
{
    return c++;
}
```

- Recall *referential transparency*: Expression can be replaced by its evaluation
- Is `multiply(23, 17)` referentially transparent?
  Yes: Same result every time, no side effects
- Is `next()` referentially transparent?
  No: Different result every time, updates memory
- *pure* function: always referentially transparent

Pure functions enable powerful optimisations

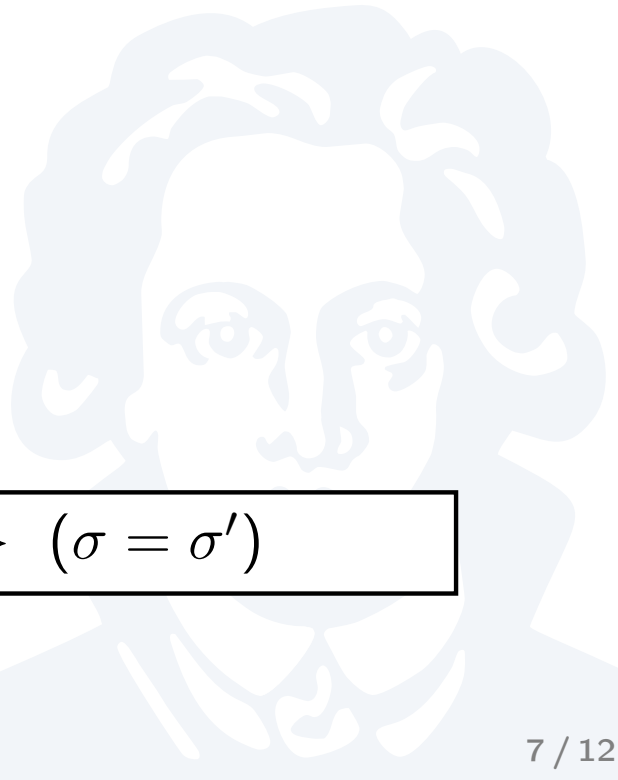
Semantics of Subprograms

Evaluating subprogram calls:

$$\langle f(\ldots)|\sigma\rangle \longrightarrow \langle v|\sigma'\rangle$$

- v: return value
- $\sigma \rightarrow \sigma'$: *side effects*
  - Updates to non-local variables
  - File reads, writes
  - Network access

  $\Rightarrow$ *Any observable input or output*



pure $\iff$ **free of side effects** $\iff$ $(\sigma = \sigma')$

Default Parameters

- Parameters with an additional binding:
  - *default value*
- parameter can be omitted at invocation time; if so, default is substituted

C++

```
int increment(int x, int step = 1)
{
      return x + step;
}

increment(2);   // => 3
increment(3, 2); // => 5
```

Keyword Parameters

- Additional bindings to function:
  - keyword parameter map identifies parameters with names
- Keyword parameters can be passed in any order
- Keyword parameters are often optional

OCaml

```
let roll_dice ?(sides = 6) ?(count = 1) () =
begin
  let result = ref 0 in
  for i = 1 to count do
    result := !result + 1 + Random.int (sides)
  done;
  !result
end;;
roll_dice();;                    (* one six-sided die *)
roll_dice ~count:3 ();;           (* 3 six-sided dice *)
roll_dice ~count:2 ~sides:10 ();; (* 2x 10-sided *)
```

Variable Arguments

- Function can take variable number of arguments
- Typed languages: parameters are all of same type

Java

```
int count(Object ... args)
{
    return args.length;
}

count(1, 2, 3); // => 3
count(0, 0);    // => 2
```

Indirect Subprograms

- Subprograms as parameters
- Subprograms as variables

Python

```
def applyi(f, list):  # f is indirect subprogram
    result = []
    for i in range(0, len(list)):
        result.append(f(list[i], i))
    return result
```

list = [ℓ_0 ℓ_1 ... ℓ_k]

result = [$f(\ell_0, 0)$ $f(\ell_1, 1)$ ... $f(\ell_k, k)$]

```
applyi(multiply, [2, 2, 2, 2])  # => [0, 2, 4, 6]
applyi(max, [1, 1, 1, 1])       # => [1, 1, 2, 3]
```

Summary

- Subprograms allow re-use of functionality
- Can be:
  - *defined*: bind name to prepare for re-use
  - *declared*: partly bind name to enable modular code validity checks
  - *invoked*: reference name to effect re-use
- *Pure* subprograms: always referentially transparent
- Special parameter types:
  - *Default*: Need not be specified, have default value
  - *Keyword*: Specified by name, usually also default
  - *Variable*: Number of parameters is flexible
- Indirect subprograms: Subprogram stored in parameter or value (*first-class functions*)