IA010: Principles of Programming Languages

Expressions

Achim Blumensath
blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Arithmetical expressions

Syntax

$$\langle expr \rangle ::= \langle num \rangle \mid \texttt{(} \langle expr \rangle \texttt{)} \mid \langle expr \rangle \texttt{+} \langle expr \rangle \mid \langle expr \rangle \texttt{*} \langle expr \rangle$$

Evaluation

recursively evaluate subexpressions

```
1+2*3
=>  1+6
=>  7
```

Syntactic sugar

$$expr_1 \texttt{-} expr_2 \quad \Longrightarrow \quad expr_1 \texttt{+} \texttt{(-1)} \texttt{*} expr_2 \,.$$

Local definitions

Syntax

$$\langle expr \rangle \ ::= \ \ldots \ | \ \langle id \rangle \ | \ \mathtt{let}\ \langle id \rangle\ \mathtt{=}\ \langle expr \rangle\ \mathtt{;}\ \langle expr \rangle$$

```
let x = 1;
let y = 2;
x + 2*y
=> 5
```

```
let pi = 3;  // the integer version ;-)
2*pi*5
=> 30
```

```
let x = (let y = 2; 2*y);
x + 3
=> 7
```

```
(let x = 2; x * x) - (let x = 1; x+4)
=> -1
```

Abstraction

naming a program entity

Abstraction

naming a program entity

- can improve readability (good names are essential)
- can improve performance (evaluate expressions only once)
- allows code reuse
- can increase memory burden on programmer
  (without good names)

Scoping

scope: part of the program where a definition is visible

```
let x = e; e′
```

binding: association of **names** with **program entities**

local variables can be **renamed**

```
let x = 2; x*x   ⟺   let y = 2; y*y
```

scopes are usually **nested** (stack implementation)

```
let x = 2;
  let y = x-1;
  x+y
```

} scope of y

} scope of x

Functions

Function definitions

main mechanism for **control abstraction**

- **code reuse**
- improved readability (if used in moderation)

Non-nested functions

```
let f₁(x) { ⟨expr⟩ };
...
let fₙ(x) { ⟨expr⟩ };
⟨expr⟩
```

⇒ function bodies have the empty environment

Functions

Function definitions

main mechanism for **control abstraction**

- **code reuse**
- improved readability (if used in moderation)

Nested functions

$$\langle \mathit{expr} \rangle ::= \dots \mid \langle \mathit{id} \rangle\ \texttt{(}\ \langle \mathit{expr} \rangle\ \texttt{)} \mid \texttt{let}\ \langle \mathit{id} \rangle\ \texttt{(}\ \langle \mathit{id} \rangle\ \texttt{)}\ \texttt{\{}\ \langle \mathit{expr} \rangle\ \texttt{\};}\ \langle \mathit{expr} \rangle$$

⇒ we need to remember the function's environment
(complicates implementation)

Static and dynamic scoping

Static scoping: uses the scope of the function's **definition**

Dynamic scoping: uses the scope of the function's **caller**

```
let x = 1;
let f(y) { x+y };
let x = 2;
f(3)
```

```
let x = 1;
let g(y) { x+y };
let f(y) { g(y) };
let x = 2;
f(3)
```

```
let x = 1;
let g(y) { x+y };
let f(x) { g(0) };
let x = 2;
f(3)
```

Static and dynamic scoping

Dynamic scoping

- used by: original Lisp, Emacs Lisp, TeX, Perl, scripting languages
- generally considered to be a **mistake**
- not **robust:** names of local variables can interfere with other parts of the program!
  - ⇒ **global reasoning** required
  - ⇒ 3rd party libraries need to document local variables
  - ⇒ **security risk:** allows access to local variables from the outside
- allows one to simulate **default parameters**

Static and dynamic scoping

Dynamic scoping

- used by: original Lisp, Emacs Lisp, TeX, Perl, scripting languages
- generally considered to be a **mistake**
- not **robust:** names of local variables can interfere with other parts of the program!
  - ⇒ **global reasoning** required
  - ⇒ 3rd party libraries need to document local variables
  - ⇒ **security risk:** allows access to local variables from the outside
- allows one to simulate **default parameters**

Static scoping

- only sane way to do scoping for large programs
- scoping is tied to syntactic structure
  - ⇒ inflexible, more control desirable (namespaces, modules)

Implementation Issues

How to store the environment?

Implementation Issues

How to store the environment?

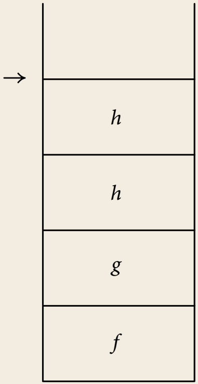
- As scopes are nested we can use a stack.
- Every function stores its local variables in an **activation record.**

Implementation Issues

How to store the environment?

- As scopes are nested we can use a stack.
- Every function stores its local variables in an **activation record.**

```
let f(x) {
  ... g(y) ...
};
let g(x) {
  ... h(y) ...
};
let h(x) {
  ... h(y) ...
};
```

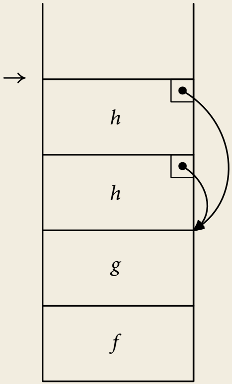


Implementation Issues

How to store the environment?

- For **nested** functions, we need to add a parent pointer.

```
let f(x) {
  ... g(y) ...
};
let g(x) {
  let h(x) {
    ... h(y) ...
  };
  ... h(y) ...
};
```



Higher-order and first-class functions

Higher-order functions: functions that take other functions as arguments

```
let f(x) { x+1 };
let g(s) { s(1) };
g(f)
=> 2
```

First-class functions: functions are ordinary values

$$\langle expr \rangle ::= \ldots \mid \textbf{fun (} \langle id \rangle \textbf{) \{ } \langle expr \rangle \textbf{ \}}$$

```
let adder(n) { fun(x) { x + n } };
let add3 = adder(3);
add3(4)
=> 7
```

Higher-order and first-class functions

Examples (a) decoupling of action and traversal

`map(update, lst)`	applies `update` to every element of `lst`
`fold(sum, 0, lst)`	adds all elements of `lst`

(b) callbacks

```
register_callback(button, pressed)
```

Implementation Issues

- Environments cannot any longer be stored in a stack.

```
let f(x) {
  let z = x + 1;
  fun (x) { x + z }
}
```

Implementation Issues

- Environments cannot any longer be stored in a stack.

```
let f(x) {
  let z = x + 1;
  fun (x) { x + z }
}
```

- The activation records form a **tree.**

Implementation Issues

- Environments cannot any longer be stored in a stack.

```
let f(x) {
  let z = x + 1;
  fun (x) { x + z }
}
```

- The activation records form a **tree.**
- One has to store them on the heap (bad for cache locality).

Function parameters

Currying

```
fun (x,y) { x*x + y*y }
⟹ fun (x) { fun(y) { x*x + y*y }}
```

Function parameters

Currying

```
fun (x,y) { x*x + y*y }
⟹ fun (x) { fun(y) { x*x + y*y }}
```

Keyword parameters

```
let f(serial_number, price, weight) { ... };
f(serial_number = 83927, weight = 60, price = 120);
```

Function parameters

Currying

```
fun (x,y) { x*x + y*y }
⟹ fun (x) { fun(y) { x*x + y*y }}
```

Keyword parameters

```
let f(serial_number, price, weight) { ... };
f(serial_number = 83927, weight = 60, price = 120);
```

Default arguments

```
let int_to_string(num, base = 10) { ... };
int_to_string(17)
```

Function parameters

Currying

```
fun (x,y) { x*x + y*y }
⟹ fun (x) { fun(y) { x*x + y*y }}
```

Keyword parameters

```
let f(serial_number, price, weight) { ... };
f(serial_number = 83927, weight = 60, price = 120);
```

Default arguments

```
let int_to_string(num, base = 10) { ... };
int_to_string(17)
```

Variable number of arguments

```
let printf(format, ...) { ... };
printf("f(%d) = %d", x, f(x));
```

Function parameters

Currying

```
fun (x,y) { x*x + y*y }
⟹ fun (x) { fun(y) { x*x + y*y }}
```

Keyword parameters

```
let f(serial_number, price, weight) { ... };
f(serial_number = 83927, weight = 60, price = 120);
```

Default arguments

```
let int_to_string(num, base = 10) { ... };
int_to_string(17)
```

Variable number of arguments

```
let printf(format, ...) { ... };
printf("f(%d) = %d", x, f(x));
```

Implicit arguments

```
let f(x : int, implicit p : f_param) { ... f(x-1) ... };
```

Conditionals

$$\langle expr \rangle ::= \ldots \mid \texttt{if}\ \langle expr \rangle\ \texttt{==}\ \langle expr \rangle\ \texttt{then}\ \langle expr \rangle\ \texttt{else}\ \langle expr \rangle$$

```
let fac(n) {
  if n == 0 then
    1
  else
    n*fac(n-1)
};
```

Boolean values

- **strict typing:** requires values of type **bool**
- **loose typing:** coercion of **truthy**/**falsy** values
  (convenient, but possibly confusing)

Constructors and pattern matching

Constructors: combine **memory allocation** and **initialisation**

```
Cons(1, Cons(2, Cons(3, Nil)))
```

Built-in constructors

`True`	`()`	`Nil`
`False`	`Pair(`x`,`y`)`	`Cons(`x`,`y`)`

Syntax

⟨*expr*⟩ ::= … | **type** ⟨*id*⟩ **= |** ⟨*variant*⟩ … **|** ⟨*variant*⟩ **;** ⟨*expr*⟩
| **type** ⟨*id*⟩ **= [** ⟨*id*⟩ **=** ⟨*id*⟩ **, … ,** ⟨*id*⟩ **=** ⟨*id*⟩ **];** ⟨*expr*⟩
| ⟨*ctor*⟩ **(** ⟨*expr*⟩ **, … ,** ⟨*expr*⟩ **)**
| **[** ⟨*id*⟩ **=** ⟨*expr*⟩ **, … ,** ⟨*id*⟩ **=** ⟨*expr*⟩ **]**
| ⟨*expr*⟩ **.** ⟨*id*⟩
| **case** ⟨*expr*⟩ **|** ⟨*pattern*⟩ **=>** ⟨*expr*⟩ **|** … **|** ⟨*pattern*⟩ **=>** ⟨

⟨*pattern*⟩ ::= ⟨*id*⟩ | ⟨*num*⟩ | ⟨*ctor*⟩ **(** ⟨*id*⟩ **, … ,** ⟨*id*⟩ **)** | **else**

⟨*variant*⟩ ::= ⟨*id*⟩ | ⟨*id*⟩ **(** ⟨*id*⟩ **, … ,** ⟨*id*⟩ **)**

```
type int_pair = | P(int, int);  type int_pair = [ x : int, y : int ];

let make_pair(x,y) { P(x,y) };  let make_pair(x,y) { [ x = x, y = y ] };

let fst(p) {                    let fst(p) { p.x };
  case p
  | P(x,y) => x
};

let snd(p) {                    let snd(p) { p.y };
  case p
  | P(x,y) => y
};

let empty_list          = Nil;
let add_to_list(x,lst) = Cons(x,lst);

let is_nil(lst)  { case lst | Nil         => True | else => False };
let is_cons(lst) { case lst | Cons(x,xs) => True | else => False };
let head(lst)    { case lst | Cons(x,xs) => x  };
let tail(lst)    { case lst | Cons(x,xs) => xs };
```

Syntactic sugar

if c_0 == c_1 **then** t **else** e	$\Longrightarrow$	**case** c_0 - c_1 | 0 => t | **else** => e
let x = e; e'	$\Longrightarrow$	**case** e | x => e'
e == e'	$\Longrightarrow$	**case** e - e' | 0 => True | **else** => False
if c **then** t **else** e	$\Longrightarrow$	**case** c | True => t | False => e

Recursion

change scope of `x` in `let x =` e`;` e'

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };
let p = (1, q) and q = (2, p);
```

needed to write **non-terminating** programs

Implementation of Recursion

Mutable state

```
let f  = fun (x) { x };        // dummy value
let f' = fun (x) { ... body using f ... }
f := f'
```

Recursion operator

```
rec(f) = f(rec(f))

let fac_body(f) {
  fun (n) { if n == 0 then 1 else n * f(n-1) }
};
let fac = rec(fac_body);
```

Considerations

- Mutual recursion
- Tail calls

Lazy evaluation

```
fun (x) {1+x*x} (1+1)
=> fun (x) {1+x*x} 2
=> 1+2*2
=> 1+4
=> 5
```

```
fun (x) {1+x*x} (1+1)
=> 1+(1+1)*(1+1)
=> 1+2*(1+1)
=> 1+2*2
=> 1+4
=> 5
```

Eager evaluation: starts with left-most, inner-most operation

Lazy evaluation: starts with left-most, outer-most operation

Lazy evaluation

```
fun (x) {1+x*x} (1+1)          fun (x) {1+x*x} (1+1)
=> fun (x) {1+x*x} 2           => 1+(1+1)*(1+1)
=> 1+2*2                       => 1+2*(1+1)
=> 1+4                         => 1+2*2
=> 5                           => 1+4
                               => 5
```

Eager evaluation: starts with left-most, inner-most operation

Lazy evaluation: starts with left-most, outer-most operation

Advantages

- terminates for more programs
- processing infinite data structures
- evaluating recursive definitions

Disadvantages

- cannot be combined with side-effects
- programs hard to understand
- slower than eager evaluation

Implementation Issues

Store either

- the value, if it is already computed,
- a pointer to the function computing it.