Function Programming

Interpreter for a simple imperative language Introduction and Exercise

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

A simple interpreter

To show the power of a functional programming language, we present a prototype for an interpreter for a simple WHILE language.

- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax.

  The interpreter for a simple imperative programming language is a function:

$$I : Program * State \rightarrow State$$

Short presentation of files needed for scanning and parsing:
— input to mosmllex and mosmlyac

succinct programs, fast prototyping

Before lunch

You can read a programs like fact from a file:

```
y:=1 ; while !x=1 do(y:= y*x;x:=x-1)
```

and parse it like:

```
val fact = parsef "factorial.while";
```

Furthermore, you can run programs like:

```
val s = [("x",4)]
I(fact,s);
```

where y is (hopefully) $4! = 24$ in the resulting state.

Arithmetic Expressions

- Abstract syntax for expressions:

```
datatype aExp =     (* arithmetical expressions *)
      N  of int                (* numbers              *)
    | V  of string             (* variables            *)
    | ++ of aExp * aExp        (* addition             *)
    | ** of aExp * aExp        (* multiplication       *)
    | -- of aExp * aExp;       (* subtraction          *)
```

- Infix directives:

```
infix 7 ** ;
infix 6 ++ -- ;
```

Semantics of Arithmetic Expressions

A state associates integers with variables

```
type State = (string * int) list      (* for now *)
```

Operations on the state:

```
update: (string * int * state) -> state
get: string * state -> int
```

The meaning of an expression is a function:

```
A: aExp * State -> int
```

defined inductively on the structure of arithmetic expressions

```
fun A(N n,s)        = n
  | A(V x,s)        = get(x,s)
  | A(a1 ++ a2,s) = A(a1,s) + A(a2,s)
  | A(a1 ** a2,s) = A(a1,s) * A(a2,s)
  | A(a1 -- a2,s) = A(a1,s) - A(a2,s);
```

Boolean Expressions

- Abstract syntax

```
datatype bExp =          (* boolean expressions  *)
           TT                    (* true             *)
         | FF                    (* false            *)
         | == of ....            (* equality         *)
         | << of ....            (* smaller than     *)
         | !! of ....            (* negation         *)
         | && of ....            (* conjunction      *)

infix 4 == << ;
infix 3 && ;
```

- Semantics `B : bExp * State -> bool`

```
fun B(TT, s)          = true
  | B(FF, s)          = false
  ....
```

Statements: Abstract Syntax

```
datatype stm  =
    <- of string * aExp                           (* assignment *)
  | Skip
  | ^^ of stm * stm            (* sequential composition *)
  | ITE   of bExp * stm * stm          (* if-then-else *)
  | While of bExp * stm                             (* while *)

infix 2 <- ;
infix 0 ^^ ;
```

Example of concrete syntax:

```
   y:=1 ; while !(x=1) do (y:= y*x ; x:=x-1)
```

Abstract syntax ?

Interpreter for Statements

- The meaning of statements is a function `I: stm * State -> State` defined by induction on the structure of statements:

```
fun I(x <- a, s)                 = update(x,A(a,s),s)
  | I(Skip, s)                   = ...
  | I(stm1 ^^ stm2, s)           = ...
  | I(ITE(b,stm1,stm2), s)       = ...
  | I(While(b, stm), s)          = ...
```

Example: Factorial function

```
val fact = "y" <-  N 1
           ^^ While(!!(V "x" == N 1),
                      "y" <- V "y" ** V "x"
                      ^^  "x" <-  V "x" -- N 1);

val s = [("x",4)]

val s’ = I(fact, s);

get("y", s’);
> val it = 24 : int
```

Exercises

- Complete the program skeleton (from the homepage) for the interpreter.
- Extend it with if-then and repeat-until statements
- Suppose that an expression of the form $inc(x)$ is added. It adds one to the value of x in the current state, and the value of the expression is this new value of x.
  How should the interpreter be refined to cope with this construct?

Exercises

- Complete the program skeleton (from the homepage) for the interpreter.
- Extend it with if-then and repeat-until statements
- Suppose that an expression of the form $\mathrm{inc}(x)$ is added. It adds one to the value of x in the current state, and the value of the expression is this new value of x.
  How should the interpreter be refined to cope with this construct?

Consider the files for scanning and parsing on the homepage.

- Extend the concrete syntax to deal with some of the above constructs and revise the the input to mosmllex and mosmlyac accordingly. (Only small extensions should be necessary.)