

I217: Functional Programming

8. A Programming Language Processor – Interpreter

Kazuhiro Ogata

Roadmap

- Minila
- Interpreter

Minila

Minila (a Mini-language) is an imperative programming language.

The data types available are natural numbers only.

Some operations of natural numbers, such as addition and multiplication, are available.

Program constructs are assignment statements ($x := e ;$), if statements (**if** $e \{s_1\}$ **else** $\{s_2\}$), while statements (**while** $e \{s\}$), for statements (**for** $i e_1 e_2 \{s\}$), and sequential composition statements ($s_1 s_2$).

Minila

A Minila program that computes the factorial of 10:

```
x := n(1) ;  
y := n(1) ;  
while y < n(10) || y == n(10) {  
  x := x * y ;  
  y := y + n(1) ;  
}
```

A natural number n is expressed as $n(n)$ in Minila

`==` checks if two natural numbers are equal.

As the program terminates, the factorial of 10 is stored in x .

Minila

For the program, the interpreter returns the following:

```
((x , 3628800) | ((y , 11) | empEnv))
```

This is what is called an *environment*. This environment says that 3628800 is stored in x and 11 is stored in y .

The compiler generates the following:

```
push(1) | store(x) | push(1) | store(y) | load(y) | push(10) | lessThan |
load(y) | push(10) | equal | or | jumpOnCond(2) | jump(10) | load(x) |
load(y) | multiply | store(x) | load(y) | push(1) | add | store(y) |
bjump(17) | quit | iln)
```

When the virtual machine executes the list of instructions, returns the same environment as that returned by the interpreter.

Minila

Expressions in Minila are inductively defined as follows:

- ✓ Natural numbers are expressions; Natural numbers in Minila are expressed as $n(0)$, $n(1)$, $n(2)$, ...;
- ✓ Variables are expressions;
- ✓ If e_1 and e_2 are expressions, so are the following:

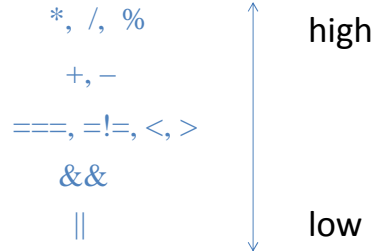
```
 $e_1 + e_2$      $e_1 - e_2$      $e_1 * e_2$      $e_1 / e_2$      $e_1 \% e_2$ 
```

```
 $e_1 === e_2$      $e_1 !== e_2$      $e_1 < e_2$      $e_1 > e_2$ 
```

```
 $e_1 \&\& e_2$      $e_1 \parallel e_2$      $(e_1)$ 
```

Minila

Precedence order of the operators is as follows:



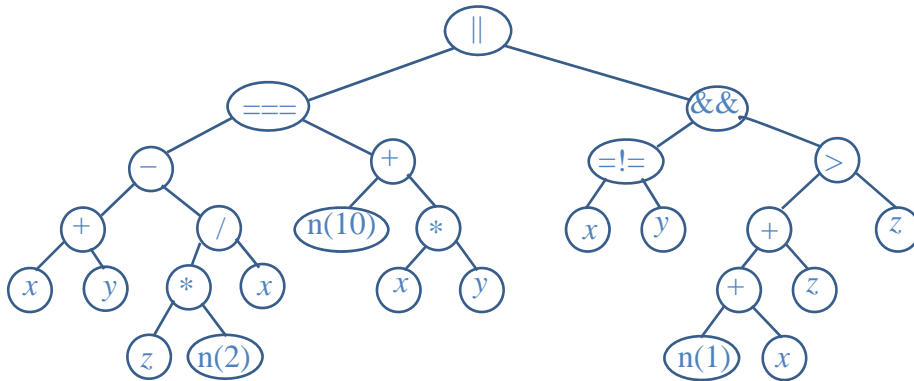
All the operators are left associative.

Minila

$x + y - z * n(2) / x == n(10) + x * y || x != y \&\& n(1) + x + y > z$

is parsed as follows:

$((x + y) - ((z * n(2)) / x)) == (n(10) + (x * y)) || ((x != y) \&\& (((n(1) + x) + y) > z))$



Minila

```
mod* VAR principal-sort Var {  
  [Var]  
}
```

Terms whose sorts are `Var` are variables in Minila.

For example,

```
ops x y z tmp : -> Var .
```

`x`, `y`, `tmp` and `z` are variables in Minila.

Minila

```
mod! EXP { pr(VAR) pr(NAT)  
  [Var < Exp]  
  op n : Nat -> Exp {constr} .  
  op _+_ : Exp Exp -> Exp {constr prec: 33 l-assoc} .  
  op _-_ : Exp Exp -> Exp {constr prec: 33 l-assoc} .  
  op *_ : Exp Exp -> Exp {constr prec: 31 l-assoc} .  
  op _/_ : Exp Exp -> Exp {constr prec: 31 l-assoc} .  
  op _%_ : Exp Exp -> Exp {constr prec: 31 l-assoc} .
```

l-assoc specifies that the binary operator is left associative.

prec: n specifies the precedence of the operator, where n is a natural number. The greater n is, the weaker the precedence of the operator is.

Minila

```

op _===_ : Exp Exp -> Exp {constr prec: 40 l-assoc} .
op _!=_ : Exp Exp -> Exp {constr prec: 40 l-assoc} .
op <_ : Exp Exp -> Exp {constr prec: 40 l-assoc} .
op >_ : Exp Exp -> Exp {constr prec: 40 l-assoc} .
op _&&_ : Exp Exp -> Exp {constr prec: 50 l-assoc} .
op _||_ : Exp Exp -> Exp {constr prec: 55 l-assoc} .
}

```

open EXP .

ops x y z : -> Var .

red x + y - z * n(2) / x === n(10) + x * y || x != y
 && n(1) + x + y > z .

close

Minila

Statements in Minila are inductively defined as follows:

- ✓ **estm** is a statement.
- ✓ If x is a variable and e is an expression, the following is a statement: $x := e ;$
- ✓ If e is an expression and s_1 and s_2 are statements, the following is a statement: **if** e $\{s_1\}$ **else** $\{s_2\}$
- ✓ If e is an expression and s is a statement, the following is a statement: **while** e $\{s\}$
- ✓ If i is a variable, e_1 and e_2 are expressions and s is a statement, the following is a statement: **for** i e_1 e_2 $\{s\}$
- ✓ If s_1 and s_2 are statements, the following is a statement:
 s_1 s_2

Minila

```
mod! STM {  
  pr(EXP)  
  [Stm]  
  op estm : -> Stm { constr } .  
  op _:=_ : Var Exp -> Stm { constr } .  
  op if_{_}else_{_} : Exp Stm Stm -> Stm { constr } .  
  op while_{_} : Exp Stm -> Stm { constr } .  
  op for_{__}_{_} : Var Exp Exp Stm -> Stm { constr } .  
  op __ : Stm Stm -> Stm { constr prec: 60 id: estm l-assoc } .  
}
```

A program in Minila is a statement.

Interpreter

How to interpret (or compute) expressions that may have variables

What is called an *environment* (or *store*) is used.

An environment is a table from the set of variables to values (natural numbers in Minila) stored in those variables.

How to interpret (or compute) a variable is to look up the value (natural number) associated to the variable in a given environment.

Interpreter

Given an expression e that may have variables and an environment env , $evalExp$ interprets (or computes) e with env and returns the result.

op $evalExp : Exp \text{ Env\&Err} \rightarrow Nat\&Err$.

vars $E \ E1 \ E2 : Exp$.

var $V : Var$.

var $EV : Env$.

var $N : Nat$.

eq $evalExp(E, errEnv) = errNat$.

If the environment is an error, then it returns $errNat$ for any expressions.

Interpreter

eq $evalExp(n(N), EV) = N$.

If the expression is a natural number $n(N)$, it returns N regardless of any valid environments.

eq $evalExp(V, EV) = lookup(EV, V)$.

If the expression is a variable V , it looks up the value (natural number) associated with V in the environment and returns the value that may be $errNat$.

eq $evalExp(E1 + E2, EV) = evalExp(E1, EV) + evalExp(E2, EV)$.

If the expression is $E1 + E2$, it interprets $E1$ and $E1$ with the same environment and returns what is obtained by adding the two results.

The result of interpreting $E1$ (or $E2$) with the environment may be $errNat$. If that is the case, the result of the addition is also $errNat$.

Interpreter

```
eq evalExp(E1 === E2,EV)
  = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
    then {errNat}
    else {if evalExp(E1,EV) == evalExp(E2,EV) then {1} else {0}} .
```

If the expression is $E1 === E2$, it interprets $E1$ and $E2$ with the same environment and returns `errNat` if at least one of the results is `errNat`; otherwise it returns `1` if the results are the same and `0` if the results are different.

Note that `0` is used as false and non-zero natural numbers are used as true in Minila.

Interpreter

For the remaining operators, equations can be described likewise for `evalExp`.

Interpreter

Given a program in Minila, `interpret` interprets the program and returns the environment at the time when the program terminates. It may return `errEnv` if something wrong, such as division by zero, happens.

op `interpret : Stm -> Env&Err .`

vars `S S1 S2 : Stm .`

eq `interpret(S) = eval(S,empEnv).`

Given a program in Minila and an environment, `eval` interprets the program with the environment and returns the environment at the time when the program terminates.

Initially, the environment is empty.

Interpreter

op `eval : Stm Env&Err -> Env&Err .`

eq `eval(S,errEnv) = errEnv .`

If the environment is `errEnv`, `eval` returns `errEnv` regardless of the program.

eq `eval(estm,EV) = EV .`

If the program is `estm`, then it returns the environment.

Interpreter

eq $\text{eval}(V := E ;, EV) = \text{evalAssign}(V, \text{evalExp}(E, EV), EV)$.

If the program is $V := E ;$, then it uses `evalAssign` to interpret the program (a single assignment) with the environment.

The expression E is interpreted with the environment.

op `evalAssign` : `Var Nat&Err Env&Err -> Env&Err` .

If the second argument is `errNat` and/or the third argument is `errEnv`, it returns `errEnv`.

Otherwise, it updates the environment by associating the second argument (a natural number) with the first argument (a variable) and returns the updated environment.

Interpreter

eq $\text{eval}(\text{if } E \{S1\} \text{ else } \{S2\}, EV) = \text{evalIf}(\text{evalExp}(E, EV), S1, S2, EV)$.

If the program is `if E {S1} else {S2}`, then it uses `evalIf` to interpret the program (a single if statement) with the environment.

The expression E is interpreted with the environment.

op `evalIf` : `Nat&Err Stm Stm Env&Err -> Env&Err` .

If the first argument is `errNat` and/or the fourth argument is `errEnv`, it returns `errEnv`.

Otherwise, if the first argument is 0, then it interprets the third argument (a statement) with the environment and returns the environment obtained by the interpretation; if the first argument is not 0, then it interprets the second argument (a statement) with the environment and returns the environment obtained by the interpretation.

Interpreter

eq $\text{eval}(\text{while } E \{S1\}, EV) = \text{evalWhile}(E, S1, EV)$.

If the program is `while E {S1}`, then it uses `evalWhile` to interpret the program (a single while statement) with the environment.

op `evalWhile : Exp Stm Env&Err -> Env&Err` .

If the third argument is `errEnv`, it returns `errEnv`.

Otherwise, it interprets the first argument (an expression) with the environment and if the result is `errNat`, then it returns `errEnv`; if the result is 0, then it returns the current environment; if the result is not 0, then it interprets^(*1) the second argument (a statement) with the current environment and interprets^(*2) the while statement with the environment obtained by the interpretation (*1), returning the environment obtained by the interpretation (*2).

Interpreter

`for V E1 E2 {S1}`

is equivalent to

```
V := E1 ;
while V < E2 || V === E2 {
  S1
  V := V + n(1) ;
}
```

This can be used to describe the equation to interpret the `for` statement with a given environment.

eq $\text{eval}(\text{for } V E1 E2 \{S1\}, EV) = \dots$.

Interpreter

How to interpret a sequential composition statement $S1\ S2$ with a given environment is as follows: $S1$ is interpreted with the environment, and then $S2$ is interpreted with the environment obtained by the first interpretation; the environment obtained by the second interpretation is returned as the result.

eq $\text{eval}(S1\ S2, EV) = \dots$

Exercises

1. Complete the interpreter and do some tests for the interpreter.

Appendices

Interpreting the program

```
x := n(1);  
for y n(1) n(10) {  
  x := y * x;  
}
```

returns the environment

```
((x , 3628800) | ((y , 11) | empEnv)):Env
```

Appendices

Interpreting the program

```
x := n(24); y := n(30);  
while y != n(0) {  
  z := x % y; x := y; y := z;  
}
```

returns the environment

```
((x , 6) | ((y , 0) | ((z , 0) | empEnv))):Env
```

Appendices

Interpreting the program

```
x := n(2000000000000000000); y := n(0); z := x;
while y != z {
  if ((z - y) % n(2)) == n(0) {
    tmp := y + (z - y) / n(2);
  } else { tmp := y + ((z - y) / n(2)) + n(1); }
  if tmp * tmp > x { z := tmp - n(1); }
  else { y := tmp; }
}
```

returns the environment

```
((x , 2000000000000000000) | ((y , 141421356) | ((z , 141421356) | ((tmp ,
141421356) | empEnv)))):Env
```