Modelos Formales de Computación

Part 3: Imperative Programming Languages

Máster en Ingeniería y Tecnología de Sistemas Software

Outline

Outline

The objectives

- Know the basic features of an imperative programming language
- Know the basic elements of a state in an imperative language
- Understand the basics of input/output in a programming language
- Introduce the concept of memory and locations as separate notions
- Introduce the concept of name scope in programming languages
- Understand the function call stack and recursive calls
- Introduce the `writeto` and `bindto` operators, useful in the rest of course

General Information

- We define the executable semantics of a very simple imperative programming language, called `SIMPLE`, an abbreviation for the "Simplest IMperative Programming Language on Earth".
- As you read the definition of SIMPLE, keep in mind that it is not intended to be an implementation of the language - that would be the job of an interpreter or a compiler.
- What follows is a semantics of `SIMPLE`, that is, a formal mathematical definition of the language.
- The operations and equations, forming together what is called a specification, represent the totality of properties which if an implementation satisfies, that implementation is considered correct for our language.

Outline

Example Program in SIMPLE

```
global y :
function c(n) {
  local x :
  x = 0 ;
  while (n != 1) {
    x = x + 1 ;
    if (n == 2 * (n / 2))
    then n = n / 2
    else n = 3 * n + 1
  } ;
  return(x)
}
function main() {
  local (n) :
  read(n) ;
  c(n)
}
```

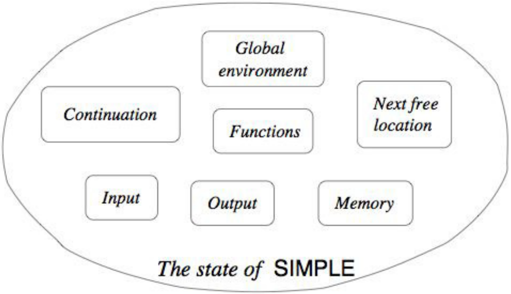
Syntax in SIMPLE

$$
\begin{aligned}
\mathit{Name} &::= \text{standard identifiers} \\
\mathit{NameList} &::= \mathit{Name} \mid \mathit{Name}, \mathit{NameList} \\
\mathit{Exp} &::= \mathit{Name} \mid 1 \mid 2 \mid 3 \mid \ \ldots \ \mid \\
&\quad\ \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid \mathit{Exp} * \mathit{Exp} \mid \mathit{Exp}/\mathit{Exp} \\
\mathit{Exp} &::= \mathit{true} \mid \mathit{false} \mid \mathit{Exp} \leq \mathit{Exp} \mid \mathit{Exp} \geq \mathit{AExp} \mid \mathit{Exp} == \mathit{Exp} \\
&\quad\ \mathit{Exp}\ \mathit{and}\ \mathit{Exp} \mid \mathit{Exp}\ \mathit{or}\ \mathit{Exp} \mid \mathit{not}\ \mathit{Exp} \\
\mathit{Exp} &::= \mathit{skip} \mid \mathit{Name} = \mathit{Exp} \mid \mathit{Exp}\ ;\ \mathit{Exp} \\
\mathit{Exp} &::= \mathit{if}\ \mathit{Exp}\ \mathit{then}\ \mathit{Exp} \mid \mathit{if}\ \mathit{Exp}\ \mathit{then}\ \mathit{Exp}\ \mathit{else}\ \mathit{Exp} \\
\mathit{Exp} &::= \mathit{while}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{for}\ (\mathit{Exp}; \mathit{Exp}; \mathit{Exp})\ \mathit{Exp} \\
\mathit{Exp} &::= \mathit{read}\ \mathit{Name} \mid \mathit{print}\ \mathit{Exp} \\
\mathit{Exp} &::= \{\} \mid \{\mathit{Exp}\} \mid \{\mathit{local}\ \mathit{NameList} : \mathit{Exp}\} \\
\mathit{Exp} &::= \mathit{Name}(\mathit{ExpList}) \mid \mathit{return}\ \mathit{Exp} \\
\mathit{ExpList} &::= \mathit{Exp} \mid \mathit{Exp}, \mathit{ExpList} \\
\mathit{Function} &::= \mathit{function}\ \mathit{Name}(\mathit{NameList})\ \mathit{Exp} \\
\mathit{FunList} &::= \mathit{Function} \mid \mathit{Function}\ \mathit{FunList} \\
\mathit{Pgm} &::= \mathit{FunList} \mid \mathit{global}\ \mathit{NameList}\ \mathit{FunList}
\end{aligned}
$$

Outline

SIMPLE Semantics: State

The following picture shows how the state of a programming language can be regarded, in our approach, as a "soup" of various state attributes. In the case of SIMPLE, there are seven such attributes of its state:



SIMPLE State Infrastructure: Locations and Environments

- In order to define the semantics of a language, we first need to define an appropriate notion of store and environment.
- Locations are needed in order to define environments and stores. We do not want to impose any particular memory/environment architecture and use locations (`loc(0)`, `loc(1)`, `loc(2)`, `...`)
- Environments are defined as mappings of names to locations.

$$
\begin{aligned}
\textit{recover location for a variable } \_[\_] &: \textit{Env Name} \to \textit{Location} \\
((X \mapsto L)\textit{Env})[X] &\Leftarrow L
\end{aligned}
$$

$$
\begin{aligned}
\textit{update environment } \_[\_] &: \textit{Env Bindings} \to \textit{Env} \\
((X' \mapsto L')\textit{Env})[X \mapsto L] &\Leftarrow \textit{if } (X = X') \\
&\quad\ \textit{then } (X' \mapsto L\ \textit{Env}) \\
&\quad\ \textit{else } (X' \mapsto L')(\textit{Env}[X \mapsto L]) \\
\varnothing[X \mapsto L] &\Leftarrow X \mapsto L \\
\textit{Env}[(X, XL) \mapsto (L, LL)] &\Leftarrow (\textit{Env}[X \mapsto L])[Xl \mapsto Ll] \\
\textit{Env}[\textit{nil} \mapsto \textit{nil}] &\Leftarrow \textit{Env}
\end{aligned}
$$

SIMPLE State Infrastructure: Store

- The store contains assignments of values to locations.
- We do not impose any restriction on what values are

$$
\begin{aligned}
\mathit{recover\ value\ for\ a\ location}\ \_[\_] &: \mathit{Store}\ \mathit{Location} \to \mathit{Value} \\
((L \mapsto V)\mathit{Mem})[L] &\Leftarrow V
\end{aligned}
$$

$$
\begin{aligned}
\mathit{update\ store}\ \_[\_] &: \mathit{Store}\ \mathit{Bindings} \to \mathit{Store} \\
([L', V']\mathit{Mem})[L \mapsto V] &\Leftarrow \mathit{if}\ (L = L') \\
&\quad\ \mathit{then}\ (L' \mapsto V)\mathit{Mem} \\
&\quad\ \mathit{else}\ (L' \mapsto V')(\mathit{Mem}[L \mapsto V]) \\
\varnothing[L \mapsto V] &\Leftarrow L \mapsto V \\
\mathit{Mem}[(L, LL) \mapsto (V, VL)] &\Leftarrow \mathit{Mem}(L \mapsto V)[LL \mapsto VL] \\
\mathit{Mem}[\mathit{nil} \mapsto \mathit{nil}] &\Leftarrow \mathit{Mem}
\end{aligned}
$$

Outline

SIMPLE Semantics: Expressions and Values

- Expressions are evaluated into values.
- Expressions are identified using the symbol `exp` as a continuation symbol for the continuation stack
- Different approaches to define expressions. We choose to put together an expression and its environment, so that name scope is always local and there is no confusion
- Values are identified using the symbol `val` as a continuation symbol for the continuation stack. No environment is associated to a value because we choose an eager evaluation semantics
- Auxiliary properties

$$
\begin{aligned}
\mathit{exp}(\mathit{nil}, \mathit{Env}) \curvearrowright K &\leftrightharpoons \mathit{val}(\mathit{nil}) \curvearrowright K \\
\mathit{exp}((E, E', \mathit{El}), \mathit{Env}) \curvearrowright K &\leftrightharpoons \mathit{exp}(E, \mathit{Env}) \curvearrowright \mathit{exp}((E', \mathit{El}), \mathit{Env}) \curvearrowright K \\
\mathit{val}(V) \curvearrowright \mathit{exp}(\mathit{El}, \mathit{Env}) \curvearrowright K &\leftrightharpoons \mathit{exp}(\mathit{El}, \mathit{Env}) \curvearrowright \mathit{val}(V) \curvearrowright K \\
\mathit{val}(\mathit{Vl}) \curvearrowright \mathit{val}(V) \curvearrowright K &\leftrightharpoons \mathit{val}(V, \mathit{Vl}) \curvearrowright K
\end{aligned}
$$

Exercises

- What happens with programs

<pre>global x :
function f() { x := 2 }
function main() {
  x := 0; f(); return(x);
}</pre>
	<pre>global x :
	function f(x) { x := 2 }
	function main() {
	  x := 0; f(x); return(x);
	}</pre>

<pre>global x :
function f() {
    local x : x := 2 }
function main() {
  x := 0; f(); return(x);
}</pre>
	<pre>global x :
	function f(x) {
	  local x : x := 2 }
	function main() {
	  x := 0; f(x); return(x);
	}</pre>

  Which is the execution result?

SIMPLE Semantics: Variables and Constants

$$exp(I, Env) \curvearrowright K \leftrightharpoons val(I) \curvearrowright K$$
$$exp(X, Env) \curvearrowright K \mid Mem \leftrightharpoons val(Mem[Env[X]]) \curvearrowright K \mid Mem$$

SIMPLE Semantics: Arithmetic Expressions

$$
\begin{aligned}
exp(E + E', Env) \curvearrowright K &\leftrightharpoons exp((E, E'), Env) \curvearrowright + \curvearrowright K\\
val(I, I') \curvearrowright + \curvearrowright K &\leftrightharpoons val(I + I') \curvearrowright K\\
exp(E - E', Env) \curvearrowright K &\leftrightharpoons exp((E, E'), Env) \curvearrowright - \curvearrowright K\\
val(I, I') \curvearrowright - \curvearrowright K &\leftrightharpoons val(I - I') \curvearrowright K\\
exp(E * E', Env) \curvearrowright K &\leftrightharpoons exp((E, E'), Env) \curvearrowright * \curvearrowright K\\
val(I, I') \curvearrowright * \curvearrowright K &\leftrightharpoons val(I * I') \curvearrowright K\\
exp(E/E', Env) \curvearrowright K &\leftrightharpoons exp((E, E'), Env) \curvearrowright / \curvearrowright K\\
val(I, I') \curvearrowright / \curvearrowright K &\leftrightharpoons val(I/I') \curvearrowright K
\end{aligned}
$$

Exercises

Which is the meaning of these programs?

- function main() { }
- function main() { 1/0 }

Anything missing?

SIMPLE Semantics: Boolean Expressions

$$exp(true, Env) \curvearrowright K \fallingdotseq val(true) \curvearrowright K$$
$$exp(false, Env) \curvearrowright K \fallingdotseq val(false) \curvearrowright K$$

$$exp(E == E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright == \curvearrowright K \qquad val(I, I') \curvearrowright == \curvearrowright K \fallingdotseq val(I = I') \curvearrowright K$$

$$exp(E\ != E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright\ != \curvearrowright K \qquad val(I, I') \curvearrowright\ != \curvearrowright K \fallingdotseq val(I \neq I') \curvearrowright K$$

$$exp(E <' E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright < \curvearrowright K \qquad val(I, I') \curvearrowright < \curvearrowright K \fallingdotseq val(I < I') \curvearrowright K$$

$$exp(E >' E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright > \curvearrowright K \qquad val(I, I') \curvearrowright > \curvearrowright K \fallingdotseq val(I > I') \curvearrowright K$$

$$exp(E <=' E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright <= \curvearrowright K \qquad val(I, I') \curvearrowright <= \curvearrowright K \fallingdotseq val(I <= I') \curvearrowright K$$

$$exp(E >=' E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright >= \curvearrowright K \qquad val(I, I') \curvearrowright >= \curvearrowright K \fallingdotseq val(I >= I') \curvearrowright K$$

$$exp(E\ and\ E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright and \curvearrowright K \qquad val(B, B') \curvearrowright and \curvearrowright K \fallingdotseq val(B \wedge B') \curvearrowright K$$

$$exp(E\ or\ E', Env) \curvearrowright K \fallingdotseq exp((E, E'), Env) \curvearrowright or \curvearrowright K \qquad val(B, B') \curvearrowright or \curvearrowright K \fallingdotseq val(B \vee B') \curvearrowright K$$

$$exp(not\ E, Env) \curvearrowright K \fallingdotseq exp(E, Env) \curvearrowright not \curvearrowright K \qquad val(B) \curvearrowright not \curvearrowright K \fallingdotseq val(\neg B) \curvearrowright K$$

Exercises

- What happens with the following program

```
function main() {
  if true or (1 / 0 >= 0)
  then 0
  else 1
}
```

Which is the execution result?
What must be modified to obtain 0?

SIMPLE Semantics: Conditionals

Define a conditional expression of the target language in terms of a conditional expression of an implementation language

$$
\begin{aligned}
\mathit{if}\ \ BE\ \ \mathit{then}\ \ E &\leftrightharpoons \mathit{if}\ \ BE\ \ \mathit{then}\ \ E\ \ \mathit{else} \\
\mathit{exp}(\mathit{if}\ \ BE\ \ \mathit{then}\ \ E\ \ \mathit{else}\ \ E', \mathit{Env}) \curvearrowright K &\leftrightharpoons \mathit{exp}(BE, \mathit{Env}) \curvearrowright \mathit{if}(E, E', \mathit{Env}) \curvearrowright K \\
\mathit{val}(B) \curvearrowright \mathit{if}(E, E', \mathit{Env}) \curvearrowright K &\leftrightharpoons \mathit{exp}(\mathit{if}\ \ B\ \ \mathit{then}\ \ E\ \ \mathit{else}\ \ E'\ \ \mathit{fi}, \mathit{Env}) \curvearrowright K
\end{aligned}
$$

SIMPLE Semantics: Assignment

Define an assignment expression in terms of the new `writeTo` construction.
The continuation `writeTo(Location)` allows a unique point of memory modification that can be reused in many situations.

$$exp(X = E, Env) \curvearrowright K \leftrightharpoons exp(E, Env) \curvearrowright writeTo(Env[X]) \curvearrowright val(nothing) \curvearrowright K$$

SIMPLE Semantics: Blocks

Define the semantics of a sequence of instructions in terms of continuations.
We define a new `discard` continuation to avoid reusing returned values.

$$\begin{aligned}
\mathit{exp}((E;E'),\mathit{Env}) \curvearrowright K &\fallingdotseq \mathit{exp}(E,\mathit{Env}) \curvearrowright \mathit{discard} \curvearrowright \mathit{exp}(E',\mathit{Env}) \curvearrowright K\\
\mathit{val}(V) \curvearrowright \mathit{discard} \curvearrowright K &\fallingdotseq K\\
\mathit{exp}(\{\},\mathit{Env}) \curvearrowright K &\fallingdotseq \mathit{val}(\mathit{nothing}) \curvearrowright K\\
\mathit{exp}(\{E\},\mathit{Env}) \curvearrowright K &\fallingdotseq \mathit{exp}(E,\mathit{Env}) \curvearrowright K
\end{aligned}$$

SIMPLE Semantics: Loops

Loops are defined as usual using while unfolding.

$$\textit{while Cond Body} \equiv \textit{if Cond then Body ; while Cond Body else skip}$$

$$\begin{aligned}
\textit{for}(\textit{Start}; \textit{Cond}; \textit{Step})\ \textit{Body} &\leftrightharpoons \textit{Start}\ ;\ \textit{while Cond Body}; \textit{Step} \\
\textit{exp}(\textit{while Cond Body}, \textit{Env}) \curvearrowright K &\leftrightharpoons \textit{exp}(\textit{Cond}, \textit{Env}) \curvearrowright \\
&\ \textit{if}((\textit{Body}; \textit{while Cond Body}), \{\}, \textit{Env}) \curvearrowright K
\end{aligned}$$

Exercises

Which is the meaning of these programs?

- $function\ main()\{\ while\ (true)\ \{\}\ \}$
- $function\ main()\{\ while\ (true)\ \{x = x + 1\}\ \}$

SIMPLE Semantics: Input/Output

Input and Output are defined using streams, as new components of the state.
The expression `read(n)` means reading from the input stream and storing in the variable `n`.
The expression `print(E)` means evaluating the expression `E` into a value (integer or boolean) and storing in the output stream.

$$exp(read(X), Env) \curvearrowright K \mid input(I, IL) \leftrightharpoons val(I) \curvearrowright writeTo(Env[X]) \curvearrowright val(nothing) \curvearrowright K \mid input(IL)$$

$$exp(print(E), Env) \curvearrowright K \leftrightharpoons exp(E, Env) \curvearrowright print \curvearrowright K$$

$$val(V) \curvearrowright print \curvearrowright K \mid output(VL) \leftrightharpoons val(nothing) \curvearrowright K \mid output(VL, V)$$

Exercises

- What happens with programs

`global x :`
`function f() {`
`  x := 2`
`}`
`function main() {`
`  f(); return(x);`
`}`
	`function f() {`
	`  x := 2`
	`}`
	`function main() {`
	`  local x : f(); return(x);`
	`}`

`function f(x) {`
`  x := 2`
`}`
`function main() {`
`  f(0); return(x);`
`}`
	`function f() {`
	`  local x : x := 2`
	`}`
	`function main() {`
	`  f(); return(x);`
	`}`

Which is the execution result?

SIMPLE Semantics: Functions 1/2

- Syntactic sugar

$$function\ F(Xl)\ \{E\} \fallingdotseq function\ F(Xl)\{\ local\ (nil) : E\}$$

- Function call

$$exp(F(El), Env) \curvearrowright K \fallingdotseq exp(El, Env) \curvearrowright apply(F) \curvearrowright K$$

$$\begin{array}{rl} val(Vl) \curvearrowright apply(F) \curvearrowright K \mid & \\ globalEnv(Env) \mid & \\ functions(function\ F(Xl)\ \{local(LXl) : E\} \ldots) \fallingdotseq & val(Vl) \curvearrowright bindTo((Xl, LXl), Env) \curvearrowright \\ & exp*(E) \curvearrowright funcall \curvearrowright K \mid \\ & globalEnv(\ldots) \mid functions(\ldots) \end{array}$$

1. Pop call `F(El)`
2. Evaluate arguments `El` and convert into values `Vl` using continuation `apply(F)`.
3. Look up for function definition "`function F(Xl) {local (LXl) : E}`".
4. Push continuation `funcall` for the semantics of the `return` expression.
5. Create new environment using continuation `bindTo((Xl,LXl), Env)`.
   First global environment, add parameters, add local variables.

SIMPLE Semantics: Functions 2/2

- Function Return

$$
\begin{aligned}
exp(return(E), Env) \curvearrowright K &\leftrightharpoons exp(E, Env) \curvearrowright return \curvearrowright K \\
val(V) \curvearrowright return \curvearrowright Ci \curvearrowright K &\leftrightharpoons if\ \ (Ci = funcall) \\
&\ then\ val(V) \curvearrowright K \\
&\ else\ \ val(V) \curvearrowright return \curvearrowright K \\
val(V) \curvearrowright funcall \curvearrowright K &\leftrightharpoons val(V) \curvearrowright K
\end{aligned}
$$

Search for continuation `funcall` recursively in the continuation stack.

Exercises

- What happens with the following program

```
function f(n) {
  return(0)
}
function main() {
  f(1 / 0);
  return(0);
}
```

Which is the execution result?
What must be modified to obtain `0`?

SIMPLE Semantics: Initial configuration

$$
\begin{aligned}
eval\ Functions\ InputList &\leftrightharpoons eval\ (global\ nil : Functions)\ InputList \\
eval\ (global\ Namelist : Functions)\ InputList &\leftrightharpoons bindTo(NameList, \varnothing) \curvearrowright \\
&\qquad exp * (main()) \curvearrowright stop \mid \\
&\quad nextLoc(1) \mid \varnothing \mid \\
&\quad input(InputList) \mid output(nil) \mid \\
&\quad globalEnv(\varnothing) \mid \\
&\quad functions(Functions) \\
val(VL) \curvearrowright stop \mid & \\
nextLoc(N) \mid Mem \mid & \\
input(IL) \mid output(OL) \mid & \\
globalEnv(Env) \mid functions(Fs) &\leftrightharpoons OL, VL
\end{aligned}
$$

Auxiliary: WriteTo and BindTo and Exp*

- Name scope is extremely important in programming languages, specially in the case of recursive functions.
- Operation writeTo allows easy manipulation of values and locations

$$val(V) \curvearrowright writeTo(L) \curvearrowright K \mid Mem \leftrightharpoons K \mid Mem[L \mapsto V]$$

- Operation bindTo allows easy manipulation of names and locations

$$\begin{aligned} val(V, Vl) \curvearrowright bindTo((X, Xl), Env) \curvearrowright K & \\ \mid Mem \mid nextLoc(N) \leftrightharpoons\ & val(Vl) \curvearrowright \\ & bindTo(Xl, Env[X \mapsto loc(N)]) \curvearrowright K \mid \\ & Mem[loc(N) \mapsto V] \mid nextLoc(N+1) \\ val(nil) \curvearrowright bindTo(Xl, Env) \curvearrowright K \leftrightharpoons\ & bindTo(Xl, Env) \curvearrowright K \\ bindTo((X, Xl), Env) \curvearrowright K \mid nextLoc(N) \leftrightharpoons\ & bindTo(Xl, Env[X \mapsto loc(N)]) \curvearrowright K \mid \\ & nextLoc(N+1) \\ bindTo(nil, Env) \curvearrowright K \leftrightharpoons\ & env(Env) \curvearrowright K \end{aligned}$$

- Operation exp* allows expressions with complex local environments

$$env(Env) \curvearrowright exp*(E) \curvearrowright K \leftrightharpoons exp(E, Env) \curvearrowright K$$

Continuation Semantics Available

- We have implemented the continuation semantics in the programming language Maude
- The implementation is available at the course repository
- Several programs can be executed and the output list is returned

```
Maude> Maude> ==========================================
reduce in TEST-SEMANTICS : eval function main () {local n,x,y,i : n = 50 ; x = 1 ; y = 1 ; for(i = 0 ; i < n ; i = i + 1){x = x + y ; y = x - y} ; print(y) ; return(y)} | nil .
rewrites: 4267 in 0ms cpu (1ms real) (8190019 rewrites/second)
result ValueList: int(20365011074),int(20365011074)
==========================================
reduce in TEST-SEMANTICS : eval global x : function f () {x = x + 1 ; return(0)} function main () {x = 0 ; print(x) ; f () ; print(x)} | nil .
rewrites: 69 in 0ms cpu (0ms real) (~ rewrites/second)
result ValueList: int(0),int(1),nothing
==========================================
reduce in TEST-SEMANTICS : eval function main () {print(max (2,3))} function max x,y {if x > y then return(x) else return(y)} | nil .
rewrites: 59 in 0ms cpu (0ms real) (~ rewrites/second)
result ValueList: int(3),nothing
==========================================
reduce in TEST-SEMANTICS : eval global x,y : function main () {x = max (2,3) ; y = max (4,5) ; print(max (y,max (max (2,max (x,y)),9)))} function max x,y {if x > y then return(x) else return(y)} | 
rewrites: 310 in 0ms cpu (0ms real) (2167832 rewrites/second)
result ValueList: int(9),nothing
==========================================
reduce in TEST-SEMANTICS : eval function h x,y,z,n {if n >= 1 then {h (x,z,y,n - 1) ; print(x) ; print(z) ; h (y,x,z,n - 1)}} function main () {local n : read(n) ; h (1,2,3,n)} | 3 .
rewrites: 1210 in 0ms cpu (0ms real) (~ rewrites/second)
result ValueList: int(1),int(3),int(1),int(2),int(3),int(2),int(1),int(3),int(2),int(1),int(2),int(3),int(1),int(3),nothing
==========================================
reduce in TEST-SEMANTICS : eval function f a {if a <= 1 then return(1) ; return(f (a - 1) + f (a - 2))} function main () {local n : read(n) ; print(f n)} | 10 .
rewrites: 7367 in 0ms cpu (4ms real) (~ rewrites/second)
result ValueList: int(89),nothing
==========================================
reduce in TEST-SEMANTICS : eval function p x,y {local t,i : t = 1 ; for(i = y ; i != 0 ; i = i - 1){t = t * x} ; return(t)} function main () {local x,y : read(x) ; read(y) ; return(p (x,y))} | 10,2
rewrites: 1626 in 0ms cpu (0ms real) (15485714 rewrites/second)
result Value: int(100000000000000000000)
==========================================
reduce in TEST-SEMANTICS : eval function c n {local x : x = 0 ; while n != 1 {x = x + 1 ; if n == 2 * (n / 2) then n = n / 2 else n = 3 * n + 1} ; return(x)} function main () {local n : read(n) ; c
rewrites: 71000 in 20ms cpu (21ms real) (3500813 rewrites/second)
result Value: int(834)
==========================================
reduce in TEST-SEMANTICS : eval function f n {return(0)} function main () {f (1 / 0)} | nil .
rewrites: 25 in 0ms cpu (0ms real) (~ rewrites/second)
result [ValueList]: [k(val (int(1),int(0)) -> / -> apply(f) -> function-call -> stop) nextLoc(0) mem(empty) input(nil) output(nil) globalEnv(empty) functions(function f n {local nil : return(0)} fu
Maude> Bye.
```