

CS 383 Course Project Specification

Version 3.0

Last Modified on May 28, 2013

Course Instructor: Kenny Q. Zhu

Teach Assistant: Tianwan Zhao

Project Target

In this project, you are going to implement an **interpreter** for a simple programming language called **SimPL**. The language is an extension of λ – *calculus* which is the basis of *functional programming* but includes some features *imperative programming* as well. Your interpreter should work like a shell, which takes in an expression and produce correct evaluation results.

This specification will give you the detailed definition for **SimPL**, along with the implementation interfaces.

Language Definition

Preliminaries

We define *key-value pairs* **S** as the form $\{\langle K, V \rangle\}$, which contains a set of pairs $\langle k, v \rangle$ that can map a certain key k to a value v .

We define operator *onion* on *key-value pairs*, denoted \oplus , which means overriding union:

$$S \oplus \langle k, v \rangle := \begin{cases} S \setminus \langle k, u \rangle \cup \langle k, v \rangle & \text{if } \exists u, \langle k, u \rangle \in S \\ S \cup \langle k, v \rangle & \text{otherwise} \end{cases}$$

Let $S(k)$ be the value v if $\langle k, v \rangle$ is contained in S and *null* otherwise.

Syntax

The following is surface syntax of SimPL. A program in SimPL is an expression e :

$e := x$	(* variable *)
$ v$	(* value *)
$ e_1 :: e_2$	(* list *)
$ (e_1, e_2)$	(* pair *)
$ (e_1 \ e_2)$	(* application *)
$ e_1 \ bop \ e_2$	(* binary operations *)
$ uop \ e$	(* unary operations *)
$ let \ x = e_1 \ in \ e_2 \ end$	(* let expression *)
$ if \ e_1 \ then \ e_2 \ else \ e_3 \ end$	(* conditionals *)
$ e_1 := e_2$	(* assignments *)
$ e_1; e_2$	(* sequence *)
$ while \ e_1 \ do \ e_2 \ end$	(* while loop *)
$ fst \ e$	(* first element of a pair *)
$ snd \ e$	(* second element of a pair *)
$ head \ e$	(* head of a list *)
$ tail \ e$	(* tail of a list *)
$ (e)$	(* bracket expression *)
$v := int$	(* integer constant value, including an <i>undef</i> value*)
$ bool$	(* boolean constant value *)
$ nil$	(* empty list *)
$ ()$	(* nop - unit value *)
$ fun \ x \rightarrow e$	(* anonymous function - function is a value *)
$ (v_1, v_2)$	(* pair value *)
$ v_1 :: v_2$	(* list value *)
$bop := \ + \ \ - \ \ * \ \ / \ \ = \ \ > \ \ < \ \ and \ \ or$	(* pay attention to the associative priority *)
$uop := \ \sim \ \ not$	(* unary minus and negation *)

Associative priority (from high to low) of the operators or punctuation characters are divided into 10 levels as listed below:

$(fst, snd, head, tail), (\sim, not), (*, /), (+, -), (=, >, <), (and, or), (::), (:=), (\rightarrow), (;)$

When operators are in the same level, the expression is associated left to right.
Other unmentioned priorities are associated left to right by default.

Names

Identifier

In SimPL, a name is a character string, which must start with a lower case letter followed by a sequence of lower case letters or digits, except those *reserved words*.

These names are allowed:

- *a*, *x1*, *f2b3*
- *foo*, *bar*

And these are not allowed:

- *1a*, *i_d*, *%p*, *\$3*
- *while*, *if*

Scoping

Our SimPL is using **static scoping** rule.

That is to say, when you are opening a new block, you should start a new **environment** ϵ , which is a set of *name to memory location pairs* $\{\langle x, \text{addr}(x) \rangle\}$; and when you are exiting a block, this environment should be eliminated.

In our SimPL, an *anonymous function* is a block, and a *let-expression* is a **block**.

Remember you should have a static area for those global names.

We can get the memory location of name x by $\epsilon(x)$.

Types

SimPL Types

The following is the SimPL types. Every expression e in SimPL has a type t :

$t := \text{int}$	(* integer *)
bool	(* boolean *)
$t_1 * t_2$	(* pair type *)
$t \text{ list}$	(* list of element with type t *)
$t_1 \rightarrow t_2$	(* function type *)
unit	(* imperative type *)

Typing Rules

Γ is a **typing context** (type map) which is a set of hypothesis of the form $x : t$, where $x : t$ means variable x is of type t .

Judgment form $\Gamma \vdash e : t$ is defined that under typing context Γ , expression e is of type t .

Now we give the inductive typing rules as below:

$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}$	T-Value
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 * t_2}$	T-Pair
$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \text{ list}}{\Gamma \vdash e_1 :: e_2 : t \text{ list}}$	T-List
$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \text{fun } x \rightarrow e : t_1 \rightarrow t_2}$	T-Func
$\frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1 \ e_2) : t_2}$	T-App
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$	T-Add
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}}$	T-Sub
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}}$	T-Mul
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 / e_2 : \text{int}}$	T-Div
$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \text{bool}}$	T-EQ
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 > e_2 : \text{bool}}$	T-BT
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 < e_2 : \text{bool}}$	T-LT
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ and } e_2 : \text{bool}}$	T-And
$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ or } e_2 : \text{bool}}$	T-Or

$\frac{\Gamma \vdash e : int}{\Gamma \vdash \sim e : int}$	T-UMi
$\frac{\Gamma \vdash e : bool}{\Gamma \vdash not\ e : bool}$	T-Not
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash let\ x = e_1\ in\ e_2\ end : t_2}$	T-Let
$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3\ end : t}$	T-Cond
$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : unit}$	T-Assign
$\frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1; e_2) : t}$	T-Seq
$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash while\ e_1\ do\ e_2\ end : unit}$	T-While
$\frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash fst\ e : t_1}$	T-Fst
$\frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash snd\ e : t_2}$	T-Snd
$\frac{\Gamma \vdash e : t\ list}{\Gamma \vdash head\ e : t}$	T-Head
$\frac{\Gamma \vdash e : t\ list}{\Gamma \vdash tail\ e : t\ list}$	T-Tail
$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e) : t}$	T-Brac

Type Inference

In SimPL there is no explicit *type declaration*. All the declarations are in the let-expression. You are also supposed to deal with *type inference* which binds a type to the name at a declaration. E.g.

let x = 1 in x end infers x:int

This can enable you to do *type checking*. Please refer to the lecture notes for details.

Memory Model

The **memory model** \mathbf{M} of our SimPL is a set of memory location to value pairs $\{\langle addr, v \rangle\}$.

Make clear that the memory model is different from environment! An environment is a symbol table which maps name to the location.

We can get the value given a memory location a by $\mathbf{M}(addr)$. We can get a newly allocated free address in the memory by calling $alloc()$.

Given a name reference x , to get this names value under current environment, to first check ϵ to get its location, and then fetch the value in \mathbf{M} . That is $\mathbf{M}(\epsilon(x))$.

Semantics

\mathbf{M} is the memory. ϵ is an environment context. e is a SimPL expression.

We define the following judgment form a *single-step reduction* in the program runtime evaluation which changes left hand side state to the right hand side one:

$$\{\epsilon, M, e\} \rightarrow \{\epsilon', M', e'\}$$

We define the following judgment form a *multiple-step reduction* in the program runtime evaluation:

$$\{\epsilon, M, e\} \rightarrow^* \{\epsilon', M', e'\}$$

The inductive definition of *multiple-step reduction* is given below:

$$\frac{\overline{\{\epsilon, M, e\} \rightarrow^* \{\epsilon, M, e\}} \quad \{\epsilon, M, e\} \rightarrow \{\epsilon', M', e'\} \quad \{\epsilon', M', e'\} \rightarrow^* \{\epsilon'', M'', e''\}}{\{\epsilon, M, e\} \rightarrow^* \{\epsilon'', M'', e''\}}$$

Now we give all the semantic rules of SimPL:

$$\frac{\epsilon(x) \neq null \quad M(\epsilon(x)) \neq null}{\{\epsilon, M, x\} \rightarrow \{\epsilon, M, M(\epsilon(x))\}} \quad \text{E-Var}$$

$$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, (e_1, e_2)\} \rightarrow \{\epsilon, M, (e'_1, e_2)\}} \quad \text{E-Pair1}$$

$$\frac{\{\epsilon, M, e_2\} \rightarrow \{\epsilon, M, e'_2\}}{\{\epsilon, M, (v_1, e_2)\} \rightarrow \{\epsilon, M, (v_1, e'_2)\}} \quad \text{E-Pair2}$$

$$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, e_1 :: e_2\} \rightarrow \{\epsilon, M, e'_1 :: e_2\}} \quad \text{E-List1}$$

$$\frac{\{\epsilon, M, e_2\} \rightarrow \{\epsilon, M, e'_2\}}{\{\epsilon, M, v_1 :: e_2\} \rightarrow \{\epsilon, M, v_1 :: e'_2\}} \quad \text{E-List2}$$

$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, (e_1 \ e_2)\} \rightarrow \{\epsilon, M, (e'_1 \ e_2)\}}$	E-App1
$\frac{\{\epsilon, M, e_2\} \rightarrow \{\epsilon, M, e'_2\}}{\{\epsilon, M, (v_1 \ e_2)\} \rightarrow \{\epsilon, M, (v_1 \ e'_2)\}}$	E-App2
$\frac{a = alloc()}{\{\epsilon, M, (fun \ x \rightarrow e \ v)\} \rightarrow \{\epsilon \oplus \langle x, a \rangle, M \oplus \langle a, v \rangle, e\}}$	E-AppAnoFunc
$\frac{\epsilon(x) \neq null \quad M(\epsilon(x)) \neq null}{\{\epsilon, M, (x \ v)\} \rightarrow \{\epsilon, M, (M(\epsilon(x)) \ v)\}}$	E-AppNameFunc
$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, e_1 \ bop \ e_2\} \rightarrow \{\epsilon, M, (e'_1 \ bop \ e_2)\}}$	E-BOp1
$\frac{\{\epsilon, M, e_2\} \rightarrow \{\epsilon, M, e'_2\}}{\{\epsilon, M, v_1 \ bop \ e_2\} \rightarrow \{\epsilon, M, (v_1 \ bop \ e'_2)\}}$	E-BOp2
$\overline{\{\epsilon, M, undef \ bop \ v\} \rightarrow \{\epsilon, M, undef\}}$	E-BOpUd1
$\overline{\{\epsilon, M, v \ bop \ undef\} \rightarrow \{\epsilon, M, undef\}}$	E-BOpUd2
$\frac{v_1 + v_2 = v_3}{\{\epsilon, M, v_1 + v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-Add
$\frac{v_1 - v_2 = v_3}{\{\epsilon, M, v_1 - v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-Sub
$\frac{v_1 * v_2 = v_3}{\{\epsilon, M, v_1 * v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-Mul
$\frac{v_2 = 0}{\{\epsilon, M, v_1/v_2\} \rightarrow \{\epsilon, M, undef\}}$	E-Div0
$\frac{v_1/v_2 = v_3}{\{\epsilon, M, v_1/v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-Div
$\frac{v_1 > v_2}{\{\epsilon, M, v_1 > v_2\} \rightarrow \{\epsilon, M, true\}}$	E-BT
$\frac{v_1 < v_2}{\{\epsilon, M, v_1 < v_2\} \rightarrow \{\epsilon, M, true\}}$	E-LT

$\frac{v_1 = v_2}{\{\epsilon, M, v_1 = v_2\} \rightarrow \{\epsilon, M, true\}}$	E-EQ
$\frac{v_1 \text{ and } v_2 = v_3}{\{\epsilon, M, v_1 \text{ and } v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-And
$\frac{v_1 \text{ or } v_2 = v_3}{\{\epsilon, M, v_1 \text{ or } v_2\} \rightarrow \{\epsilon, M, v_3\}}$	E-Or
$\frac{\{\epsilon, M, e\} \rightarrow \{\epsilon, M, e'\}}{\{\epsilon, M, uop\ e\} \rightarrow \{\epsilon, M, uop\ e'\}}$	E-UOp
$\overline{\{\epsilon, M, true\} \rightarrow \{\epsilon, M, false\}}$	E-NotT
$\overline{\{\epsilon, M, false\} \rightarrow \{\epsilon, M, true\}}$	E-NotF
$\frac{0 - v = v'}{\{\epsilon, M, \sim v\} \rightarrow \{\epsilon, M, v'\}}$	E-UMi
$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, let\ x = e_1\ in\ e_2\ end\} \rightarrow \{\epsilon, M, let\ x = e_1\ in\ e_2\ end\}}$	E-LetDec
$\frac{a = alloc()}{\{\epsilon, M, let\ x = v\ in\ e_2\ end\} \rightarrow \{\epsilon \oplus \langle x, a \rangle, M \oplus \langle a, v \rangle, e_2\}}$	E-LetIn
$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, true\}}{\{\epsilon, M, if\ e_1\ then\ e_2\ else\ e_3\ end\} \rightarrow \{\epsilon, M, e_2\}}$	E-IfT
$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, false\}}{\{\epsilon, M, if\ e_1\ then\ e_2\ else\ e_3\ end\} \rightarrow \{\epsilon, M, e_3\}}$	E-IfF
$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, e_1 := e_2\} \rightarrow \{\epsilon, M, e'_1 := e_2\}}$	E-Assign1
$\frac{\{\epsilon, M, e_2\} \rightarrow \{\epsilon, M, e'_2\}}{\{\epsilon, M, v_1 := e_2\} \rightarrow \{\epsilon, M, v_1 := e'_2\}}$	E-Assign2
$\frac{\epsilon(x) \neq null}{\{\epsilon, M, x := v\} \rightarrow \{\epsilon, M \oplus \langle x, v \rangle, ()\}}$	E-Assign

$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e'_1\}}{\{\epsilon, M, e_1; e_2\} \rightarrow \{\epsilon, M, e'_1; e_2\}}$	E-Seq1
$\overline{\{\epsilon, M, (); e_2\} \rightarrow \{\epsilon, M, e_2\}}$	E-Seq2
$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, true\}}{\{\epsilon, M, while\ e_1\ do\ e_2\ end\} \rightarrow \{\epsilon, M, e_2; while\ e_1\ do\ e_2\ end\}}$	E-WhileT
$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, false\}}{\{\epsilon, M, while\ e_1\ do\ e_2\ end\} \rightarrow \{\epsilon, M, ();\}}$	E-WhileF
$\overline{\{\epsilon, M, fst\ (e_1; e_2)\} \rightarrow \{\epsilon, M, e_1\}}$	E-Fst
$\overline{\{\epsilon, M, snd\ (e_1; e_2)\} \rightarrow \{\epsilon, M, e_2\}}$	E-Snd
$\overline{\{\epsilon, M, head\ e_1 :: e_2\} \rightarrow \{\epsilon, M, e_1\}}$	E-Head
$\overline{\{\epsilon, M, head\ nil\} \rightarrow \{\epsilon, M, nil\}}$	E-HeadNil
$\overline{\{\epsilon, M, tail\ e_1 :: e_2\} \rightarrow \{\epsilon, M, e_2\}}$	E-Tail
$\overline{\{\epsilon, M, tail\ nil\} \rightarrow \{\epsilon, M, nil\}}$	E-TailNil
$\overline{\{\epsilon, M, (e)\} \rightarrow \{\epsilon, M, e\}}$	E-Brac

Implementation Details

Developing Language

The developing language is **Java**.

The abstract syntax are given to you by a package of Java classes.

Lexical Aspects

In SimPL, an **identifier** is a character string starting with a lower case letter followed by a sequence of lower case letters or digits, except those *reserved words*.

Whitespace (spaces, tabs, newlines, returns, and formfeeds) or **comments** may appear between tokens and is ignored. A comment begins with */** and ends with **/*. Comments may nest.

The **reserved words** are *fun, let, in, end, if, then, else, while, do, nil, fst, snd, head, tail, and, or, not*.

The **punctuation characters** are `:: , () = := - > + - * / > < ~`

Input and Output

As your interpreter is supposed to work like a *shell*, it takes in a SimPL program, and output either an error or an evaluated value.

An error can be a *syntax error*, a *type error* (compile error), or a *runtime error*.

If the input program does not have any error, it should be evaluate to a value, which will be finally print to the *standard out*.

The out functions are given to you in the java classes. Call the *toString()* methods when you stop evaluation.

You should also support a *file input/output* for batch tests.

Console Control and Standard Out

When your interpreter is launched by calling:

```
java -jar SimPL.jar -s
```

We shall enter a shell like:

```
SimPL>
```

This is the prompt waiting for expression.

Now if we type in an expression (\$ is the end of the program):

```
SimPL> let
      f = fun x -> (if x = 1 then 1 else x * (f (x-1)) end)
    in
      (f 4)
    end
    $
```

You should output to the standard out the correct evaluation result:

```
SimPL> 24
```

File Input and Output

When your interpreter is launched by calling:

```
java -jar SimPL.jar -f sample.spl
```

It takes the content within *sample.spl* as input program (which also ends with a \$), and outputs the evaluation results to the file *sample.rst* in the same directory of the .spl file.

Sample Programs and Test Cases

Error Programs Examples

Syntax error programs:

```
SimPL> if x = 1 then 2
      $
SimPL> Syntax Error!
```

```
SimPL> 1 /*
      $
SimPL> Syntax Error!
```

```
SimPL> let
      x = true
    in
      1
    $
SimPL> Syntax Error!
```

Type error programs:

```
SimPL> let
      x = (1,2)
    in
      head x
    end
    $
SimPL> Type Error!
```

```
SimPL> let
      f = fun x -> fun y -> x + y
    in
      ((f 1) true)
    end
    $
SimPL> Type Error!
```

```
SimPL> while
      true
    do
      0
```

```
        end
      $
SimPL> Type Error!
```

Runtime error programs:

```
SimPL> x
      $
SimPL> Runtime Error!
```

```
SimPL> let
      x = 1
    in
      let
        y = 2
      in
        ()
      end;
      x+y
    end
  $
SimPL> Runtime Error!
```

Correct Program Examples

Factorial

```
SimPL> let
      f = fun x -> (if x = 1 then 1 else x * (f (x-1)) end)
    in
      (f 4)
    end
  $
SimPL> 24
```

GCD

```
SimPL> let
      mod = fun x -> fun y -> x - x / y * y
    in
      let
        gcd = fun x -> fun y ->
          if ((mod x) y) = 0
          then y
          else ((gcd y) ((mod x) y))
          end
      in
        ((gcd 34986) 3087)
      end
```

```
end
$
SimPL> 1029
```