CS 383 Course Project Specification

Version 1.0

Last Modified on March 14, 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 $\lambda - calculus$ 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:

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

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 *)
$\|if\ e_1\ then\ e_2\ else\ e_3$	(* 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 $undef$ value*)
$\|bool$	(* boolean constant value *)
$\|nil$	(* empty list *)
$\|()$	(* nop - unit value *)
$\|fun\ x \to e$	(* anonymous function - function is a value *)
$\|(v_1, v_2)$	(* pair value *)
$\|v_1 :: v_2$	(* list value *)
$bop := \ + \mid - \mid * \mid / \mid = \mid > \mid < \mid and \mid or$	(* pay attention to the associative priority *)
$uop := \ \sim \mid not$	(* unary minus and negation *)

Associative priority (from high to low) of the operators are divided into 5 levels as listed below:

$$(\sim,\ not),\ (*,\ /),\ (+,\ -),\ (=,\ >,\ <),\ (and,\ or)$$

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* $\{\langle x, addr(x)\rangle\}$; 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:

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

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:

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{T-Value}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 * t_2} \qquad \text{T-Pair}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t\ list}{\Gamma \vdash e_1 :: e_2 : t\ list} \qquad \text{T-List}$$

$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash fun\ x \to e : t_1 \to t_2} \qquad \text{T-Func}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2} \qquad \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int} \qquad \text{T-Add}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 - e_2 : int} \qquad \text{T-Sub}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 * e_2 : int} \qquad \text{T-Mul}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 / e_2 : int} \qquad \text{T-Div}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : bool} \qquad \text{T-EQ}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 > e_2 : bool} \qquad \text{T-BT}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 < e_2 : bool} \qquad \text{T-LT}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ and\ e_2 : bool} \qquad \text{T-And}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ or\ e_2 : bool} \qquad \text{T-Or}$$

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash \sim e : int} \quad \text{T-UMi}$$

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash not\ e : bool} \quad \text{T-Not}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash let\ x = e_1\ in\ e_2\ end : t_2} \quad \text{T-Let}$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : t} \quad \text{T-Cond}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : unit} \quad \text{T-Assign}$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash while\ e_1\ do\ e_2\ end : unit} \quad \text{T-While}$$

$$\frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash fst\ e : t_1} \quad \text{T-Fst}$$

$$\frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash snd\ e : t_2} \quad \text{T-Snd}$$

$$\frac{\Gamma \vdash e : t\ list}{\Gamma \vdash head\ e : t} \quad \text{T-Head}$$

$$\frac{\Gamma \vdash e : t\ list}{\Gamma \vdash tail\ e : t\ list} \quad \text{T-Tail}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e) : t} \quad \text{T-Brac}$$

Type Inference

In SimPL there is no explicit *type declaration*.
All the declarations (binding to names) 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 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:

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

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

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

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

$$\frac{}{\{\epsilon, M, e\} \to^* \{\epsilon, M, e\}}$$

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

Now we give all the semantic rules of SimPL:

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

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, (e_1, e_2)\} \to \{\epsilon, M, (e_1', e_2)\}} \qquad \text{E-Pair1}$$

$$\frac{\{\epsilon, M, e_2\} \to \{\epsilon, M, e_2'\}}{\{\epsilon, M, (v_1, e_2)\} \to \{\epsilon, M, (v_1, e_2')\}} \qquad \text{E-Pair2}$$

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, e_1 :: e_2\} \to \{\epsilon, M, e_1' :: e_2\}} \qquad \text{E-List1}$$

$$\frac{\{\epsilon, M, e_2\} \to \{\epsilon, M, e_2'\}}{\{\epsilon, M, v_1 :: e_2\} \to \{\epsilon, M, v_1 :: e_2'\}} \qquad \text{E-List2}$$

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, (e_1\ e_2)\} \to \{\epsilon, M, (e_1'\ e_2)\}} \quad \text{E-App1}$$

$$\frac{\{\epsilon, M, e_2\} \to \{\epsilon, M, e_2'\}}{\{\epsilon, M, (v_1\ e_2)\} \to \{\epsilon, M, (v_1\ e_2')\}} \quad \text{E-App2}$$

$$\frac{a = alloc()}{\{\epsilon, M, (fun\ x \to e\ v)\} \to \{\epsilon \oplus \langle x, a\rangle, M \oplus \langle a, v\rangle, e\}} \quad \text{E-AppAnoFunc}$$

$$\frac{\epsilon(x) \neq null \quad M(\epsilon(x)) \neq null}{\{\epsilon, M, (x\ v)\} \to \{\epsilon, M, (M(\epsilon(x))\ v)\}} \quad \text{E-AppNameFunc}$$

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, e_1\ bop\ e_2\} \to \{\epsilon, M, (e_1'\ bop\ e_2)\}} \quad \text{E-BOp1}$$

$$\frac{\{\epsilon, M, e_2\} \to \{\epsilon, M, e_2'\}}{\{\epsilon, M, v_1\ bop\ e_2\} \to \{\epsilon, M, (v_1\ bop\ e_2')\}} \quad \text{E-BOp2}$$

$$\frac{}{\{\epsilon, M, undef\ bop\ v\} \to \{\epsilon, M, undef\}} \quad \text{E-BOpUd1}$$

$$\frac{}{\{\epsilon, M, v\ bop\ undef\} \to \{\epsilon, M, undef\}} \quad \text{E-BOpUd2}$$

$$\frac{v_1 + v_2 = v_3}{\{\epsilon, M, v_1 + v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-Add}$$

$$\frac{v_1 - v_2 = v_3}{\{\epsilon, M, v_1 - v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-Sub}$$

$$\frac{v_1 * v_2 = v_3}{\{\epsilon, M, v_1 * v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-Mul}$$

$$\frac{v_2 = 0}{\{\epsilon, M, v_1 / v_2\} \to \{\epsilon, M, undef\}} \quad \text{E-Div0}$$

$$\frac{v_1 / v_2 = v_3}{\{\epsilon, M, v_1 / v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-Div}$$

$$\frac{v_1 > v_2}{\{\epsilon, M, v_1 > v_2\} \to \{\epsilon, M, true\}} \quad \text{E-BT}$$

$$\frac{v_1 < v_2}{\{\epsilon, M, v_1 < v_2\} \to \{\epsilon, M, true\}} \quad \text{E-LT}$$

$$\frac{v_1 = v_2}{\{\epsilon, M, v_1 = v_2\} \to \{\epsilon, M, true\}} \quad \text{E-EQ}$$

$$\frac{v_1 \; and \; v_2 = v_3}{\{\epsilon, M, v_1 \; and \; v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-And}$$

$$\frac{v_1 \; or \; v_2 = v_3}{\{\epsilon, M, v_1 \; or \; v_2\} \to \{\epsilon, M, v_3\}} \quad \text{E-Or}$$

$$\frac{\{\epsilon, M, e\} \to \{\epsilon, M, e'\}}{\{\epsilon, M, uop\; e\} \to \{\epsilon, M, uop\; e')\}} \quad \text{E-UOp}$$

$$\frac{}{\{\epsilon, M, true\} \to \{\epsilon, M, false\}} \quad \text{E-NotT}$$

$$\frac{}{\{\epsilon, M, false\} \to \{\epsilon, M, true\}} \quad \text{E-NotF}$$

$$\frac{0 - v = v'}{\{\epsilon, M, \sim v\} \to \{\epsilon, M, v'\}} \quad \text{E-UMi}$$

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, let\; x = e_1 \; in\; e_2\; end\} \to \{\epsilon, M, let\; x = e_1 \; in\; e_2\; end\}} \quad \text{E-LetDec}$$

$$\frac{a = alloc()}{\{\epsilon, M, let\; x = v\; in\; e_2\; end\} \to \{\epsilon \oplus \langle x, a\rangle, M \oplus \langle a, v\rangle, e_2\}} \quad \text{E-LetIn}$$

$$\frac{\{\epsilon, M, e_1\} \to^* \{\epsilon, M, true\}}{\{\epsilon, M, if\; e_1\; then\; e_2\; else\; e_3\} \to \{\epsilon, M, e_2\}} \quad \text{E-IfT}$$

$$\frac{\{\epsilon, M, e_1\} \to^* \{\epsilon, M, false\}}{\{\epsilon, M, if\; e_1\; then\; e_2\; else\; e_3\} \to \{\epsilon, M, e_3\}} \quad \text{E-IfF}$$

$$\frac{\{\epsilon, M, e_1\} \to \{\epsilon, M, e_1'\}}{\{\epsilon, M, e_1 := e_2\} \to \{\epsilon, M, e_1' := e_2\}} \quad \text{E-Assign1}$$

$$\frac{\{\epsilon, M, e_2\} \to \{\epsilon, M, e_2'\}}{\{\epsilon, M, v_1 := e_2\} \to \{\epsilon, M, v_1 := e_2'\}} \quad \text{E-Assign2}$$

$$\frac{\epsilon(x) \neq null}{\{\epsilon, M, x := v\} \to \{\epsilon, M \oplus \langle x, v\rangle, ()\}} \quad \text{E-Assign}$$

$$\frac{\{\epsilon, M, e_1\} \rightarrow \{\epsilon, M, e_1'\}}{\{\epsilon, M, e_1; e_2\} \rightarrow \{\epsilon, M, e_1'; e_2\}} \quad \text{E-Seq1}$$

$$\frac{}{\{\epsilon, M, (); e_2\} \rightarrow \{\epsilon, M, e_2\}} \quad \text{E-Seq2}$$

$$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, true\}}{\{\epsilon, M, while\ e_1\ do\ e_2\ end\} \rightarrow \{\epsilon, M, e_2; while\ e_1\ do\ e_2\ end\}} \quad \text{E-WhileT}$$

$$\frac{\{\epsilon, M, e_1\} \rightarrow^* \{\epsilon, M, false\}}{\{\epsilon, M, while\ e_1\ do\ e_2\ end\} \rightarrow \{\epsilon, M, ()\}} \quad \text{E-WhileF}$$

$$\frac{}{\{\epsilon, M, fst\ (e_1; e_2)\} \rightarrow \{\epsilon, M, e_1\}} \quad \text{E-Fst}$$

$$\frac{}{\{\epsilon, M, snd\ (e_1; e_2)\} \rightarrow \{\epsilon, M, e_2\}} \quad \text{E-Snd}$$

$$\frac{}{\{\epsilon, M, head\ e_1 :: e_2\} \rightarrow \{\epsilon, M, e_1\}} \quad \text{E-Head}$$

$$\frac{}{\{\epsilon, M, head\ nil\} \rightarrow \{\epsilon, M, nil\}} \quad \text{E-HeadNil}$$

$$\frac{}{\{\epsilon, M, tail\ e_1 :: e_2\} \rightarrow \{\epsilon, M, e_2\}} \quad \text{E-Tail}$$

$$\frac{}{\{\epsilon, M, tail\ nil\} \rightarrow \{\epsilon, M, nil\}} \quad \text{E-TailNil}$$

$$\frac{}{\{\epsilon, M, (e)\} \rightarrow \{\epsilon, M, e\}} \quad \text{E-Brac}$$

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 $::\quad ,\quad (\quad)\quad =\quad :=\quad ->\quad +\quad -\quad *\quad /\quad >\quad <\quad \sim$

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)))
       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)))
        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))
            in
                ((gcd 34986) 3087)
            end
        end
```

```
            $
SimPL>  1029
```