CSE3302-004/CSE5307-001 Course Project Specification

Version 5.0

Last Modified on October 2, 2023

Course Instructor:	Kenny Zhu	`kenny.zhu@uta.edu`
Teaching Assistant:	Sinong Wang	`sinong.wang@uta.edu`

1 Introduction

In this project, you are required to implement an *interpreter* for the programming language SimPL (pronounced *simple*). SimPL is a simplified dialect of ML, which can be used for both *functional* and *imperative* programming. This specification presents a definition of SimPL and provides guidelines to help you implement the interpreter.

2 Lexical Definition

The lexical definition of SimPL consists of four aspects: comments, atoms, keywords, and operators.

2.1 Comments

- Comments in SimPL are enclosed by pairs of `(*` and `*)`.
- Comments are nestable, e.g. `(* (* *) *)` is a valid comment, while `(* (* *)` is invalid.
- A comment can spread over multiple lines.
- Comments and whitespaces (spaces, tabs, newlines) should be ignored and not evaluated.

2.2 Atoms

- Atoms are either integer literals or identifiers.
- Integer literals are matched by regular expression `[0-9]+`.
- Integer literals only represent non-negative integers less than 2^{31}.
- Integer literals are in decimal format, and leading zeros are insignificant, e.g. both `0123` and `000123` represent the integer 123.
- Identifiers are matched by regular expression `[_a-z][_a-zA-Z0-9']*`.

2.3 Keywords

All the following identifiers are keywords. Related keywords are grouped in the same line for better readability. Keywords cannot be bound to anything.

- `nil`
- `ref`

- `fn  rec`
- `let  in  end`
- `if  then  else`
- `while  do`
- `true  false`
- `not  andalso  orelse`

2.4 Operators

- `+  -  *  /  %  ~`
- `=  <>  <  <=  >  >=`
- `::  ()  =>`
- `:=  !`
- `,  ;  (  )`

3 Syntax

- All SimPL programs are expressions. **Exp** is the set of all expressions.
- Names are non-keyword identifiers. **Var** is the set of all names.
- Expressions, names, and integer literals are denoted by meta variables e, x, and n.
- Unary operator $uop \in \{$`~`, `not`, `!`$\}$
- Binary operator $bop \in \{$`+`, `-`, `*`, `/`, `%`, `=`, `<>`, `<`, `<=`, `>`, `>=`, `andalso`, `orelse`, `::`, `:=`, `;`$\}$

Expression e	::=	n	integer literal
	|	x	name
	|	`true`	true value
	|	`false`	false value
	|	`nil`	empty list
	|	`ref` e	reference creation
	|	`fn` x `=>` e	function
	|	`rec` x `=>` e	recursion
	|	`(`e`,` e`)`	pair construction
	|	$uop\ e$	unary operation
	|	$e\ bop\ e$	binary operation
	|	$e\ e$	application
	|	`let` x `=` e `in` e `end`	binding
	|	`if` e `then` e `else` e	conditional
	|	`while` e `do` e	loop
	|	`()`	unit
	|	`(`e`)`	grouping

3.1 Operator Precedence

Priority	Operator(s)	Associativity
1	`;`	left
2	`:=`	none
3	`orelse`	right
4	`andalso`	right
5	`=` `<>` `<` `<=` `>` `>=`	none
6	`::`	right
7	`+` `-`	left
8	`*` `/` `%`	left
9	(application)	left
10	`~` `not` `!`	right

4 Typing

4.1 Definition

$$
\begin{array}{rcl}
\text{Arbitrary type } t & ::= & \texttt{int} \\
 & | & \texttt{bool} \\
 & | & \texttt{unit} \\
 & | & t\,\texttt{list} \\
 & | & t\,\texttt{ref} \\
 & | & t \times t \\
 & | & t \to t
\end{array}
\qquad
\begin{array}{rcl}
\text{Equality type } \alpha & ::= & \texttt{int} \\
 & | & \texttt{bool} \\
 & | & \alpha\,\texttt{list} \\
 & | & t\,\texttt{ref} \\
 & | & \alpha \times \alpha
\end{array}
$$

The set of all types **Typ** is the smallest set satisfying the following rules.

- $\texttt{int} \in \textbf{Typ}$, $\texttt{bool} \in \textbf{Typ}$, $\texttt{unit} \in \textbf{Typ}$
- $\forall t \in \textbf{Typ}$, $t\,\texttt{list} \in \textbf{Typ}$
- $\forall t \in \textbf{Typ}$, $t\,\texttt{ref} \in \textbf{Typ}$
- $\forall t_1, t_2 \in \textbf{Typ}$, $t_1 \times t_2 \in \textbf{Typ}$
- $\forall t_1, t_2 \in \textbf{Typ}$, $t_1 \to t_2 \in \textbf{Typ}$

Type environment $\Gamma : \textbf{Var} \to \textbf{Typ}$ is a mapping from names to arbitrary types. The replacement of x with t in Γ, i.e. $\Gamma[x : t]$, is defined as follows.

$$\Gamma[x : t](y) = \begin{cases} t & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$$

Typing rules:

$$\frac{}{\Gamma \vdash n : \texttt{int}} \quad (\text{T-Int})$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad (\text{T-Name})$$

$$\frac{\Gamma[x : t_1] \vdash e : t_2}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : t_1 \to t_2} \quad (\text{T-Fn})$$

$$\frac{\Gamma[x : t] \vdash e : t}{\Gamma \vdash \texttt{rec}\ x \texttt{=>}\ e : t} \quad (\text{T-Rec})$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \quad (\text{T-True})$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \quad (\text{T-False})$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{\~{}}e : \texttt{int}} \quad (\text{T-Neg})$$

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

$$\frac{}{\Gamma \vdash \texttt{nil} : \alpha\ \texttt{list}} \quad (\text{T-Nil})$$

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}} \quad (\text{T-Unit})$$

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

$$\frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \texttt{;} e_2 : t_2} \quad (\text{T-Seq})$$

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

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

$$\frac{\Gamma \vdash e : t\ \texttt{ref}}{\Gamma \vdash \texttt{!}e : t} \quad (\text{T-Deref})$$

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

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t\ \texttt{list}}{\Gamma \vdash e_1 \texttt{ :: } e_2 : t\ \texttt{list}} \quad (\text{T-Cons})$$

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

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2\ \texttt{end} : t_2} \quad (\text{T-Let})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad bop \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{int}} \quad (\text{T-Arith})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad bop \in \{\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{bool}} \quad (\text{T-Rel})$$

$$\frac{\Gamma \vdash e_1 : \alpha \quad \Gamma \vdash e_2 : \alpha \quad bop \in \{\texttt{=},\texttt{<>}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{bool}} \quad (\text{T-Eq})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \texttt{andalso}\ e_2 : \texttt{bool}} \quad (\text{T-AndAlso})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \texttt{orelse}\ e_2 : \texttt{bool}} \quad (\text{T-OrElse})$$

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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : \texttt{unit}} \quad (\text{T-Loop})$$

4.2 Polymorphism

There is no explicit type annotations in SimPL. So principal type inference is necessary.

1. Some constraint typing rules:

$$\frac{}{\Gamma \vdash n : \texttt{int}, \{\}} \quad \text{(CT-INT)}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t, \{\}} \quad \text{(CT-NAME)}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}, \{\}} \quad \text{(CT-TRUE)}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}, \{\}} \quad \text{(CT-FALSE)}$$

$$\frac{\Gamma \vdash e_1 : t_1, q_1 \quad \Gamma \vdash e_2 : t_2, q_2 \quad bop \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1 \ bop \ e_2 : \texttt{int}, q_1 \cup q_2 \cup \{t_1 = int, t_2 = int\}} \quad \text{(CT-ARITH)}$$

$$\frac{\Gamma \vdash e_1 : t_1, q_1 \quad \Gamma \vdash e_2 : t_2, q_2 \quad bop \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \ bop \ e_2 : \texttt{bool}, q_1 \cup q_2 \cup \{t_1 = int, t_2 = int\}} \quad \text{(CT-REL)}$$

$$\frac{\Gamma, x : a \vdash e : t, q}{\Gamma \vdash \texttt{fn} \ x : a \texttt{=>} e : b : a \to b, q \cup \{t = b\}} \quad \text{(CT-FN)}$$

You need to think of how to deal with other cases.

2. Unification algorithm:

$$\frac{}{(S, \{int = int\} \cup q) -> (S, q)}$$

$$\frac{}{(S, \{bool = bool\} \cup q) -> (S, q)}$$

$$\frac{}{(S, \{a = a\} \cup q) -> (S, q)}$$

$$\frac{}{(S, \{s_{11} -> s_{12} = s_{21} -> s_{22}\} \cup q) -> (S, \{s_{11} = s_{21}, s_{12} = s_{22}\} \cup q)}$$

$$\frac{}{(S, \{a = s\} \cup q) -> ([a = s] \circ S, q[s/a])} (a \ not \ in \ FV(s))$$

$$\frac{}{(S, \{s = a\} \cup q) -> ([a = s] \circ S, q[s/a])} (a \ not \ in \ FV(s))$$

3. Let-Polymorphism is also needed.

$$\frac{\Gamma \vdash e_2[e_1/x] : t_2 \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash \texttt{let} \ x = e_1 \ \texttt{in} \ e_2 \ \texttt{end} : t_2} \quad (T-LetPoly)$$

$$\frac{\Gamma \vdash e_2[e_1/x] : t_2, q_1 \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash \texttt{let} \ x = e_1 \ \texttt{in} \ e_2 \ \texttt{end} : t_2, q_1} \quad (CT-LetPoly)$$

5 Semantics

5.1 State

A state $s \in \mathbf{State}$ is a triple (E, M, p) where $E \in \mathbf{Env}$ is the environment, $M \in \mathbf{Mem}$ is the memory, and $p \in \mathbb{N}$ is the memory pointer.

$$\begin{aligned}
\mathbf{State} &= \mathbf{Env} \times \mathbf{Mem} \times \mathbb{N} \\
\mathbf{Env} &= \mathbf{Var} \to \mathbf{Val} \cup \mathbf{Rec} \\
\mathbf{Mem} &= \mathbb{N} \to \mathbf{Val}
\end{aligned}$$

Both the environment and the memory are updateable mappings. Given an updateable mapping $f : X \to Y$, the updated mapping $f[x \mapsto y]$, where $x \in X, y \in Y$, is defined as

$$f[x \mapsto y](x') = \begin{cases} y & \text{if } x = x' \\ f(x) & \text{otherwise.} \end{cases}$$

Val consists of integers, booleans, lists, references, pairs, and functions. **Rec** is the set of recursions.

$$\begin{aligned}
\mathbf{Val} &= \bigcup_{t \in \mathbf{Typ}} \mathcal{V}_t \\
\mathcal{V}_{\texttt{unit}} &= \{\texttt{unit}\} \\
\mathcal{V}_{\texttt{int}} &= \mathbb{Z} = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\} \\
\mathcal{V}_{\texttt{bool}} &= \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\} \\
\mathcal{V}_{t\ \texttt{list}} &= \bigcup_{i=0}^{\infty} \mathcal{V}_{t\ \texttt{list}}^{(i)} \\
\mathcal{V}_{t\ \texttt{list}}^{(0)} &= \{\texttt{nil}\} \\
\mathcal{V}_{t\ \texttt{list}}^{(i+1)} &= \{\texttt{cons}\} \times \mathcal{V}_t \times \mathcal{V}_{t\ \texttt{list}}^{(i)} \\
\mathcal{V}_{t\ \texttt{ref}} &= \{\texttt{ref}\} \times \mathbb{N} \\
\mathcal{V}_{t_1 \times t_2} &= \{\texttt{pair}\} \times \mathcal{V}_{t_1} \times \mathcal{V}_{t_2} \\
\mathcal{V}_{t_1 \to t_2} &= {\mathcal{V}_{t_1}}^{\mathcal{V}_{t_2}} \subseteq \{\texttt{fun}\} \times \mathbf{Env} \times \mathbf{Var} \times \mathbf{Exp} \\
\mathbf{Rec} &= \{\texttt{rec}\} \times \mathbf{Env} \times \mathbf{Var} \times \mathbf{Exp}
\end{aligned}$$

5.2 Rules

Judgement form: $E, M, p; e \Downarrow M', p'; v$ where $E \in \mathbf{Env}$, $M, M' \in \mathbf{Mem}$, $p \in \mathbb{N}$, $e \in \mathbf{Exp}$, $v \in \mathbf{Val}$

$$\frac{\mathbf{n} = \mathcal{N}[\![n]\!]}{E, M, p; n \Downarrow M, p; \mathbf{n}} \quad \text{(E-Int)}$$

$$\frac{E(x) = (\texttt{rec}, E_1, x_1, e_1) \quad E_1, M, p; \texttt{rec}\ x_1\ \texttt{=>}\ e_1 \Downarrow M', p'; v}{E, M, p; x \Downarrow M', p'; v} \quad \text{(E-Name1)}$$

$$\frac{E(x) = v}{E, M, p; x \Downarrow M, p; v} \quad \text{(E-Name2)}$$

$$\frac{}{E, M, p; \texttt{true} \Downarrow M, p; \mathbf{tt}} \quad \text{(E-True)}$$

$$\frac{}{E, M, p; \texttt{false} \Downarrow M, p; \mathbf{ff}} \quad \text{(E-False)}$$

$$\frac{}{E, M, p; \texttt{nil} \Downarrow M, p; \texttt{nil}} \quad \text{(E-Nil)}$$

$$\frac{}{E, M, p; \texttt{()} \Downarrow M, p; \texttt{unit}} \quad \text{(E-Unit)}$$

$$\frac{E, M, p; e \Downarrow M, p; v}{E, M, p; \texttt{(}e\texttt{)} \Downarrow M, p; v} \quad \text{(E-Group)}$$

$$\frac{E, M, p+1; e \Downarrow M'', p'; v \quad M' = M''[p \mapsto v]}{E, M, p; \texttt{ref}\ e \Downarrow M', p'; (\texttt{ref}, p)} \quad \text{(E-Ref)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; (\texttt{ref}, p_1) \quad E, M', p'; e_2 \Downarrow M''', p''; v \quad M'' = M'''[p_1 \mapsto v]}{E, M, p; e_1 \texttt{ := } e_2 \Downarrow M'', p''; \texttt{unit}} \quad \text{(E-Assign)}$$

$$\frac{E, M, p; e \Downarrow M', p'; (\texttt{ref}, p_1) \quad v = M'(p_1)}{E, M, p; \texttt{!}e \Downarrow M', p'; v} \quad \text{(E-Deref)}$$

$$\frac{}{E, M, p; \texttt{fn}\ x\ \texttt{=>}\ e \Downarrow M, p; (\texttt{fun}, E, x, e)} \quad \text{(E-Fn)}$$

$$\frac{E[x \mapsto (\texttt{rec}, E, x, e)], M, p; e \Downarrow M', p'; v}{E, M, p; \texttt{rec}\ x\ \texttt{=>}\ e \Downarrow M', p'; v} \quad \text{(E-Rec)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; (\texttt{fun}, E_1, x, e) \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad E_1[x \mapsto v_2], M'', p''; e \Downarrow M''', p'''; v}{E, M, p; e_1\ e_2 \Downarrow M''', p''', v} \quad \text{(E-App)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2}{E, M, p; \texttt{(}e_1 \texttt{,}\ e_2\texttt{)} \Downarrow M'', p''; (\texttt{pair}, v_1, v_2)} \quad \text{(E-Pair)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2}{E, M, p; e_1 \texttt{ :: } e_2 \Downarrow M'', p''; (\texttt{cons}, v_1, v_2)} \quad \text{(E-Cons)}$$

$$\frac{E, M, p; e \Downarrow M', p'; v \quad v' = -v}{E, M, p; \texttt{~}e \Downarrow M', p'; v'} \quad \text{(E-NEG)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v = v_1 + v_2}{E, M, p; e_1 \texttt{ + } e_2 \Downarrow M'', p''; v} \quad \text{(E-ADD)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v = v_1 - v_2}{E, M, p; e_1 \texttt{ - } e_2 \Downarrow M'', p''; v} \quad \text{(E-SUB)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v = v_1 v_2}{E, M, p; e_1 \texttt{ * } e_2 \Downarrow M'', p''; v} \quad \text{(E-MUL)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_2 \neq 0 \quad v = v_1 \texttt{ div } v_2}{E, M, p; e_1 \texttt{ / } e_2 \Downarrow M'', p''; v} \quad \text{(E-DIV)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_2 \neq 0 \quad v = v_1 \texttt{ mod } v_2}{E, M, p; e_1 \texttt{ \% } e_2 \Downarrow M'', p''; v} \quad \text{(E-MOD)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 < v_2}{E, M, p; e_1 \texttt{ < } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-LESS1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \geq v_2}{E, M, p; e_1 \texttt{ < } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-LESS2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \leq v_2}{E, M, p; e_1 \texttt{ <= } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-LESSEQ1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 > v_2}{E, M, p; e_1 \texttt{ <= } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-LESSEQ2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 > v_2}{E, M, p; e_1 \texttt{ > } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-GREATER1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \leq v_2}{E, M, p; e_1 \texttt{ > } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-GREATER2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \geq v_2}{E, M, p; e_1 \texttt{ >= } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-GREATEREQ1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 < v_2}{E, M, p; e_1 \texttt{ >= } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-GREATEREQ2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 = v_2}{E, M, p; e_1 \texttt{ = } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-EQ1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \neq v_2}{E, M, p; e_1 \texttt{ = } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-EQ2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 \neq v_2}{E, M, p; e_1 \texttt{ <> } e_2 \Downarrow M'', p''; \mathbf{tt}} \quad \text{(E-NEQ1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2 \quad v_1 = v_2}{E, M, p; e_1 \texttt{ <> } e_2 \Downarrow M'', p''; \mathbf{ff}} \quad \text{(E-NEQ2)}$$

$$\frac{E, M, p; e \Downarrow M', p'; \mathbf{tt}}{E, M, p; \texttt{not}\ e \Downarrow M', p'; \mathbf{ff}} \quad \text{(E-NOT1)}$$

$$\frac{E, M, p; e \Downarrow M', p'; \mathbf{ff}}{E, M, p; \texttt{not}\ e \Downarrow M', p'; \mathbf{tt}} \quad \text{(E-NOT2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2 \Downarrow M'', p''; v}{E, M, p; e_1\ \texttt{andalso}\ e_2 \Downarrow M'', p''; v} \quad \text{(E-ANDALSO1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{ff}}{E, M, p; e_1\ \texttt{andalso}\ e_2 \Downarrow M', p'; \mathbf{ff}} \quad \text{(E-ANDALSO2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{tt}}{E, M, p; e_1\ \texttt{orelse}\ e_2 \Downarrow M', p'; \mathbf{tt}} \quad \text{(E-ORELSE1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{ff} \quad E, M', p'; e_2 \Downarrow M'', p''; v}{E, M, p; e_1\ \texttt{orelse}\ e_2 \Downarrow M'', p''; v} \quad \text{(E-ORELSE2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E, M', p'; e_2 \Downarrow M'', p''; v_2}{E, M, p; e_1\,;\,e_2 \Downarrow M'', p''; v_2} \quad \text{(E-SEQ)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; v_1 \quad E[x \mapsto v_1], M', p'; e_2 \Downarrow M'', p''; v_2}{E, M, p; \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow M'', p''; v_2} \quad \text{(E-LET)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2 \Downarrow M'', p''; v}{E, M, p; \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow M'', p''; v} \quad \text{(E-COND1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{ff} \quad E, M', p'; e_3 \Downarrow M'', p''; v}{E, M, p; \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow M'', p''; v} \quad \text{(E-COND2)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2\ ;\ \texttt{while}\ e_1\ \texttt{do}\ e_2 \Downarrow M'', p''; v}{E, M, p; \texttt{while}\ e_1\ \texttt{do}\ e_2 \Downarrow M'', p''; v} \quad \text{(E-LOOP1)}$$

$$\frac{E, M, p; e_1 \Downarrow M', p'; \mathbf{ff}}{E, M, p; \texttt{while}\ e_1\ \texttt{do}\ e_2 \Downarrow M', p'; \texttt{unit}} \quad \text{(E-LOOP2)}$$

5.3 Supplementary Details

- $\mathcal{N}[\![n]\!]$ is the value of n, e.g. $\mathcal{N}[\![123]\!] = 123$.
- If a and d are integers, with d non-zero, then a remainder a `mod` d is an integer r such that $a = qd + r$ for some integer q, and with $|r| < |d|$. a `div` $d = q$ is the quotient.
- Equality comparisons ($=$ and $\neq$) work for any equality type.
  - $\forall n \in \mathbb{Z}, n = n$
  - $\mathbf{tt} = \mathbf{tt}, \mathbf{ff} = \mathbf{ff}$
  - $\texttt{nil} = \texttt{nil}$
  - $(\texttt{cons}, h_1, t_1) = (\texttt{cons}, h_2, t_2) \iff h_1 = h_2 \wedge t_1 = t_2$
  - $(\texttt{ref}, p_1) = (\texttt{ref}, p_2) \iff p_1 = p_2$
  - $(\texttt{pair}, a_1, b_1) = (\texttt{pair}, a_2, b_2) \iff a_1 = a_2 \wedge b_1 = b_2$
  - $v_1 \neq v_2 \iff \neg(v_1 = v_2)$

6 Examples

```
let add = fn x => fn y => x + y
in  add 1 2
end
(* ==> 3 *)
```

examples/plus.spl

```
let fact = rec f => fn x => if x=1 then 1 else x * (f (x-1))
in  fact 4
end
(* ==> 24 *)
```

examples/factorial.spl

```
let gcd = rec g => fn a => fn b =>
            if b=0 then a else g b (a % b)
in  gcd 34986 3087
end
(* ==> 1029 *)
```

examples/gcd1.spl

```
let gcd = fn x => fn y =>
            let a = ref x in
              let b = ref y in
                let c = ref 0 in
                  (while !b <> 0 do c := !a; a := !b; b := !c % !b);
                  !a
                end
              end
            end
in  gcd 34986 3087
end
(* ==> 1029 *)
```

examples/gcd2.spl

```
let
  sum = rec sum =>
      fn a => if a=nil
                then 0
                else hd a + sum (tl a)
in
  sum (1::2::3::nil)
end
(* ==> 6 *)
```

examples/sum.spl

7 Implementation

7.1 Command-line Interface

You are required to implement the SimPL interpreter in Java, and submit a runnable JAR file, say `SimPL.jar`. Your interpreter should accept exactly one command-line argument, which is the path of the SimPL program, and then read the program file for execution. Your interpreter should output the result of the execution to the standard output (`System.out`).

- Your interpreter is started by using `java -jar SimPL.jar` *`program`*`.spl`.
- If there is a syntax error, output `syntax error`.
- If there is a type error, output `type error`.
- If there is a runtime error, output `runtime error`.
- If the result is an integer, output its value.
- If the result is **tt**, output `true`.
- If the result is **ff**, output `false`.
- If the result is `nil`, output `nil`.
- If the result is `unit`, output `unit`.
- If the result is a list, output `list@` followed by its length.
- If the result is a reference, output `ref@` followed by its content.
- If the result is a pair, output $\texttt{pair@}v_1\texttt{@}v_2$ where v_i is i-th element of the pair.
- If the result is a function, output `fun`.
- Spaces in the output are insignificant.
- For any test program, your interpreter has up to 5 seconds to execute it.
- Your interpreter is started in a sandbox environment and can only read the current test program.

7.2 Predefined Functions

$$
\begin{aligned}
fst &: t_1 \times t_2 \to t_1 \\
snd &: t_1 \times t_2 \to t_2 \\
hd &: t\ \texttt{list} \to t \\
tl &: t\ \texttt{list} \to t\ \texttt{list} \\
fst(\texttt{pair}, v_1, v_2) &= v_1 \\
snd(\texttt{pair}, v_1, v_2) &= v_2 \\
hd(\texttt{cons}, v_1, v_2) &= v_1 \\
tl(\texttt{cons}, v_1, v_2) &= v_2 \\
hd(\texttt{nil}) &= \texttt{error} \\
tl(\texttt{nil}) &= \texttt{error}
\end{aligned}
$$

`fst`, `snd`, `hd`, and `tl` are not keywords. They are predefined names in the topmost environment, work in the same way as user-defined functions, and can be bound to other values.

8 Bonus

- Mutually recursive combinator
- Infinite streams
- Garbage collection (of ref cells)
- Tail recursion
- Lazy evaluation
- Other features or optimizations