

# CSE3302-004/CSE5307-001 Course Project Specification

Version 6.0

Last Modified on October 9, 2024

Course Instructor:      Kenny Zhu      kenny.zhu@uta.edu  
Teaching Assistant:    Essam Abdelghany    abdelghanye@uta.edu

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Overview</b>	<b>2</b>
<b>3</b>	<b>Lexical Definition</b>	<b>3</b>
3.1	Comments	3
3.2	Atoms	3
3.3	Keywords	3
3.4	Operators	3
<b>4</b>	<b>Syntax</b>	<b>4</b>
4.1	Operator Precedence	4
<b>5</b>	<b>Typing</b>	<b>5</b>
5.1	Definition	5
5.2	Polymorphism	6
<b>6</b>	<b>Semantics</b>	<b>8</b>
6.1	State	8
6.2	Rules	8
6.3	Supplementary Details	11
<b>7</b>	<b>Implementation</b>	<b>12</b>
7.1	Command-line Interface	12
7.2	Predefined Functions	12
<b>8</b>	<b>Bonus</b>	<b>13</b>

# 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 Overview

SimPL has support for control structures (branching, looping, recursion), variable definition, and type inference, among other features.

```
1 let add = fn x => fn y => x + y
2 in   add 1 2
3 end
4 (* ==> 3 *)
```

```
1 (fn p =>
2   if (fst p) > (snd p)
3   then fst p
4   else snd p
5 ) (1, 2)
6 (* ==> 2 *)
```

```
1 let fact = rec f =>
2   fn x =>
3     if x = 1 then 1
4     else x * (f (x - 1))
5 in
6   fact 4
7 end
8 (* ==> 24 *)
```

```
1 let gcd = fn x => fn y =>
2   let a = ref x in
3   let b = ref y in
4   let c = ref 0 in
5     while !b <> 0 do (c := !a; a := !b; b := !c % !b);
6     !a
7   end
8   end
9   end
10 in   gcd 68 51
11 end
12 (* ==> 17 *)
```

## 3 Lexical Definition

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

### 3.1 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.

### 3.2 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']*`.

### 3.3 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`

### 3.4 Operators

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

## 4 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  $uop \in \{\sim, \text{not}, !\}$
- Binary operator  $bop \in \{+, -, *, /, \%, =, <>, <, <=, >, >=, \text{andalso}, \text{orelse}, ::, :=, ;\}$

Expression $e$	::=	$n$	integer literal
		$x$	name
		true	true value
		false	false value
		nil	empty list
		ref $e$	reference creation
		fn $x \Rightarrow e$	function
		rec $x \Rightarrow e$	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

### 4.1 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

## 5 Typing

### 5.1 Definition

Arbitrary type $t$	::=	int	Equality type $\alpha$	::=	int
		bool			bool
		unit			$\alpha$ list
		$t$ list			$t$ ref
		$t$ ref			$\alpha \times \alpha$
		$t \times t$			
		$t \rightarrow t$			

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

- $\text{int} \in \mathbf{Typ}, \text{bool} \in \mathbf{Typ}, \text{unit} \in \mathbf{Typ}$
- $\forall t \in \mathbf{Typ}, t \text{ list} \in \mathbf{Typ}$
- $\forall t \in \mathbf{Typ}, t \text{ ref} \in \mathbf{Typ}$
- $\forall t_1, t_2 \in \mathbf{Typ}, t_1 \times t_2 \in \mathbf{Typ}$
- $\forall t_1, t_2 \in \mathbf{Typ}, t_1 \rightarrow t_2 \in \mathbf{Typ}$

Type environment  $\Gamma : \mathbf{Var} \rightarrow \mathbf{Typ}$  is a mapping from names to arbitrary types. The replacement of  $x$  with  $t$  in  $\Gamma$ , 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:

$\overline{\Gamma \vdash n : \text{int}}$	(T-INT)	$\overline{\Gamma \vdash \text{true} : \text{bool}}$	(T-TRUE)	$\overline{\Gamma \vdash \text{nil} : \alpha \text{ list}}$	(T-NIL)
$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$	(T-NAME)	$\overline{\Gamma \vdash \text{false} : \text{bool}}$	(T-FALSE)	$\overline{\Gamma \vdash () : \text{unit}}$	(T-UNIT)
$\frac{\Gamma[x : t_1] \vdash e : t_2}{\Gamma \vdash \text{fn } x \Rightarrow e : t_1 \rightarrow t_2}$	(T-FN)	$\frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \sim e : \text{int}}$	(T-NEG)	$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e) : t}$	(T-GROUP)
$\frac{\Gamma[x : t] \vdash e : t}{\Gamma \vdash \text{rec } x \Rightarrow e : t}$	(T-REC)	$\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{not } e : \text{bool}}$	(T-NOT)	$\frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 ; e_2 : t_2}$	(T-SEQ)

$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{ref } e : t \text{ ref}}$	(T-REF)	$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad bop \in \{+, -, *, /, \%\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \text{int}}$	(T-ARITH)
$\frac{\Gamma \vdash e_1 : t \text{ ref} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : \text{unit}}$	(T-ASSIGN)	$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \text{bool}}$	(T-REL)
$\frac{\Gamma \vdash e : t \text{ ref}}{\Gamma \vdash !e : t}$	(T-DEREF)	$\frac{\Gamma \vdash e_1 : \alpha \quad \Gamma \vdash e_2 : \alpha \quad bop \in \{=, <\>\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \text{bool}}$	(T-EQ)
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$	(T-PAIR)	$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ andalso } e_2 : \text{bool}}$	(T-ANDALSO)
$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \text{ list}}{\Gamma \vdash e_1 :: e_2 : t \text{ list}}$	(T-CONS)	$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ orelse } e_2 : \text{bool}}$	(T-ORELSE)
$\frac{\Gamma \vdash e_1 : t_2 \rightarrow t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1}$	(T-APP)	$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$	(T-COND)
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : t_2}$	(T-LET)	$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}}$	(T-LOOP)

## 5.2 Polymorphism

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

1. Some constraint typing rules:

$\overline{\Gamma \vdash n : \text{int}, \{\}}$	(CT-INT)	$\frac{\Gamma \vdash e_1 : t_1, q_1 \quad \Gamma \vdash e_2 : t_2, q_2 \quad bop \in \{+, -, *, /, \%\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \text{int}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}}$	(CT-ARITH)
$\frac{\Gamma(x) = t}{\Gamma \vdash x : t, \{\}}$	(CT-NAME)	$\frac{\Gamma \vdash e_1 : t_1, q_1 \quad \Gamma \vdash e_2 : t_2, q_2 \quad bop \in \{<, <=, >, >=\}}{\Gamma \vdash e_1 \text{ bop } e_2 : \text{bool}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}}$	(CT-REL)
$\overline{\Gamma \vdash \text{true} : \text{bool}, \{\}}$	(CT-TRUE)		
$\overline{\Gamma \vdash \text{false} : \text{bool}, \{\}}$	(CT-FALSE)	$\frac{\Gamma, x : a \vdash e : t, q}{\Gamma \vdash \text{fn } x : a \Rightarrow e : b : a \rightarrow b, q \cup \{t = b\}}$	(CT-FN)

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

2. Unification algorithm:

$$\begin{array}{c}
\overline{(S, \{int = int\} \cup q) \rightarrow (S, q)} \\
\overline{(S, \{bool = bool\} \cup q) \rightarrow (S, q)} \\
\overline{(S, \{a = a\} \cup q) \rightarrow (S, q)} \\
\overline{(S, \{s_{11} \rightarrow s_{12} = s_{21} \rightarrow s_{22}\} \cup q) \rightarrow (S, \{s_{11} = s_{21}, s_{12} = s_{22}\} \cup q)} \\
\overline{(S, \{a = s\} \cup q) \rightarrow ([a = s] \circ S, q[s/a])} \text{ (} a \text{ not in } FV(s) \text{)} \\
\overline{(S, \{s = a\} \cup q) \rightarrow ([a = s] \circ S, q[s/a])} \text{ (} a \text{ not in } FV(s) \text{)}
\end{array}$$

3. Let-Polymorphism is also needed.

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

## 6 Semantics

### 6.1 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.

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

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

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

$\mathbf{Val}$  consists of integers, booleans, lists, references, pairs, and functions.  $\mathbf{Rec}$  is the set of recursions.

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

### 6.2 Rules

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



$\frac{\mathbf{n} = \mathcal{N}[\llbracket n \rrbracket]}{E, M, p; n \rightarrow M, p; \mathbf{n}}$	(E-INT)
$\frac{E(x) = (\text{rec}, E_1, x_1, e_1) \quad E_1, M, p; \text{rec } x_1 \Rightarrow e_1 \rightarrow M', p'; v}{E, M, p; x \rightarrow M', p'; v}$	(E-NAME1)
$\frac{E(x) = v}{E, M, p; x \rightarrow M, p; v}$	(E-NAME2)
$\frac{}{E, M, p; \text{true} \rightarrow M, p; \mathbf{tt}}$	(E-TRUE)
$\frac{}{E, M, p; \text{false} \rightarrow M, p; \mathbf{ff}}$	(E-FALSE)
$\frac{}{E, M, p; \text{nil} \rightarrow M, p; \mathbf{nil}}$	(E-NIL)
$\frac{}{E, M, p; () \rightarrow M, p; \mathbf{unit}}$	(E-UNIT)
$\frac{E, M, p; e \rightarrow M, p; v}{E, M, p; (e) \rightarrow M, p; v}$	(E-GROUP)
$\frac{E, M, p+1; e \rightarrow M'', p'; v \quad M' = M''[p \mapsto v]}{E, M, p; \text{ref } e \rightarrow M', p'; (\text{ref}, p)}$	(E-REF)
$\frac{E, M, p; e_1 \rightarrow M', p'; (\text{ref}, p_1) \quad E, M', p'; e_2 \rightarrow M''', p''; v \quad M'' = M'''[p_1 \mapsto v]}{E, M, p; e_1 := e_2 \rightarrow M'', p''; \mathbf{unit}}$	(E-ASSIGN)
$\frac{E, M, p; e \rightarrow M', p'; (\text{ref}, p_1) \quad v = M'(p_1)}{E, M, p; !e \rightarrow M', p'; v}$	(E-DEREF)
$\frac{}{E, M, p; \text{fn } x \Rightarrow e \rightarrow M, p; (\text{fun}, E, x, e)}$	(E-FN)
$\frac{E[x \mapsto (\text{rec}, E, x, e)], M, p; e \rightarrow M', p'; v}{E, M, p; \text{rec } x \Rightarrow e \rightarrow M', p'; v}$	(E-REC)
$\frac{E, M, p; e_1 \rightarrow M', p'; (\text{fun}, E_1, x, e) \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad E_1[x \mapsto v_2], M'', p''; e \rightarrow M''', p'''; v}{E, M, p; e_1 \ e_2 \rightarrow M''', p''', v}$	(E-APP)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2}{E, M, p; (e_1, e_2) \rightarrow M'', p''; (\text{pair}, v_1, v_2)}$	(E-PAIR)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2}{E, M, p; e_1 :: e_2 \rightarrow M'', p''; (\text{cons}, v_1, v_2)}$	(E-CONS)

$\frac{E, M, p; e \rightarrow M', p'; v \quad v' = -v}{E, M, p; \sim e \rightarrow M', p'; v'}$	(E-NEG)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v = v_1 + v_2}{E, M, p; e_1 + e_2 \rightarrow M'', p''; v}$	(E-ADD)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v = v_1 - v_2}{E, M, p; e_1 - e_2 \rightarrow M'', p''; v}$	(E-SUB)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v = v_1 v_2}{E, M, p; e_1 * e_2 \rightarrow M'', p''; v}$	(E-MUL)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_2 \neq 0 \quad v = v_1 \text{ div } v_2}{E, M, p; e_1 / e_2 \rightarrow M'', p''; v}$	(E-DIV)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_2 \neq 0 \quad v = v_1 \text{ mod } v_2}{E, M, p; e_1 \% e_2 \rightarrow M'', p''; v}$	(E-MOD)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 < v_2}{E, M, p; e_1 < e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-LESS1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \geq v_2}{E, M, p; e_1 < e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-LESS2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \leq v_2}{E, M, p; e_1 \leq e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-LESSEQ1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 > v_2}{E, M, p; e_1 \leq e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-LESSEQ2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 > v_2}{E, M, p; e_1 > e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-GREATER1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \leq v_2}{E, M, p; e_1 > e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-GREATER2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \geq v_2}{E, M, p; e_1 \geq e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-GREATEREQ1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 < v_2}{E, M, p; e_1 \geq e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-GREATEREQ2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 = v_2}{E, M, p; e_1 = e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-EQ1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \neq v_2}{E, M, p; e_1 = e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-EQ2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 \neq v_2}{E, M, p; e_1 <> e_2 \rightarrow M'', p''; \mathbf{tt}}$	(E-NEQ1)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2 \quad v_1 = v_2}{E, M, p; e_1 <> e_2 \rightarrow M'', p''; \mathbf{ff}}$	(E-NEQ2)

$\frac{E, M, p; e \rightarrow M', p'; \mathbf{tt}}{E, M, p; \text{not } e \rightarrow M', p'; \mathbf{ff}}$	(E-NOT1)
$\frac{E, M, p; e \rightarrow M', p'; \mathbf{ff}}{E, M, p; \text{not } e \rightarrow M', p'; \mathbf{tt}}$	(E-NOT2)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2 \rightarrow M'', p''; v}{E, M, p; e_1 \text{ andalso } e_2 \rightarrow M'', p''; v}$	(E-ANDALSO1)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{ff}}{E, M, p; e_1 \text{ andalso } e_2 \rightarrow M', p'; \mathbf{ff}}$	(E-ANDALSO2)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{tt}}{E, M, p; e_1 \text{ orelse } e_2 \rightarrow M', p'; \mathbf{tt}}$	(E-ORELSE1)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{ff} \quad E, M', p'; e_2 \rightarrow M'', p''; v}{E, M, p; e_1 \text{ orelse } e_2 \rightarrow M'', p''; v}$	(E-ORELSE2)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E, M', p'; e_2 \rightarrow M'', p''; v_2}{E, M, p; e_1; e_2 \rightarrow M'', p''; v_2}$	(E-SEQ)
$\frac{E, M, p; e_1 \rightarrow M', p'; v_1 \quad E[x \mapsto v_1], M', p'; e_2 \rightarrow M'', p''; v_2}{E, M, p; \text{let } x = e_1 \text{ in } e_2 \text{ end} \rightarrow M'', p''; v_2}$	(E-LET)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2 \rightarrow M'', p''; v}{E, M, p; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow M'', p''; v}$	(E-COND1)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{ff} \quad E, M', p'; e_3 \rightarrow M'', p''; v}{E, M, p; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow M'', p''; v}$	(E-COND2)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{tt} \quad E, M', p'; e_2; \text{while } e_1 \text{ do } e_2 \rightarrow M'', p''; v}{E, M, p; \text{while } e_1 \text{ do } e_2 \rightarrow M'', p''; v}$	(E-LOOP1)
$\frac{E, M, p; e_1 \rightarrow M', p'; \mathbf{ff}}{E, M, p; \text{while } e_1 \text{ do } e_2 \rightarrow M', p'; \text{unit}}$	(E-LOOP2)

### 6.3 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  $a \bmod d$  is an integer  $r$  such that  $a = qd + r$  for some integer  $q$ , and with  $|r| < |d|$ .  $a \text{ div } d = q$  is the quotient.
- Equality comparisons ( $=$  and  $\neq$ ) work for any equality type.

- $\forall n \in \mathbb{Z}, n = n$
- $\mathbf{tt} = \mathbf{tt}, \mathbf{ff} = \mathbf{ff}$
- $\mathbf{nil} = \mathbf{nil}$
- $(\text{cons}, h_1, t_1) = (\text{cons}, h_2, t_2) \iff h_1 = h_2 \wedge t_1 = t_2$
- $(\text{ref}, p_1) = (\text{ref}, p_2) \iff p_1 = p_2$
- $(\text{pair}, a_1, b_1) = (\text{pair}, a_2, b_2) \iff a_1 = a_2 \wedge b_1 = b_2$
- $v_1 \neq v_2 \iff \neg(v_1 = v_2)$

## 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 `pair@ $v_1$ @ $v_2$`  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

Function	Type/Result
<code>fst</code>	$t_1 \times t_2 \rightarrow t_1$
<code>snd</code>	$t_1 \times t_2 \rightarrow t_2$
<code>hd</code>	$t \text{ list} \rightarrow t$
<code>tl</code>	$t \text{ list} \rightarrow t \text{ list}$
<code>fst(pair(<math>v_1, v_2</math>))</code>	$v_1$
<code>snd(pair(<math>v_1, v_2</math>))</code>	$v_2$
<code>hd(cons(<math>v_1, v_2</math>))</code>	$v_1$
<code>tl(cons(<math>v_1, v_2</math>))</code>	$v_2$
<code>hd(nil)</code>	<b>error</b>
<code>tl(nil)</code>	<b>error</b>

Table 1: Function types and results

`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