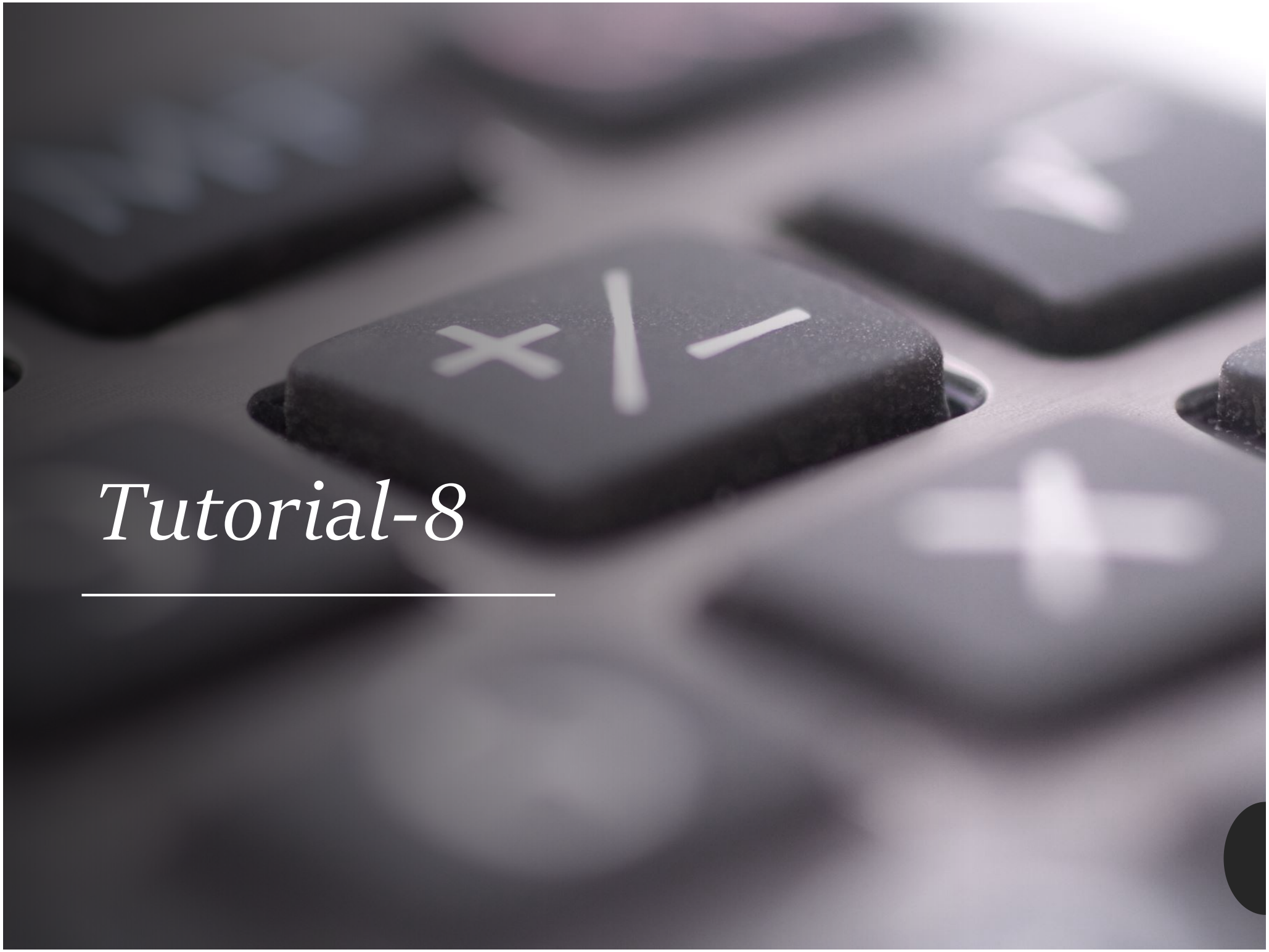


Tutorial-8


Quiz-7

7. Why use substitution to implement function application is not efficient?

7. Why use substitution to implement function application is not efficient?

Search through e1 for free occurrences of x during substitution

Go though e1 again to evaluate it: e1 → v1*

That's double the work!

8. In environment model, we bind __ to __?

a. Variable; type
b. Variable; value
c. Expression; type
d. Expression; value

8. In environment model, we bind __ to __?

a. Variable; type

b. Variable; value

c. Expression; type

d. Expression; value

9. What is closures in environment model?

- a pair of a function and its environment

10. Which rule is incorrect for the evaluation under the environment model?

Environment Model (evaluation)

$$\frac{E(x) = v}{(E, x) \rightarrow^* v} \text{ (E - var)} \qquad \frac{}{(E, \lambda x.e) \rightarrow^* \{\lambda x.e, \mathrm{E}\}} \text{ (E - fun)}$$

$$\frac{(E, \mathrm{e}_1) \rightarrow^* \{\lambda x.e, E_1\} \quad (E, \mathrm{e}_2) \rightarrow^* v_2 \quad (E_1[x \mapsto v_2], \mathrm{e}) \rightarrow^* v,}{(E, (\mathrm{e}_1 \ \ \mathrm{e}_2)) \rightarrow^* v} \text{ (E - app)}$$

$$\frac{(E, \mathrm{e}_1) \rightarrow^* v_1 \quad (E[x \mapsto v_1], \mathrm{e}_2) \rightarrow^* v_2}{(E, \text{let x} = \mathrm{e}_1 \text{ in } \mathrm{e}_2) \rightarrow^* v_2} \text{ (E - let)}$$

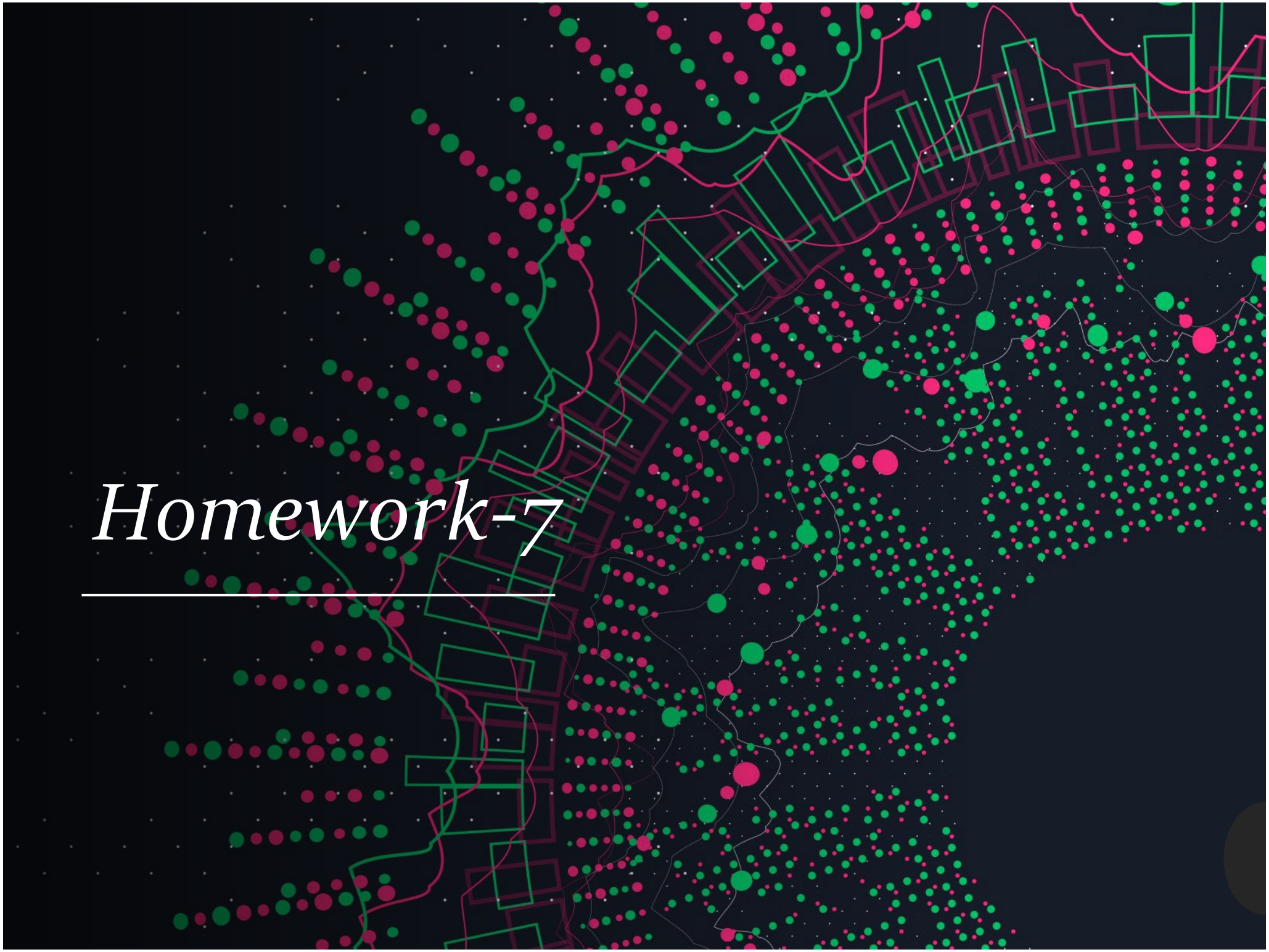
10. Which rule is incorrect for the evaluation under the environment model?

a. $$\frac{E(x) \square v}{(E, x) \rightarrow^{*} v} \quad \text{(E - var)}$$

b. $$\frac{}{(E, \lambda x.e) \rightarrow^{*} \{\lambda x.e, \mathrm{E}\}} \quad \text{(E - fun)}$$

c. $$\frac{(E, \mathrm{e1}) \rightarrow^{*} \{\lambda x.e, E_1\} \quad (E, \mathrm{e}_2) \rightarrow^{*} v_2 \quad (E_1[x \mapsto v_2], \mathrm{e}) \rightarrow^{*} v,}{(E, (\mathrm{e1}\ \ \mathrm{e2})) \rightarrow^{*} v} \quad \text{(E - app)}$$

d. $$\frac{(E, \mathrm{e}_1) \rightarrow v_1 \quad (E[x \mapsto v_1],\ \mathrm{e}_2) \rightarrow^{*} v_2}{(E,\ \text{let x} = \mathrm{e}_1\ \text{in}\ \mathrm{e}_2) \rightarrow^{*} v_2} \quad \text{(E-let)}$$


Homework-7

Problem-1

Problem 1. (30 points) Wouldn't it be simpler just to require the programmer to annotate error with its intended type in each context where it is used ? Why ?

Solution. Annotating error with its intended type would break the type preservation property. For example, the well-typed term

$$(\lambda x : Nat.x)\ ((\lambda y : Bool.5)\ (error\ as\ Bool));$$

(where error as T is the type-annotated syntax for exceptions) would evaluate in one step to an ill-typed term:

$$(\lambda x : Nat.x)\ \ (error\ as\ Bool);$$

As the evaluation rules propagate an error from the point where it occurs up to the top-level of a program, we may view it as having different types. The flexibility in the T-ERROR rule permits us to do this. □

Problem-2

Problem 2. (35 points) In lecture *Going Imperative*, the language is extended with while loop. In this problem, you are required to define the syntax and the semantics (including evaluation rules and typing rules) of while loop with `break` and `continue`

(a) Syntax: (x and x_i are names)

$$e ::= \ ... \mid \mathit{while}\ e_1\ \mathit{do}\ e_2 \mid \mathit{break} \mid \mathit{continue}$$

(Because we define the type of break and continue as unit, here we don't need to extend values and types)

While Loop

- Loops are essential in imperative programs:

```
while (!n>1) do
        x := !x * !n;
        n := !n -1
```

- Syntax:

```
e::= … | while e1 do e2
```

- Evaluation:

$$\frac{}{(M, \text{while } e_1 \text{ do } e_2) \rightarrow (M, \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else } ())} \text{ (E - While)}$$

- Typing:

$$\frac{\Sigma;\Gamma \mid\!- e_1 : \text{bool} \quad \Sigma;\Gamma \mid\!- e_2 : \text{unit}}{\Sigma;\Gamma \mid\!- \text{while } e_1 \text{ do } e_2 : \text{unit}} \text{ (T - While)}$$

Problem-2

Problem 2. (35 points) In lecture *Going Imperative*, the language is extended with while loop. In this problem, you are required to define the syntax and the semantics (including evaluation rules and typing rules) of while loop with `break` and `continue`

(b) Semantics:

- Evaluation Rules (We introduce $< e_1, e_2 >$)

$$\frac{}{(M, \mathit{while}\ e_1\ \mathit{do}\ e_2) \to (M, \mathit{if}\ e_1\ \mathit{then}\ < e2, \mathit{while}\ e_1\ \mathit{do}\ e_2 >\ \ \mathit{else}\ ())} \quad \text{(E-while)}$$

$$\frac{(M, e_1) \to (M', e_1')}{(M, < e_1, e_2 >) \to (M', < e_1', e_2 >)} \quad \text{(E-whilePair)}$$

$$\frac{}{(M, < \mathit{break}, e >) \to (M, ())} \quad \text{(E-whilePairBreak)}$$

$$\frac{}{(M, < \mathit{continue}, e >) \to (M, e)} \quad \text{(E-whilePairContinue)}$$

$$\frac{}{(M, < \mathit{break}; e_1, e_2 >) \to (M, ())} \quad \text{(E-breakSeq)}$$

$$\frac{}{(M, < \mathit{continue}; e_1, e_2 >) \to (M, e_2)} \quad \text{(E-continueSeq)}$$

$$\frac{}{(M, < (), e_2 >) \to (M, e_2)} \quad \text{(E-whilePairUnit)}$$

Problem-2

Problem 2. (35 points) In lecture *Going Imperative*, the language is extended with while loop. In this problem, you are required to define the syntax and the semantics (including evaluation rules and typing rules) of while loop with `break` and `continue`

- Typing Rules (We don't need to define typing rules for $< e_1, e_2 >$)

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

$$\frac{}{\Sigma; \Gamma \vdash break : unit} \quad \text{(T-break)}$$

$$\frac{}{\Sigma; \Gamma \vdash continue : unit} \quad \text{(T-continue)}$$

Problem-3

Proof **Preservation Theorem**: If $\Sigma; \Gamma \vdash e : t, \Sigma; \Gamma \vdash M$, and $(M, e) \to (M', e')$, then for some $\Sigma' \supseteq \Sigma, \Sigma'; \Gamma \vdash e' : t, \Sigma'; \Gamma \vdash M'$. ($\Sigma' \supseteq \Sigma$ means Σ' agrees with Σ on all the old locations.)

Hint: You don't need to write "need to prove..." in this problem since in all cases it's quite similar. Also, you can use directly the following two lemma whose proofs are quite easy:

Lemma 1. ***Substitution:*** *If* $\Sigma; \Gamma, x : t_1 \vdash e : t_2$ *and* $\Sigma; \Gamma \vdash v : t_1$, *then* $\Sigma; \Gamma \vdash e[v/x] : t_2$ *(similar to the proof of previous substitution lemma)*

Lemma 2. *If* $\Sigma; \Gamma \vdash e : t$ *and* $\Sigma' \supseteq \Sigma$, *then* $\Sigma'; \Gamma \vdash e : t$ *(easy induction)*

Problem-3

Proof.

$$\frac{}{\Sigma;\Gamma \vdash x : \Gamma(x)} \quad (T-Var)$$

$$\frac{\Sigma;\Gamma x : t_1 \vdash e : t_2}{\Sigma;\Gamma \vdash \lambda x : t_1.e : t_1 \to t_2} \quad (T-Abs)$$

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

$$\frac{}{\Sigma;\Gamma \vdash () : unit} \quad (T-Unit)$$

$$\frac{\Sigma(l) = t}{\Sigma;\Gamma \vdash l : t\ ref} \quad (T-Loc)$$

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

$$\frac{\Sigma;\Gamma \vdash e : t\ ref}{\Sigma;\Gamma \vdash !e : t} \quad (T-DeRef)$$

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

(Here I don't list Boolean an Condition rules since they are not in the slides. Actually we should also proof these rules and the proof of these rules are similar.)

Problem-3

By induction on the derivation of $\Sigma; \Gamma \vdash e : t$

1. case $\dfrac{}{\Sigma; \Gamma \vdash x : \Gamma(x)}$

   Can't happen (There are no evaluations rules).

2. case $\dfrac{\Sigma; \Gamma x : t_1 \vdash e : t_2}{\Sigma; \Gamma \vdash \lambda x : t_1 . e : t_1 \to t_2}$

   Can't happen.

3. case $\dfrac{\Sigma; \Gamma \vdash e_1 : t_1 \to t_2 \quad \Sigma; \Gamma \vdash e_2 : t_1}{\Sigma; \Gamma \vdash e_1\ e_2 : t_2}$

   - Subcase E-App1: $\dfrac{(M, e_1) \to (M', e_1')}{(M, (e_1\ e_2)) \to (M', e_1'\ e_2)}$

     (1) $\Sigma; \Gamma \vdash e_1 : t_1 \to t_2, (M, e_1) \to (M', e_1')$ (by assumption)

     (2) $\exists \Sigma' \supseteq \Sigma, \Sigma'; \Gamma \vdash e_1' : t_1 \to t_2, \Sigma'; \Gamma \vdash M'$ (by I.H.)

     (3) $\Sigma; \Gamma \vdash e_2 : t_1$ (by assumption)

     (4) $\Sigma'; \Gamma \vdash e_2 : t_1$ (by (2), (3) and **Lemma 2**)

     (5) $\Sigma'; \Gamma \vdash e_1'\ e_2 : t_2$ (by (2), (4) and T-App)

REFERENCES (OPERATIONAL SEMANTICS)

$$\frac{(M, e_1) \rightarrow (M', e_1')}{(M, (e_1\ \ e_2)) \rightarrow (M', (e_1'\ e_2))}\ (E\text{-}App1) \qquad \frac{(M, e_2) \rightarrow (M', e_2')}{(M, (v_1\ \ e_2)) \rightarrow (M', (v_1\ e_2'))}\ (E\text{-}App2)$$

$$\frac{}{(M, (\backslash x : t.\, e_1)\ \ v_2) \rightarrow (M, e_1[v_2/x])}\ (E\text{-}AppAbs)$$

$$\frac{(M, e_1) \rightarrow (M', e_1')}{(M, \text{let } x = e_1 \text{ in } e_2)) \rightarrow (M', \text{let } x = e_1' \text{ in } e_2)}\ (E\text{-}Let1)$$

$$\frac{}{(M,\ \text{let } x = v_1 \text{ in } e_2)) \rightarrow (M, e_2[v_1/x])}\ (E\text{-}Let2)$$

REFERENCES (OPERATIONAL SEMANTICS, CONT'D)

$$\frac{(M, e) \to (M', e')}{(M, \text{ref } e) \to (M', \text{ref } e')} \text{ (E - Ref)} \qquad \frac{l \notin \text{dom}(M)}{(M, \text{ref } v) \to ((M, l \mapsto v), l)} \text{ (E - RefV)}$$

$$\frac{(M, e) \to (M', e')}{(M, !e) \to (M', !e')} \text{ (E - DeRef)} \qquad \frac{}{(M, !l) \to (M, M(l))} \text{ (E - DeRefLoc)}$$

$$\frac{(M, e_1) \to (M', e_1')}{(M, e_1 := e_2) \to (M', e_1' := e_2)} \text{ (E - Assign1)} \qquad \frac{(M, e_2) \to (M', e_2')}{(M, v_1 := e_2) \to (M', v_1 := e_2')} \text{ (E - Assign2)}$$

$$\frac{}{(M, l := v) \to (M[l \mapsto v], ())} \text{ (E - Assign)}$$

- Subcase E-App2: $\dfrac{(M, e_2) \to (M', e_2')}{(M, (v_1\ e_2)) \to (M', v_1'\ e_2')}$

  (1) $\Sigma; \Gamma \vdash e_2 : t_2, (M, e_2) \to (M', e_2')$ (by assumption)

  (2) $\exists \Sigma' \supseteq \Sigma, \Sigma'; \Gamma \vdash e_2' : t_2, \Sigma'; \Gamma \vdash M'$ (by I.H.)

  (3) $\Sigma; \Gamma \vdash v_1 : t_1 \to t_2$ (by assumption)

  (4) $\Sigma'; \Gamma \vdash v_1 : t_1 \to t_2$ (by (2), (3) and **Lemma 2**)

  (5) $\Sigma'; \Gamma \vdash v_1'\ e_2 : t_2$ (by (2), (4) and T-App)

- Subcase E-AppAbs: $\dfrac{}{(M, \lambda x : t_1.e_3\ v_2) \to (M, e_3[v_2/x])}$

  (1) $\Sigma; \Gamma \vdash \lambda x : t_1.e_3 : t_1 \to t_2$ (by assumption)

  (2) $\Sigma; \Gamma.x : t_1 \vdash e_3 : t_2$ (by inversion of T-Abs)

  (3) $\Sigma; \Gamma \vdash v_2 : t_1$ (by assumption)

  (4) $\Sigma; \Gamma \vdash e_3[v_2/x] : t_2$ (by (2), (3) and **Lemma 1**)

  (5) $\Sigma' = \Sigma$ satisfies.

4. case $\dfrac{}{\Sigma; \Gamma \vdash () : unit}$

   Can't happen.

5. case $\dfrac{\Sigma(l) = t}{\Sigma; \Gamma \vdash l : t\ ref}$

   Can't happen.

6. case $\dfrac{\Sigma; \Gamma \vdash e : t}{\Sigma; \Gamma \vdash ref\ e : t\ ref}$

- Subcase E-RefV: $\dfrac{l \notin dom(M)}{(M, ref\ v) \to ((M, l \mapsto v), l)}$

  (1) Let $\Sigma' = \Sigma, l : t$

  (2) $\Sigma'(l) = t$ (by (1))

  (3) $\Sigma'; \Gamma \vdash l : t\ ref$ (by (2))

  (4) $\Sigma; \Gamma \vdash M$ (by I.H.)

  (5) $M'(l) = v$

  (6) $\Sigma; \Gamma \vdash v : t$ (by assumption and $\Sigma' \supseteq \Sigma$)

  (7) $\Sigma'; \Gamma \vdash M'$ (by (2), (4), (5), (6) and definition of $\Sigma; \Gamma \vdash M$)

- Subcase E-Ref: $\dfrac{(M, e) \to (M', e')}{(M, ref\ e) \to (M', ref\ e')}$

  (1) $\Sigma; \Gamma \vdash e : t, (M, e) \to (M', e')$ (by assumption)

  (2) $\exists \Sigma', \Sigma'; \Gamma \vdash e' : t, \Sigma'; \Gamma \vdash M'$ (by (1) and I.H.)

  (3) $\Sigma'; \Gamma \vdash ref\ e' : t\ ref$ (by (2) and T-Ref)

7. case $\dfrac{\Sigma;\Gamma \vdash e : t\ ref}{\Sigma;\Gamma \vdash !e : t}$

- Subcase E-DeRefLoc: $\dfrac{}{(M, !l) \to (M, M(l))}$

  (1) Let $M(l) = v$ and $\Sigma' = \Sigma$

  Now we only need to prove $\Sigma;\Gamma \vdash v : t$

  (2) $\Sigma;\Gamma \vdash M$ (by I.H.)

  (3) $\Sigma;\Gamma \vdash !l : t$ (by assumption)

  (4) $\Sigma(l) = t$ (by (3))

  (5) $\Sigma;\Gamma \vdash v : t$ (by (1), (2), (4) and the definition of $\Sigma;\Gamma \vdash M$)

- Subcase E-DeRef: $\dfrac{(M, e) \to (M', e')}{(M, !e) \to (M', !e')}$

  (1) $\Sigma;\Gamma \vdash e : t\ ref$ and $(M, e) \to (M', e')$ (by assumption)

  (2) $\exists \Sigma' \supseteq \Sigma, \Sigma';\Gamma \vdash e' : t\ ref$ and $\Sigma';\Gamma \vdash M'$ (by (1) and I.H.)

  (3) $\Sigma';\Gamma \vdash !e' : t$ (by (2) and T-DeRef)

8. case $\dfrac{\Sigma;\Gamma \vdash e_1 : t\ ref \quad \Sigma;\Gamma \vdash e_2 : t}{\Sigma;\Gamma \vdash e_1 := e_2 : unit}$

- Subcase E-Assign1: $\dfrac{(M, e_1) \to (M', e_1')}{(M, e_1 := e_2) \to (M', e_1' := e_2)}$

  (1) $\Sigma;\Gamma \vdash e_1 : t\ ref$, $\Sigma;\Gamma \vdash e_2 : t$ and $(M, e_1) \to (M', e_1')$ (by assumption)

  (2) $\exists \Sigma' \supseteq \Sigma, \Sigma';\Gamma \vdash e_1 : t\ ref$ and $\Sigma';\Gamma \vdash M'$ (by (1) and I.H.)

  (3) $\Sigma';\Gamma \vdash e_1' := e_2 : unit$ (by (1), (2) and T-Assign)

- Subcase E-Assign2: $\dfrac{(M, e_2) \to (M', e_2')}{(M, v_1 := e_2) \to (M', v_1 := e_2')}$

  (1) $\Sigma;\Gamma \vdash v_1 : t\ ref$, $\Sigma;\Gamma \vdash e_2 : t$ and $(M, e_2) \to (M', e_2')$ (by assumption)

  (2) $\exists \Sigma' \supseteq \Sigma, \Sigma';\Gamma \vdash e_2 : t$ and $\Sigma';\Gamma \vdash M'$ (by (1) and I.H.)

  (3) $\Sigma';\Gamma \vdash v_1 := e_2' : unit$ (by (1), (2) and T-Assign)

- Subcase E-Assign: $\dfrac{}{(M, l := v) \to (M[l \mapsto v]), ()}$

  (1) Let $\Sigma' = \Sigma, l : t$

  (2) $\Sigma';\Gamma \vdash () : unit$ (by T-unit)

  (3) $\Sigma;\Gamma \vdash M$ (by I.H.)

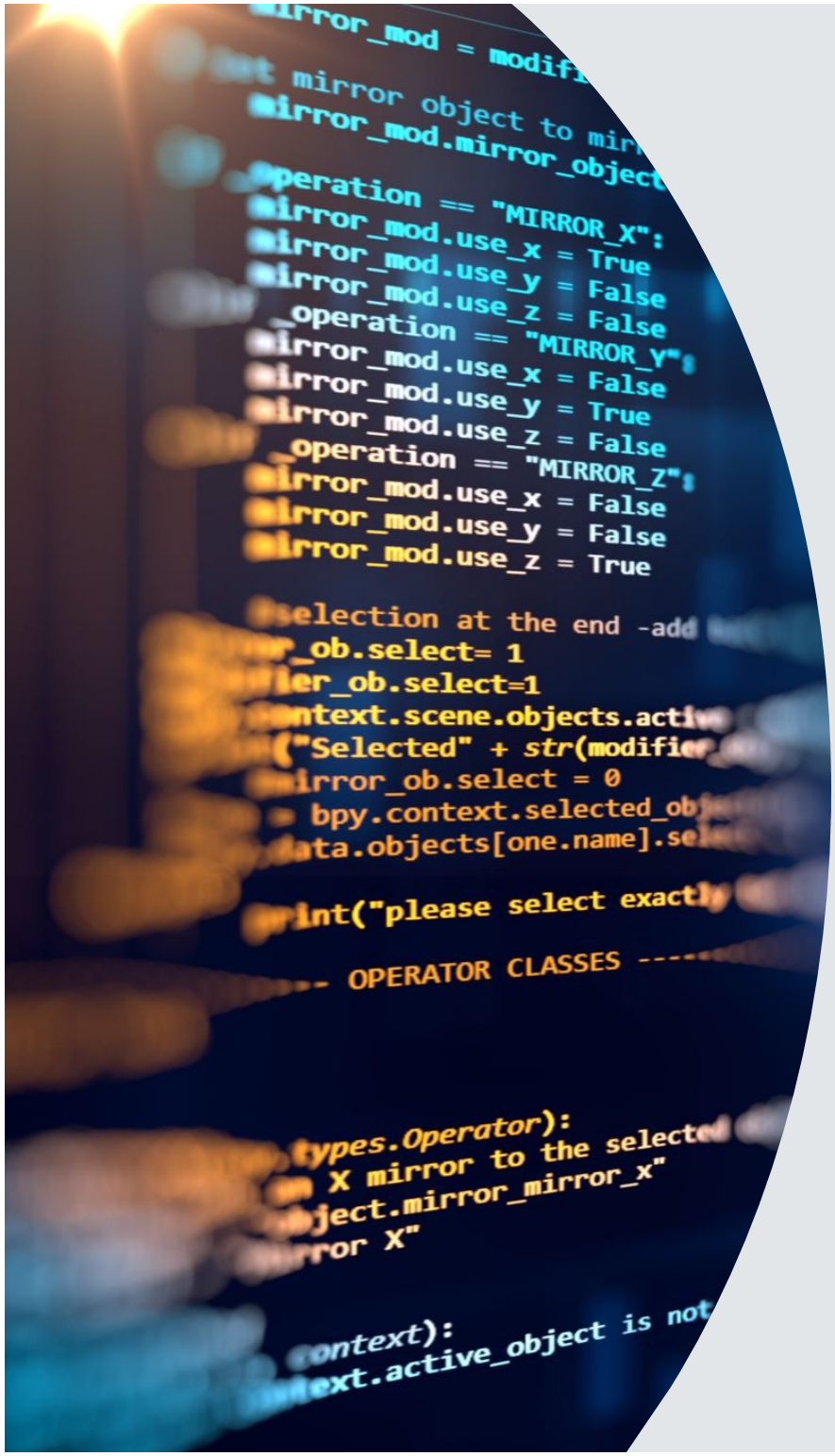
  (5) $\Sigma'(l) = t$ and $M'(l) = v$

  (6) $\Sigma', \Gamma \vdash v : t$ (by assumption and **Lemma 2**)

  (7) $\Sigma';\Gamma \vdash M'$ (by definition of $\Sigma;\Gamma \vdash M$)

CSE3302/5307 Programming Language (Concepts) Project Specification

TA-Sinong



Introduction

- You are required to implement an *interpreter* for the programming language **SimPL** (pronounced *simple*)
- SimPL is a simplified dialect of ML

Lexical Definition

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

Lexical Definition

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

Lexical Definition

- 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

Lexical Definition

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

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*.

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

Operator Precednece

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

Typing

Arbitrary type t	::=	
	|	`int`
	|	`bool`
	|	`unit`
	|	t `list`
	|	t `ref`
	|	$t \times t$
	|	$t \to t$

Equality type α	::=	
	|	`int`
	|	`bool`
	|	α `list`
	|	t `ref`
	|	$\alpha \times \alpha$

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

- `int` $\in$ **Typ**, `bool` $\in$ **Typ**, `unit` $\in$ **Typ**
- $\forall\, t \in$ **Typ**, t `list` $\in$ **Typ**
- $\forall\, t \in$ **Typ**, t `ref` $\in$ **Typ**
- $\forall\, t_1, t_2 \in$ **Typ**, $t_1 \times t_2 \in$ **Typ**
- $\forall\, t_1, t_2 \in$ **Typ**, $t_1 \to t_2 \in$ **Typ**

Typing

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.

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

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

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

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

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

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

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

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

Semantics

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

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

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

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

Semantics

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

$$\mathbf{Val} = \bigcup_{t \in \mathbf{Typ}} \mathcal{V}_t$$

$$\mathcal{V}_{\texttt{unit}} = \{\texttt{unit}\}$$

$$\mathcal{V}_{\texttt{int}} = \mathbb{Z} = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$$

$$\mathcal{V}_{\texttt{bool}} = \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$$

$$\mathcal{V}_{t\ \texttt{list}} = \bigcup_{i=0}^{\infty} \mathcal{V}_{t\ \texttt{list}}^{(i)}$$

$$\mathcal{V}_{t\ \texttt{list}}^{(0)} = \{\texttt{nil}\}$$

$$\mathcal{V}_{t\ \texttt{list}}^{(i+1)} = \{\texttt{cons}\} \times \mathcal{V}_t \times \mathcal{V}_{t\ \texttt{list}}^{(i)}$$

$$\mathcal{V}_{t\ \texttt{ref}} = \{\texttt{ref}\} \times \mathbb{N}$$

$$\mathcal{V}_{t_1 \times t_2} = \{\texttt{pair}\} \times \mathcal{V}_{t_1} \times \mathcal{V}_{t_2}$$

$$\mathcal{V}_{t_1 \to t_2} = \mathcal{V}_{t_1}^{\mathcal{V}_{t_2}} \subseteq \{\texttt{fun}\} \times \mathbf{Env} \times \mathbf{Var} \times \mathbf{Exp}$$

$$\mathbf{Rec} = \{\texttt{rec}\} \times \mathbf{Env} \times \mathbf{Var} \times \mathbf{Exp}$$

Semantics

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

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

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

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

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

Examples

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

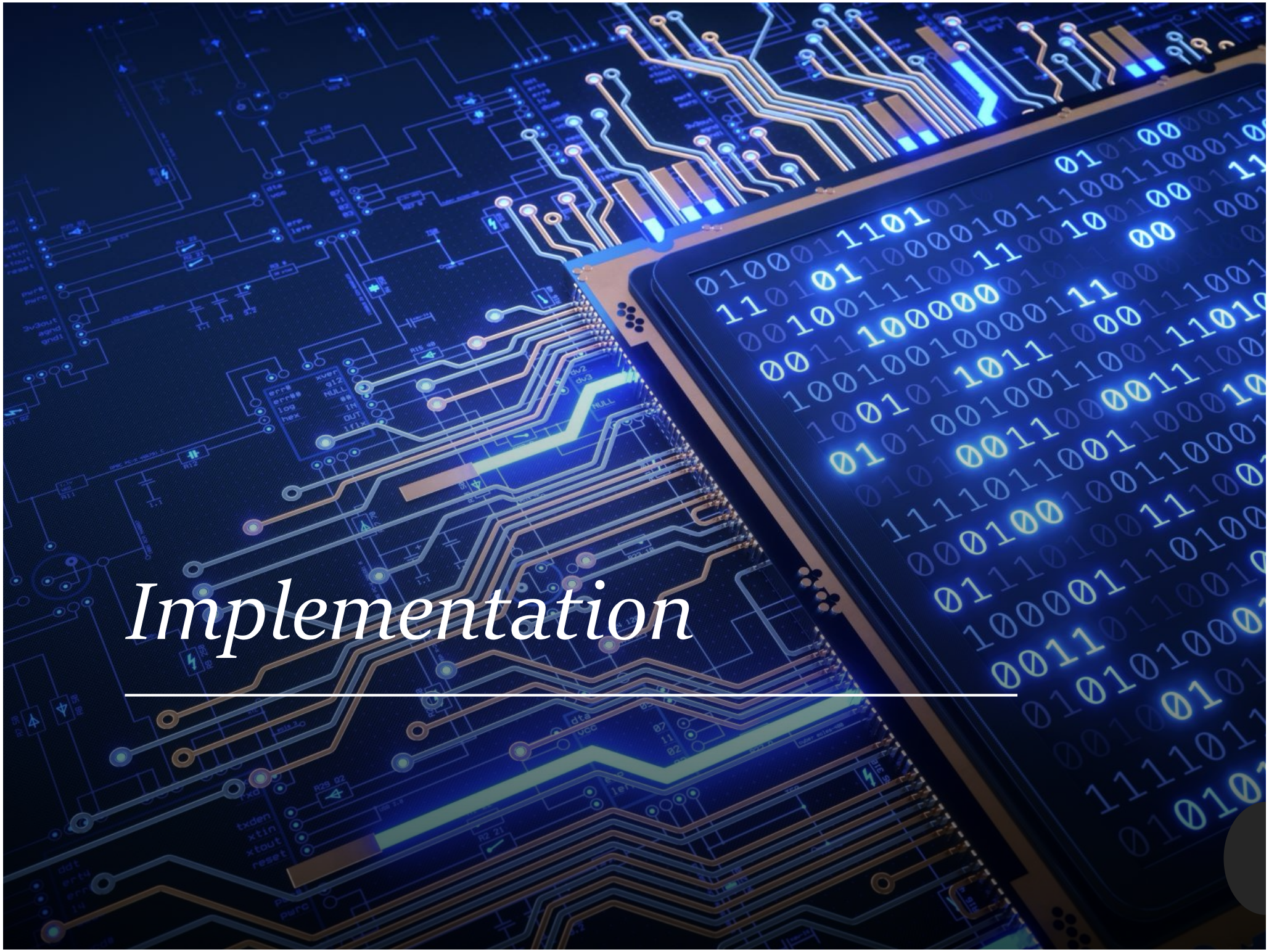
examples/plus.spl

```
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


Implementation

Command-line Interface

- You are required to implement the SimPL interpreter in Java
- And submit a runnable JAR file, say SimPL.jar
- Input parameter: the path of the SimPL program file, *.spl
- Output: the result of the execution to the standard output (System.out

Command-line Interface

- 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`.

Command-line Interface

- 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.

Predefined Functions

- *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.

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

Bonus (TBD)

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

Report

Represent your Procedure

Academic Integrity

Please add the following statement and sign it at the beginning of your report.

I have neither given nor received unauthorized assistance on this work. I will not post the project description and the solution online.

Sign: *Data:*

<u>If there is no statement like this in your report, you will lose 60% points!!</u>

Content

- *Why and how to implement basic functionality*
- *Why and how to implement Bonus functionality*
- *Figures are the best demonstration*
- *LaTex is highly recommended*

Let's have a look at Skeleton