PPL252

Assignment 2

Responsible Lecturers: Meni Adler, Yaron Gonen
Responsible TA: Israel Zexer

Submission Date: 18/5/2025

General Instructions

Submit your answers to the theoretical questions in a pdf file called id1_id2.pdf and your code in the src folder, and ZIP those files together into a file called *id1_id2.zip*.
Do not send assignment related questions by e-mail, use the forum instead. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.

You are provided with the templates *ex2.zip*.
Unpack the template files inside a folder. From the command line in that folder, invoke `npm install`, and work on the files in that directory, preferably working in the Visual Studio Code IDE (refer to the Useful Links). In order to run the tests, run `npm test` from the command line.

Important: Do not add any extra libraries and do not change the provided package.json and tsconfig.json configuration files. **The graders will use the exact provided files**. If you find any missing necessary libraries, please let us know.

Question 1: Language variations [7 points]

Q1.1 Let us define the L11 language as L1 excluding the special form 'define'. Is there a program in L1 which cannot be transformed to an equivalent program in L11? Explain or give a contradictory example [1 point]

Q1.2 Let us define the L21 language as L2 excluding the special form 'define'. Is there a program in L2 which cannot be transformed to an equivalent program in L21? Explain or give a contradictory example [2 points]

Q1.3 Let us define the L22 language as L2, where procedures (`lambda`) can only have one parameter and a body with one expression. Is there a program in L2 which cannot be transformed to an equivalent program in L22? Explain or give a contradictory example [2 points]

Q1.4 Let us define the L23 language as L2, where procedures (`lambda`) are first-order, *i.e.* cannot get functions as arguments. Is there a program in L2 which cannot be transformed to an equivalent program in L23? Explain or give a contradictory example [2 points]

Write your answers in file id1_id2.pdf

Question 2: Adding Dictionary to L3 [73 points]

The 'L3' directory in the assignment template contains the parser of L3 and the interpreter for both substitution and environment models.
In this question we extend L3 with a 'dictionary' expression, in the same manner as JavaScript's Map.

2.1 First implementation: as primitive operators [15 points]

In order to support dictionaries, we define primitive operators for dictionaries:

`dict` - constructs a dictionary
`get` - gets value from a dictionary according to a given key
`dict?` - checks if a given expression is a dictionary

```
(dict <lit-exp>)

(dict '((a . 1) (b . 2)))
→ '((a . 1) (b . 2))

(get (dict '((a . 1) (b . 2))) 'a)
→ 1

(get (dict '((a . 1) (b . 2))) 'c)
→ Error…

(dict? (dict '((a . 1) (b . 2))))
→ #t

(dict? '((a . 1) b))
→ #f

(dict? '((a . 1) (b)))
→ #t
['(b) is actually '(b . '())]
```

a. Extend L3 concrete and abstract syntax with the new primitives

```
<program> ::= (L3 <exp>+)              / Program(exps:List(exp))
<exp> ::= <define> | <cexp>             / DefExp | CExp
<define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                  / VarRef(var:string)
<cexp> ::= <number>                     / NumExp(val:number)
        |  <boolean>                    / BoolExp(val:boolean)
        |  <string>                     / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
        |                                /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
        |                                      then: CExp,
        |                                      alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
        |                                          body:CExp[]))
        |  ( quote <sexp> )                 / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )               / AppExp(operator:CExp,
        |                                             operands:CExp[]))
<binding>  ::= ( <var> <cexp> )             / Binding(var:VarDecl,
        |                                              val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                 | cons | car | cdr | list | pair? | list? | number?
                 | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

Write your answer in file id1_id2.pdf

b. Extend L3 parser with the new primitives [1 point]

c. Extend L3 interpreter (applicative order, substitution model) with the new primitives

   Note: The correctness check of the operators' parameter should be applied as part of the semantics (*i.e.*, in the interpreter), rather as part of the syntax (*i.e.*, in the parser)

The code should be submitted in directory src/L31

You can test your code with test/q21.tests.ts

2.2 Second implementation: as a special form [15 points]

Implement dictionary as a special form: the 'dict’ special form is evaluated to a new DictValue value. A DictValue can be applied with a key in order to get its value:

```
(define d (dict (a 1) (b 2)))
(d 'a)
→ 1

(d 'b)
→ 2

(d 'c)
→ Error…
```

a. Extend L3 concrete and abstract syntax with the new special form [1 point]

```
<program> ::= (L3 <exp>+)              / Program(exps:List(exp))
<exp> ::= <define> | <cexp>             / DefExp | CExp
<define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                  / VarRef(var:string)
<cexp> ::= <number>                     / NumExp(val:number)
        |  <boolean>                    / BoolExp(val:boolean)
        |  <string>                     / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
        |                                /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
        |                                      then: CExp,
        |                                      alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
        |                                         body:CExp[]))
        |  ( quote <sexp> )                 / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )               / AppExp(operator:CExp,
        |                                          operands:CExp[]))
<binding>  ::= ( <var> <cexp> )            / Binding(var:VarDecl,
        |                                           val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                 | cons | car | cdr | list | pair? | list? | number?
                 | boolean? | symbol? | string?
```

```
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

Write your answer in file id1_id2.pdf

b. Extend L3 parser with the new special form.

c. Extend L3 interpreter (applicative order, substitution model) with the new special form.

   Note: The correctness check of the *dict* special form parameters should be applied as part of the as part of the syntax (*i.e.*, in the parser), rather semantics (*i.e.*, in the interpreter)

The code should be submitted in directory src/L32
You can test your code with test/q22.tests.ts

2.3 Third implementaion: as L3 user procedures [10 points]

Write L3 code which supports dictionaries

- `dict, get, dict?` procedures
- Error handling - procedures for error values: `make-error, is-error?, bind`

```
(dict '((a . 1) (b . 2)))
→ '((a . 1) (b . 2))

(get (dict '((a . 1) (b . 2))) 'a)
→ 1

(is-error? (get (dict '((a . 1) (b . 2))) 'c))
→ #t

(dict? '((a . 1) (b . 2)))
→ #t

(bind (get (dict '((a . 1) (b . 2))) 'b) (lambda (x) (* x x)))
→ 4
```

The code should be submitted in file src/q23.l3
You can test your code with test/q23.tests.ts

2.4 Theoretical questions [18 points]

a. Should your implementations for the three dictionary versions(2.1, 2.2, 2.3) be modified for the case of normal order? [3 points]
b. Should your implementations for the three dictionary versions(2.1, 2.2, 2.3) be modified for the case of the environment model? [3 points]
c. Explain why the `dict` primitive-operator (2.1) and user-procedure (2.3) cannot get the dictionary fields in the same way as the `dict` special form (2.2). *i.e.* `(dict '((a . 1) (b . 2)))` but not `(dict (a 1) (b 2))`
   In your answer, refer to parsing as well as interpreting, for both applicative and normal order [5 points]
d. Are there expressions which can be defined as a field's value in `dict` special form (2.2) but not in `dict` primitive operator (2.1) or user procedure (2.3)?
   Are there expressions in L32 which cannot be transformed to equivalent expressions in L3 (according to 2.5 method below)?
   [2 points]
e. Which of the three dictionary implementations (2.1, 2.2, 2.3) would you prefer? List advantages and disadvantages. [5 points]

Write your answers in file id1_id2.pdf

2.5 Syntactic Transformation [15 points]

a. Implement Dict2App procedure, which gets an L32 program (the second dictionary implementation of Q2.2) and replace each DictExp with `dict` with AppExp, where the operator is a VarRef named `dict` and the operand is a list of key-val:

```
(dict (a 1) (b 2))
→ (dict '((a . 1) (b . 2)))
```

b. Implement L32ToL3 procedure, which gets an L32 program and returns an equivalent L3 program.
   Note: In the transformed program, in order to get the value of a given key, the dictionary is applied with this key, as done in 2.2:

```
( (dict (a 1) (b 2)) 'a )
→ 1
```

The code should be submitted in file src/q24.l3
You can test your code with test/q24.tests.ts

Question 3: Code translation [10 points]

Write a procedure *l2ToJS*. The procedure gets an L2 AST and returns a string of the equivalent JavaScript program.

For example:

```
(+ 3 5 7)
→ (3 + 5 + 7)

(= 3 (+ 1 2))
→ (3 === (1 + 2))

(if (> x 3) 4 5)
→ ((x > 3) ? 4 : 5)

(lambda (x y) (* x y))
→((x,y) => (x * y))

((lambda (x y) (* x y)) 3 4)
→((x,y) ⇒ (x * y))(3,4)

(define pi 3.14)
→ const pi = 3.14

(define f (lambda (x y) (* x y)))
→ const f = ((x,y) ⇒ (x * y))

(f 3 4)-->
f(3,4)
```

Notes:

- The primitive operators of L2 are: `+, -, *, /, <, >, =, number?, boolean?, eq?, and, or, not`
  You can see their exact semantics in the applyPrimitive function in the interpreter of L3.
- You can assume that the body of the lambda contains one expression

Hint: Take a look at the unparse procedure.

The code should be submitted in file src/q3.ts

You can test your code with test/q3.tests.ts

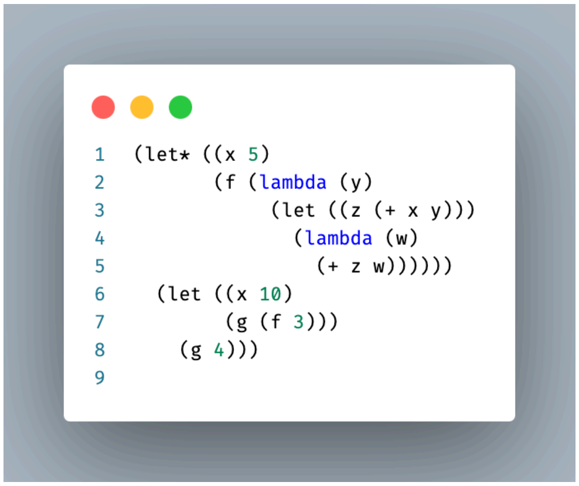
4 Environment diagram [10 points]

4a.

Read about `let*` here.

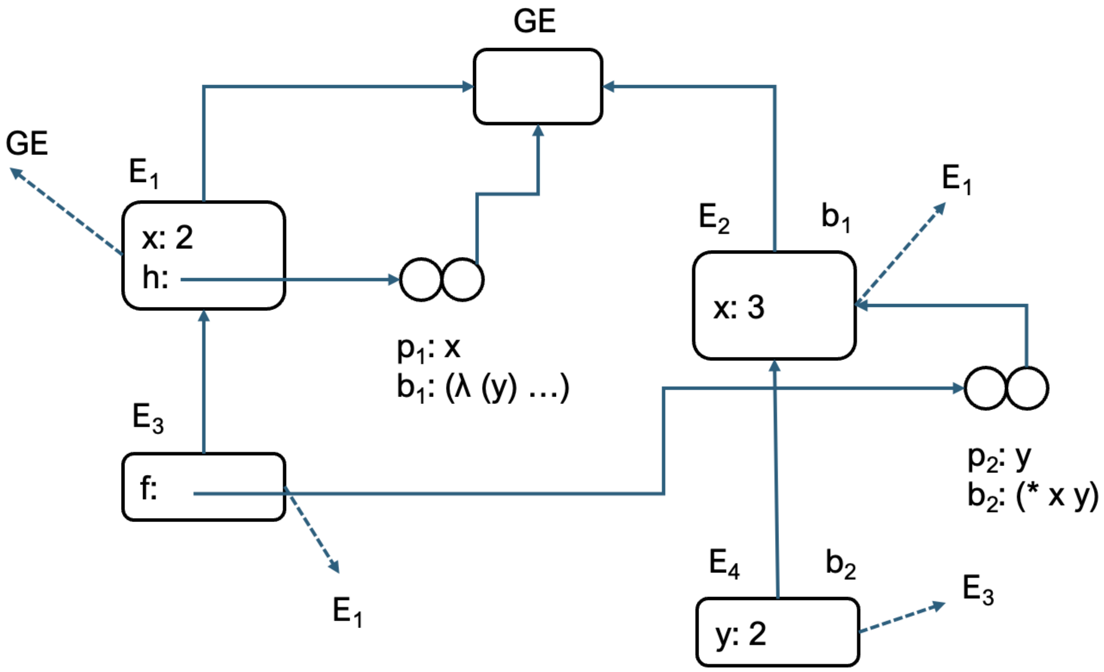
Draw the **environment diagram** showing the evaluation of this program. Your diagram should include:

1. The **global environment**.
2. Any **closure objects** created during evaluation, clearly showing their **parameter(s)**, **body**, and **defining environment**.
3. Any **frames** created as a result of `let,` `let*` or `lambda` application.
4. All the **control links**.
5. The **final result** of the expression.



```
(let* ((x 5)
       (f (lambda (y)
            (let ((z (+ x y)))
              (lambda (w)
                (+ z w))))))
  (let ((x 10)
        (g (f 3)))
    (g 4)))
```

4.2 Define a program which fits the following environment diagram [5 points]



Answers should be submitted in file id1_id2.pdf

Good Luck!