CS 496: Homework Assignment 5
Due: 29 April, 2017, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

In this assignment you are asked to extend the interpreter for the language EXPLICIT-REFS: with pairs, lists and trees. This requires:

- Extending the syntax of the language for expressions (pairs, lists and trees)
- Extending the syntax of the language for types (references, pairs, lists and trees)

The new language will be called T-EXPLICIT-REFS. Note that since the language seen in class, CHECKED, is an extension of REC with type-checking, the first step in this assignment is to adapt the type-checker of CHECKED to EXPLICIT-REFS, as explained next.

3 Adapting the Type-Checker of CHECKED to EXPLICIT-REFS ✓

One new grammar production is needed for the concrete syntax of expressions.

<Expression> ::= unitE

Note that this requires changing `lang.scm` as follows so that the scanner and parser can recognize it:

```
(expression
  ("unitE")
  unit-exp)
```

The set of expressed values should include a value for the unit expression:

```
(define-datatype expval expval?
  (num-val
   (value number?))
  (bool-val
   (boolean boolean?))
  (proc-val
   (proc proc?))
  (unit-val)
```

The concrete syntax of types must be extended so as to allow the following typing rules to be implemented:

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{newref(e)} :: \texttt{ref(t)}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{ref(t)}}{\texttt{tenv} \vdash \texttt{deref(e)} :: \texttt{t}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{ref(t)} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{setref(e1,e2)} :: \texttt{unit}}$$

$$\frac{}{\texttt{tenv} \vdash \texttt{unitE} :: \texttt{unit}}$$

Note the use of the type `unit`, to indicate that the return result of the assignment operation is not important. Also, you must change the interpreter so that rather than returning the dummy value `num-val 23` as the return value for an assignment, it now returns `unit-val`.

The concrete syntax of types is thus extended with two new productions

```
<Type>  ::=  int
<Type>  ::=  bool
<Type>  ::=  unit
<Type>  ::=  ref(<Type>)
<Type>  ::=  (<Type> -> <Type>)
```

This requires updating the `lang.scm` file with two new cases:

```
(type
  ("unit")
  unit-type)

(type
```

```
  ("ref" "(" type ")")
  ref-type)
```

You also have to update the code for the typechecker by adapting the file `checker.scm` of the CHECKED to EXPLICIT-REFS so that `type-of` can handle :

```
(newref-exp (exp)
  (write "implement me"))

(deref-exp (exp)
  (write "implement me"))

(setref-exp (id exp)
  (write "implement me"))
```

4 Pairs ✓

4.1 Concrete syntax

Two new productions are added to the grammar of EXPLICIT-REFS:

<Expression> ::= pair(<Expression>,<Expression>)
<Expression> ::= unpair (<Identifier>,<Identifier>)=<Expression> in <Expression>

Some examples of programs using pairs follow:

```
pair(3,4)

pair(pair(3,4),5)

pair(zero?(0),3)

pair(proc (x:int) -(x,2),4)

proc (z:<int*int>) unpair (x,y)=z in x

proc (z:<int*bool>) unpair (x,y)=z in pair(y,x)
```

Note that the concrete syntax of the types is also extended with a new production:

<Type> ::= int
<Type> ::= bool
<Type> ::= ref(<Type>)
<Type> ::= (<Type> -> <Type>)
<Type> ::= <<Type> * <Type>>

Regarding the behavior of these expressions they are clear. For example, the expression `proc (z:<int*int>) unpair (x,y)=z in x` is a function that given a pair of integers, projects the first component of the pair.

```
> (run "(proc (z:<int*int>) unpair (x,y)=z in x pair(2,3))")
(num-val 2)
```

Likewise, the last expression swaps the two components of a pair.

```
> (run "(proc (z:<int*int>) unpair (x,y)=z in pair(y,x) pair(2,3))")
(pair-val (num-val 3) (num-val 2))
```

Updating the Grammar File for the Parser Generator

You should add the following lines, in the `lang.scm` file, as new cases in the definition of the function `the-grammar`.

```
(expression
  ("pair"
   "(" expression "," expression ")")
  pair-exp)

(expression
  ("unpair"  "(" identifier "," identifier ")" "=" expression
             "in" expression)
 unpair-exp)
```

Also, you should add the new cases for the grammar of types. It goes in the same file and function definition:

```
(type
  ("<" type "*" type ">")
  pair-type)
```

Finally, update the function `type-to-external-form` so that it pretty-prints pair types correctly. This is also in the file `lang.scm`.

All this will allow you to parse expressions such as the ones given above using `scan&parse`.

4.2 Expressed Values

In preparation for extending the interpreter to deal with pairs, you should first extend the set of *expressed values*. You should replace the definition of `expval` in the fiel `data-structures.scm` with:

```
(define-datatype expval expval?
  (num-val
   (value number?))
  (bool-val
   (boolean boolean?))
  (proc-val
   (proc proc?))
  (unit-val)
  (pair-val
```

```
    (fst expval?)
    (snd expval?)))
```

Also, add the functions

```
(define expval->fst
  (lambda (v)
    (cases expval v
      (pair-val (e1 e2) e1)
      (else (expval-extractor-error 'pairFst v)))))

(define expval->snd
  (lambda (v)
    (cases expval v
      (pair-val (e1 e2) e2)
      (else (expval-extractor-error 'pairSnd v)))))
```

These will come in handy when you implement the interpreter.

4.3 Interpreter

You should add two new cases in the definition of the function `value-of` in the file `interp.scm`.

```
(pair-exp (exp1 exp2)
          (write "implement me"))
(unpair-exp (id1 id2 exp1 body)
          (write "implement me"))
```

4.4 Type Checker

Add two new cases to the definition of the function `type-of` in the file `checker.scm`:

```
(pair-exp (exp1 exp2)
  (write "implement me"))

(unpair-exp (id1 id2 exp1 body)
  (write "implement me"))
```

The typing rules you should follow are those that you gave as solution to Exercise 6 in Exercise Booklet 8.

5 Lists ✓

5.1 Concrete syntax

The concrete syntax for expressions should be extended with the following productions:

<Expression>	::=	emptylist <Type>
<Expression>	::=	cons (<Expression>, <Expression>)
<Expression>	::=	null? (<Expression>)
<Expression>	::=	car (<Expression>)
<Expression>	::=	cdr (<Expression>)

The concrete syntax for types includes one new production (the last one listed below):

<Type>	::=	`int`
<Type>	::=	`bool`
<Type>	::=	`unit`
<Type>	::=	`ref(`<Type>`)`
<Type>	::=	`(`<Type> `->` <Type>`)`
<Type>	::=	`<<`<Type> * <Type>`>>`
<Type>	::=	`list(`<Type>`)`

The new type constructor is for typing lists. For example,

1. `list(int)` is the type of lists of integers
2. `(int -> list(int))` is the type of functions that given an integers produce a list of integers.

Here are some sample expressions in the extended language. They are supplied in order you to help you understand how each construct works.

```
> (run "cons(1,emptylist int)")
(list-val (list (num-val 1)))

> (run "cons(cons(1,emptylist int ),emptylist list(int))")
(list-val (list (list-val (list (num-val 1)))))

> (run "let x = 4
      in cons(cons(-(x,1),
                           emptylist int),
                     emptylist list(int))")
(list-val (list (list-val (list (num-val 3)))))

> (run "null?(emptylist list(int))" )
(bool-val #t)

> (run "cdr(cons(cons(1,emptylist int),emptylist list(int)))")
(list-val '())
```

Remember to extend the interpreter and then the type-checker.

5.2 Typing rules

$$\frac{}{\texttt{tenv} \vdash \texttt{emptylist t} :: \texttt{list(t)}} \qquad \frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{cons(e1,e2)} :: \texttt{list(t)}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{cdr(e)} :: \texttt{list(t)}} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{car(e)} :: \texttt{t}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{null?(e)} :: \texttt{bool}}$$

6 Trees

6.1 Concrete syntax

The concrete syntax for expressions should be extended with the following productions:

<Expression>	::=	emptytree <Type>
<Expression>	::=	node (<Expression>, <Expression>, <Expression>)
<Expression>	::=	nullT? (<Expression>)
<Expression>	::=	getData (<Expression>)
<Expression>	::=	getLST (<Expression>)
<Expression>	::=	getRST (<Expression>)

The concrete syntax for types includes one new production (the last one listed below):

<Type>	::=	`int`
<Type>	::=	`bool`
<Type>	::=	`unit`
<Type>	::=	`ref(`<Type>`)`
<Type>	::=	`(`<Type> `->` <Type>`)`
<Type>	::=	`<<`<Type> `*` <Type>`>>`
<Type>	::=	`list(`<Type>`)`
<Type>	::=	`tree` (<Type>)

Here is an example program that assumes you have implemented the `append` operation on lists.

```
letrec list(int) append(xs:list(int)) =
       proc (ys:list(int))
          if null?(xs)
            then ys
            else cons(car(xs),((append cdr(xs)) ys)) in
letrec list(int) inorder(x:tree(int)) =
  if nullT?(x)
    then emptylist int
    else ((append (inorder getLST(x))) cons(getData(x),
     (inorder getRST(x))))
in
```

```
(inorder node(1,node(2,emptytree int,emptytree int),emptytree int))
```

Remember to extend the interpreter and then the type-checker.

6.2 Typing rules

$$\frac{}{\texttt{tenv} \vdash \texttt{emptytree t} :: \texttt{tree(t)}} \qquad \frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{tree(t)} \quad \texttt{tenv} \vdash \texttt{e3} :: \texttt{tree(t)}}{\texttt{tenv} \vdash \texttt{node(e1,e2,e3)} :: \texttt{tree(t)}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{tree(t)}}{\texttt{tenv} \vdash \texttt{getData(e)} :: \texttt{t}} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{tree(t)}}{\texttt{tenv} \vdash \texttt{getLST(e)} :: \texttt{tree(t)}} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{tree(t)}}{\texttt{tenv} \vdash \texttt{getRST(e)} :: \texttt{tree(t)}}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{tree(t)}}{\texttt{tenv} \vdash \texttt{nullT?(e)} :: \texttt{bool}}$$

7 Submission instructions

Submit a file named `HW5_<SURNAME>.zip` through Canvas which includes all the source files required to run the interpreter and type-checker. Please include your name in the file `top.scm`.