CS 496: Homework Assignment 6
Due: 25 April 2018, 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 typer-checker for the language REC with references, pairs, lists and trees. This requires:

- Extending the syntax of the language for expressions (references, 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 `parser.mly` with a new case of expr as follows so that the scanner and parser can recognize it:

```
  | UNIT { Unit }
```

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

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | RefVal of int
  | UnitVal
```

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. We assumed that the interpreter was changed so that rather than returning the dummy value `NumVal 28` as the return value for an assignment, it now returns `UnitVal`.

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 `parser.mly` file with two new cases of texpr:

```
  | UNITTYPE { UnitType }
  | REFTYPE; t = texpr { RefType(t) }
```

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

```
type_of_texpr = function
  ...
  | Unit ->
```

```
      UnitType
  ...
  | NewRef (e) ->
      failwith "Implement me!"
  | DeRef (e) ->
      failwith "Implement me!"
  | SetRef(e1, e2) ->
      failwith -> 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 - 1 }, 4)

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

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

let f = proc (z:<int*bool>) { unpair (x,y)=z in pair(y,x) } in (f pair
    (1, zero?(0)))
```

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 pair(2, 3))` is a function that given a pair of integers, projects the first component of the pair.

```
# Checker.chk "(proc (z:<int*int>) {unpair (x,y)=z in x} pair(2, 3))"
    ;;
- : Ast.texpr = Ast.IntType
```

Updating the Grammar File for the Parser Generator

You should add the following lines, in the `parser.mly` file, as productions in the definition of the nonterminal `expr`.

```
  | PAIR; LPAREN; e1 = expr; COMMA; e2 = expr; RPAREN { Pair(e1, e2) }
  | UNPAIR; LPAREN; id1 = ID; COMMA; id2 = ID; RPAREN; EQUALS; e_pair = expr; IN; e_body
    = expr { Unpair(id1, id2, e_pair, e_body) }
```

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

Also, you should add the new cases for the grammar of types. It goes in the same file but is added as a new production for the nonterminal `texpr`:

```
  | LESS_THAN; t1 = texpr; TIMES; t2 = texpr; GREATER_THAN { PairType(t1, t2) }
```

Finally, update the function `string_of_texpr` so that it pretty-prints pair types correctly. It is in the file `ast.ml`.

4.2 Expressed Values

In preparation for extending the interpreter to deal with pairs, you should first extend the set of *expressed values*. The set of expressed values should include a value for the pair expression. `ds.ml` with:

```
type exp_val =
  | NumVal of int
  | Boolval of bool
  | ProcVal of string*Ast.expr*env
  | RefVal of int
  | UnitVal
  | PairVal of exp_val*exp_val
```

Also, add the observers

```
let pairVal_to_fst = function
  | PairVal (exp_val1, exp_val2) -> exp_val1
  | _ -> failwith "Expected a pair!"

let pairVal_to_snd = function
  | PairVal (exp_val1, exp_val2) -> exp_val2
  | _ -> failwith "Expected a pair!"
```

These will come in handy when you implement the interpreter. Remember to update the function `string_of_expval` in the file `ds.ml` for pretty-prints.

4.3 Type Checker

Add two new cases to the definition of the function `type_of_expr` in the file `checker.ml`:

```
  | Pair(e1, e2) ->
      failwith "implement me"
  | Unpair(e1, e2) ->
```

```
4         failwith "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> ::= hd (<Expression>)
<Expression> ::= tl (<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 constructor works.

```
1  # Checker.chk "emptylist int";;
2  - : Ast.texpr = Ast.ListType Ast.IntType
3
4  # Checker.chk "cons(1, emptylist int)";;
5  - : Ast.texpr = Ast.ListType Ast.IntType
6
7  # Checker.chk "hd(cons(1, emptylist int))";;
8  - : Ast.texpr = Ast.IntType
9
10 # Checker.chk "tl(cons(1, emptylist int))";;
11 - : Ast.texpr = Ast.ListType Ast.IntType
12
13 # Checker.chk "cons(null?(emptylist int), emptylist int)";;
```

```
Exception: Failure "Checker: cons: type of head and tail do not match"
    .
```

Remember that you need to extend the parser in `parser.mly`, the expressed values and the observers in `ds.ml`, `string_of_texpr` in `ast.ml`, and then the type-checker in `checker.ml`. The concrete syntax and the examples given above should give you enough information to make the extension.

5.2 Typing rules

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

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{tl(e)} :: \texttt{list(t)}} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{list(t)}}{\texttt{tenv} \vdash \texttt{hd(e)} :: \texttt{t}}$$

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

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

```
# Checker.chk "emptytree int";;
- : Ast.texpr = Ast.TreeType Ast.IntType
```

```
# Checker.chk "nullT?(node(1, node(2, emptytree int, emptytree int),
    emptytree int))";;
- : Ast.texpr = Ast.BoolType

# Checker.chk"getData(node(1, node(2, emptytree int, emptytree int),
    emptytree int))";;
- : Ast.texpr = Ast.IntType

# Checker.chk "getLST(node(1, node(2, emptytree int, emptytree int),
    emptytree int))";;
- : Ast.texpr = Ast.TreeType Ast.IntType
```

Here is a sample program that assumes you have implemented the `append` operation on lists. Note that you will not be able to run this program unless you have extended the interpreter to deal with lists and trees. This assignment only asks you to write the type-checker, not the interpreter. You are, of course, encouraged to write the interpreter too!

```
letrec list(int) append(xs:list(int)) =
       proc (ys:list(int))
          if null?(xs)
            then ys
            else cons(hd(xs),((append tl(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 that you need to extend the parser in `parser.mly`, the expressed values and the observers in `ds.ml`, `string_of_texpr` in `ast.ml`, and then the type-checker in `checker.ml`. The concrete syntax and the examples given above should give you enough information to make the extension.

6.2 Typing rules

$$\frac{}{\texttt{tenv} \vdash \texttt{emptytree t} :: \texttt{tree(t)}}$$

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

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

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

7 Submission instructions

Submit a file named `HW6_<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 `checker.ml`. Your grade will be determined as follows, for a total of 100 points:

Section	Grade
Unit	5
Reference	15
Pair	20
List	30
Tree	30