Swinburne University Of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code:	COS30023
Subject Title:	Languages in Software Development
Assignment number and title:	7, Simple Lambda Calculus Language
Due date:	**October 20, 2014, 10:30, on paper**
Lecturer:	Dr. Markus Lumpe

Your name:____________________

Marker's comments:

Problem	Marks	Obtained
1	2+6+2+7+14+18+19=68	
3	9	
Total	77	

Extension certification:

This assignment has been given an extension and is now due on __________________

Signature of Convener:_________________________

Problem Set 7: Lambda Calculus Language

Problem 1

Add the `reduce` method to the abstract syntax classes developed in tutorial 8.

```
package ast;

import java.util.Set;
import java.util.Hashtable;

public abstract class LCLExpression
{
 public abstract Set<String> freeNames();

 public abstract LCLExpression substitute( String aVar, LCLExpression aExp );

 public abstract LCLExpression reduce( Hashtable<String,LCLExpression> aSymTable );

 public abstract String toString();
}
```

The method `reduce` implements "head-normal form applicative-order reduction." Follow the rules for applicative-order reduction given in class (see lecture notes page 198ff.). In our interpreter every function (i.e., a `LambdaFunction` object) is in "head normal form." Technically, a lambda term is in head normal form (cf. lecture notes page 201) if it starts with an abstraction of the form `(lambda x . e)`. So, when `reduce` is applied to a `LambdaFunction` object, it just has to return the very same object. (In order words, the method `reduce` for a `LambdaFunction` object is the identity function, i.e., f x = x.)

In order to evaluate a given term, the method `reduce` takes a symbol table as argument. The symbol table provides a lookup environment to resolve the meaning of free occurrences of variables in lambda terms. The reduction of the LCL expressions is as follows:

- A `LambdaNumber` is already in head normal form. Hence, `reduce` has to return the very same object (i.e., `this`).
- A `LambdaVariable` denotes a free occurrence of a name in an LCL expression. To resolve this name, we have to look it up in `aSymTable`. If the name is not defined in `aSymTable`, we need to raise a `RuntimeException` with a proper error message. Otherwise, `reduce` for `LambdaVariable` returns the LCL expression bound to the variable name in `aSymTable`.
- A `LambdaFunction` is in head normal form. Hence, `reduce` has to return the very same object (i.e., `this`).
- In a declaration (i.e., `LCLDeclaration`) we need to evaluate the sub-expression first. If the symbol table already contains a binding for the declared name, then we replace it with the result of the sub-expression. Otherwise, we just add a new binding. We return the evaluated (i.e., reduced) sub-expression.
- The evaluation of an if-then-else (i.e., `IfThenElse`) first reduces the condition. If the value of the condition is different from 0, then we return the result of the then-expression. Otherwise, we evaluate the else-expression. The condition must evaluate to a `LambdaNumber`. If not, we need to raise a `RuntimeException` with a proper error message.

You may need to add a `getValue` function to `LambdaNumber` in order to access the denoted integer value.

- A `LambdaApplication` is evaluated by first reducing its components to head normal form. If the first component (i.e., the function) evaluates to an object of type `LambdaFunction`, apply β-reduction (cf. Lecture notes page 173) and reduce the resulting expression to head normal form.

  If the first component does not evaluate to a `LambdaFunction` object, `reduce` has to return a new `LambdaApplication` object made up of the reduced (i.e., evaluated) components.

- The load expression creates a local interpreter object (i.e., an object of type `LCLParser`) to parse the contents of the file denoted by the string component. We use "`.lam`" as default extension. The result of parsing the included file is an array list of `LCLExpression` objects. The objects (i.e., LCL expressions) need to be evaluated using `reduce` also. Use a for-each loop for this purpose. The result of the last evaluation has to be returned as the result of evaluating a load expression.

  The result of the evaluation of a load declaration is twofold. If the included file contains any lambda declarations, then the symbol table is updated with corresponding bindings. The value of the evaluation of a load declaration is the value of the last expression occurring in the imported file.

  Please note that a `ParseException` and a `FileNotFoundException` may occur. You have to catch them and throw instead a `RuntimeException`.

Problem 2

Define the built-in lambda expressions `Increment`, `Decrement`, `Zero`, and `NotZero`. The expression `Increment` implements the successor function, `Decrement` the predecessor function, `Zero` a test for 0, and `NotZero` a test for not 0 for integers. The test primitives have to return a `LambdaNumber` with 1 for true and 0 for false. Follow the format as shown below:

```
package ast;

import java.util.Set;
import java.util.HashSet;
import java.util.Hashtable;

public class Increment extends LCLExpression
{
  private String fVariable;

  public String getVariable() { return fVariable; }

  public Increment( String aVariable )
  {
    fVariable = aVariable;
  }

  public Set<String> freeNames() { return new HashSet<String>(); }

  public LCLExpression reduce( Hashtable<String,LCLExpression> aSymTable )
  {
      return this;
  }

  public LCLExpression substitute( String aVar, LCLExpression aExp )
  {
    if ( getVariable().equals( aVar ) )
    {
      if ( aExp instanceof LambdaNumber )
      {
        Integer lNumber = ((LambdaNumber)aExp).getValue() + 1;
        return new LambdaNumber( lNumber.toString() );
      }
      else
        throw new ArithmeticException( "Illegal argument: " + aExp );
    }
    else
      return this;
  }

  public String toString()
  {
    return "incr(" + fVariable + ")";
  }
}
```

Using this approach, you can implement all built-in functions.

Define the corresponding symbol table, that is, define the symbols `succ`, `pred`, `isZero`, and `notZero` in the `main` method as show below:

```
public static void main( String[] Args )
{
  try
  {
    LCLParser lParser = new LCLParser( new FileInputStream( Args[0] ) );
    ArrayList<LCLExpression> lExpressions = lParser.CompilationUnit();

    Hashtable< String, LCLExpression > lSymbolTable =
                                    new Hashtable< String, LCLExpression >();

    lSymbolTable.put( "succ", new LambdaFunction( "x", new Increment( "x" ) ) );
    lSymbolTable.put( "pred", new LambdaFunction( "x", new Decrement( "x" ) ) );
    lSymbolTable.put( "isZero", new LambdaFunction( "x", new Zero( "x" ) ) );
    lSymbolTable.put( "notZero", new LambdaFunction( "x", new NotZero( "x" ) ) );

    LCLExpression Result = null;

    for ( LCLExpression e : lExpressions )
    {
      Result = e.reduce( lSymbolTable );
    }

    System.out.println( Result );
  }
  catch (ParseException e)
  {
    System.out.println("Syntax Error : \n"+ e.toString());
  }
  catch (FileNotFoundException e)
  {
    System.out.println( e.toString() );
  }
}
```

The new `main` method implements the evaluation of a sequence of lambda terms, but prints only the result of the last evaluation.

Examples located in directory Tests:

fix.lam:

```
(define
   fix
   (lambda f . ((lambda x . (f (lambda y . ((x x) y))))
                (lambda x . (f (lambda y . ((x x) y)))))))
```

plus.lam:

```
(define rplus
 (lambda plus.
  (lambda n .
    (lambda m . (if (notZero n) ((plus (pred n)) (succ m)) m)))))

(load "Tests/fix.lam")

(define plus (fix rplus))
```

test_plus.lam:

```
(load "Tests/plus.lam")

((plus 1) 1)
```

Running the test:

```
% java LCLParser Tests/test_plus.lam
2
%
```

Submission deadline: Monday, October 20, 2014, 10:30.
Submission procedure: on paper.