

**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: \_\_\_\_\_

# Assignment 7

COS30023 - Languages in Software Development

Daniel Parker - 971328X

October 22, 2014

## 1. LCLParser (Problems 1 & 2)

### 1.1. LCLParser.jj

```
options
{
    JDK_VERSION = "1.7";
    static = false;
    OUTPUT_DIRECTORY="parser";
}

PARSER_BEGIN(LCLParser)
package parser;

import java.io.*;
import java.util.*;
import ast.*;

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

            Hashtable< String, LCLEExpression > lSymbolTable =
                new Hashtable< String, LCLEExpression >();
            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" ) ) );
```

```

        LCLEExpression Result = null;

        for ( LCLEExpression 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() );
    }
}
}
PARSER_END(LCLParser)

ArrayList<LCLEExpression> CompilationUnit():
{
    ArrayList<LCLEExpression> Result = new ArrayList<LCLEExpression>();
    LCLEExpression e;
}
{
    (e = LCLEExp() { Result.add( e ); })* < EOF >
    { return Result; }
}

LCLEExpression LCLEExp():
{
    Token t;
    LCLEExpression e1;
    LCLEExpression e2;
    LCLEExpression e3;
}
{
    t = < NUMBER >
    { return new LambdaNumber(t.image); }
|
    t = < VARIABLE >
    { return new LambdaVariable(t.image); }
|
    LOOKAHEAD(2)

```

```

    "(" "define" t = < VARIABLE > e1 = LCLExp() ")"
    { return new LCLDeclaration( t.image, e1 ); }
|
LOOKAHEAD(2)
    "(" "load" t = < STRING > ")"
    { return new LoadDeclaration( t.image.substring(1, t.image.length() - 1 )); }
|
LOOKAHEAD(2)
    "(" "if" e1 = LCLExp() e2 = LCLExp() e3 = LCLExp() ")"
    { return new IfThenElse( e1, e2, e3 ); }
|
LOOKAHEAD(2)
    "(" "lambda" t = < VARIABLE > "." e1 = LCLExp() ")"
    { return new LambdaFunction( t.image, e1 ); }
|
    "(" e1 = LCLExp() e2 = LCLExp() ")"
    { return new LambdaApplication( e1, e2 ); }
}

SKIP :
{
    " "
|
    "\r"
|
    "\t"
|
    "\n"
|
    < "/" " (~["\n"])* "\n">
}

TOKEN :
{
    < NUMBER: (["0"-"9"])+ >
|
    < STRING: "\" (~["\""])* "\"" >
|
    < VARIABLE: ["a" - "z", "A" - "Z"](["a"-"z", "A"-"Z", "0" - "9", "_"])* >
}

```

## 1.2. ast.LCLEExpression

```
package ast;

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

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

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

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

    public abstract String toString();
}
```

## 1.3. ast.LCLDeclaration

```
package ast;

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

public class LCLDeclaration extends LCLEExpression{
    private String fLabel;
    private LCLEExpression fExpression;

    public String getVariable() {
        return fLabel;
    }

    public LCLEExpression getExpression() {
        return fExpression;
    }

    public LCLDeclaration( String aLabel, LCLEExpression aExpression ) {
        fLabel = aLabel;
        fExpression = aExpression;
    }

    @Override
    public Set<String> freeNames() {
```

```

        return fExpression.freeNames();
    }

    @Override
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
        return new LCLDeclaration( fLabel, fExpression.substitute(aVar, aExp));
    }

    @Override
    public String toString() {
        return "(define " + fLabel + " " + fExpression.toString() + ")";
    }

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        LCLEExpression lSubExpression = fExpression.reduce(aSymTable);
        aSymTable.put(fLabel, lSubExpression);
        return lSubExpression;
    }
}

```

#### 1.4. ast.LambdaFunction

```

package ast;

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

public class LambdaFunction extends LCLEExpression {
    private String fVariable;
    private LCLEExpression fBody;

    public String getVariable() {
        return fVariable;
    }

    public LCLEExpression getExpression() {
        return fBody;
    }

    public LambdaFunction( String aVariable, LCLEExpression aBody) {
        fVariable = aVariable;
    }
}

```

```

    fBody = aBody;
}

@Override
public Set<String> freeNames() {
    Set<String> Result = new HashSet<String>(fBody.freeNames());
    Result.remove(getVariable());

    return Result;
}

@Override
public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
    if (getVariable().equals(aVar)) {
        return this;
    } else {
        Set<String> fFrees = aExp.freeNames();

        // Substitution Rule 6
        if ( fFrees.contains( getVariable() ) ) {
            String lFresh = fVariable + "%";
            while ( fFrees.contains( lFresh ) ){
                lFresh += "%";
            }

            LCLEExpression lNewBody = fBody.substitute( fVariable, new LambdaVariable(lFresh));
            lNewBody.substitute(aVar, aExp);
            return new LambdaFunction(lFresh, lNewBody);
        } else {
            // Substitution Rule 5
            return new LambdaFunction( fVariable, fBody.substitute(aVar, aExp));
        }
    }
}

@Override
public String toString() {
    return "(lambda " + fVariable + "." + fBody.toString() + ")";
}

@Override
public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
    return this;
}

```

```
}  
  
}
```

## 1.5. ast.LambdaNumber

```
package ast;  
  
import java.util.Hashtable;  
import java.util.Set;  
import java.util.HashSet;  
  
public class LambdaNumber extends LCLEExpression {  
    private Integer fNumber;  
  
    public Integer getNumber() {  
        return fNumber;  
    }  
  
    public LambdaNumber( String aNumber ) {  
        fNumber = Integer.parseInt( aNumber );  
    }  
  
    @Override  
    public Set<String> freeNames() {  
        return new HashSet<String>();  
    }  
  
    @Override  
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {  
        return this;  
    }  
  
    @Override  
    public String toString() {  
        return fNumber.toString();  
    }  
  
    @Override  
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {  
        return this;  
    }  
}
```



## 1.6. ast.LambdaVariable

```
package ast;

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

public class LambdaVariable extends LCLEExpression{
    private String fValue;

    public LambdaVariable( String aVariable ) {
        fValue = aVariable;
    }

    @Override
    public Set<String> freeNames() {
        HashSet<String> Result = new HashSet<String>();
        Result.add(fValue);
        return Result;
    }

    @Override
    public LCLEExpression substitute( String aVar, LCLEExpression aExp ) {
        if (fValue.equals(aVar)) {
            return aExp;
        } else {
            return this;
        }
    }

    @Override
    public String toString() {
        return fValue;
    }

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        if (aSymTable.containsKey(fValue)) {
            return aSymTable.get(fValue);
        } else {
            throw new RuntimeException("Error reducing " + fValue + ", symbol not found");
        }
    }
}
```

```

    }
}

```

## 1.7. ast.LoadDeclaration

```

package ast;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Set;

import parser.LCLParser;
import parser.ParseException;

public class LoadDeclaration extends LCLEExpression {
    private String fUnitName;

    public String getUnitName() {
        return fUnitName;
    }

    public LoadDeclaration( String aUnitName ) {
        fUnitName = aUnitName;

        if ( !fUnitName.endsWith( ".lam" ) ) {
            fUnitName += ".lam";
        }
    }

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

    @Override
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
        return this;
    }

    @Override

```

```

public String toString() {
    return "(load \"" + fUnitName + "\")";
}

@Override
public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
    try {
        LCLParser lParser = new LCLParser(new FileInputStream(fUnitName));
        ArrayList<LCLEExpression> LCLEExpressions = lParser.CompilationUnit();

        LCLEExpression lastExpression = null;

        for ( LCLEExpression e: LCLEExpressions ) {
            lastExpression = e.reduce(aSymTable);
        }

        return lastExpression;

    } catch (FileNotFoundException e) {
        throw new RuntimeException("The file '" + fUnitName + "' was not found.");
    } catch ( ParseException e ) {
        throw new RuntimeException("Syntax Error : \n"+ e.toString());
    }
}
}

```

## 1.8. ast.LambdaApplication

```

package ast;

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

public class LambdaApplication extends LCLEExpression {
    private LCLEExpression fFunction;
    private LCLEExpression fArgument;

    public LCLEExpression getFunction() {
        return fFunction;
    }

    public LCLEExpression getArgument() {

```

```

    return fArgument;
}

public LambdaApplication( LCLEExpression aFunction, LCLEExpression aArgument) {
    fFunction = aFunction;
    fArgument = aArgument;
}

@Override
public Set<String> freeNames() {
    Set<String> Result = new HashSet<String>(fFunction.freeNames());
    Result.addAll(fArgument.freeNames());

    return Result;
}

@Override
public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
    return new LambdaApplication( fFunction.substitute(aVar, aExp),
        fArgument.substitute(aVar, aExp));
}

@Override
public String toString() {
    return "(" + fFunction.toString() + " " + fArgument.toString() + ")";
}

@Override
public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
    LCLEExpression lArgument = fArgument.reduce(aSymTable);
    LCLEExpression lFunction = fFunction.reduce(aSymTable);

    if (lFunction instanceof LambdaFunction) {
        LambdaFunction lLambdaFunction = (LambdaFunction)lFunction;

        LCLEExpression subExpression = lLambdaFunction.getExpression().substitute(lLambdaFunction);
        return subExpression.reduce(aSymTable);
    } else {
        return new LambdaApplication(lFunction, lArgument);
    }
}
}

```

## 1.9. ast.IfThenElse

```
package ast;

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

public class IfThenElse extends LCLEExpression {

    private LCLEExpression fCondition;
    private LCLEExpression fThen;
    private LCLEExpression fElse;

    public IfThenElse( LCLEExpression aCondition, LCLEExpression aThen, LCLEExpression aElse ){
        fCondition = aCondition;
        fThen = aThen;
        fElse = aElse;
    }

    public LCLEExpression getCondition() {
        return fCondition;
    }

    public LCLEExpression getThen() {
        return fThen;
    }

    public LCLEExpression getElse() {
        return fElse;
    }

    @Override
    public Set<String> freeNames() {
        Set<String> Result = new HashSet<String>( fCondition.freeNames() );
        Result.addAll( fThen.freeNames() );
        Result.addAll( fElse.freeNames() );

        return Result;
    }

    @Override
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
```

```

        return new IfThenElse( fCondition.substitute( aVar, aExp ),
                                fThen.substitute(aVar, aExp),
                                fElse.substitute(aVar, aExp));
    }

    @Override
    public String toString() {
        return "(if " + fCondition.toString() + " " + fThen.toString() + " " + fElse.toString()
    }

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        LambdaNumber lConditionReduced;

        if (!(fCondition.reduce(aSymTable) instanceof LambdaNumber)) {
            throw new RuntimeException("Condition is not an instance of LambdaNumber");
        } else {
            lConditionReduced = (LambdaNumber)fCondition.reduce(aSymTable);
        }

        if (lConditionReduced.getNumber() != 0) {
            return fThen.reduce(aSymTable);
        } else {
            return fElse.reduce(aSymTable);
        }
    }
}

```

## 1.10. ast.Increment

```

package ast;

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

public class Increment extends LCLEExpression {
    private String fVariable;

    public String getVariable() {
        return fVariable;
    }
}

```

```

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

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

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

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

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

```

### 1.11. ast.Decrement

```

package ast;

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

public class Decrement extends LCLEExpression {
    private String fVariable;

    public String getVariable() {
        return fVariable;
    }
}

```

```

    }

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

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

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        return this;
    }

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

    @Override
    public String toString() {
        return "decr(" + fVariable + ")";
    }
}

```

## 1.12. ast.Zero

```

package ast;

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

```



```

public class Zero extends LCLEExpression {
    private String fVariable;

    public String getVariable() {
        return fVariable;
    }

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

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

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        return this;
    }

    @Override
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
        if ( getVariable().equals(aVar) ) {
            if ( aExp instanceof LambdaNumber ) {
                LambdaNumber lNumber = (LambdaNumber)aExp;
                if (lNumber.getNumber() == 0) {
                    return new LambdaNumber("1");
                } else {
                    return new LambdaNumber("0");
                }
            } else {
                throw new ArithmeticException( "Illegal argument: " + aExp );
            }
        } else {
            return this;
        }
    }

    @Override
    public String toString() {
        return "zero(" + fVariable + ")";
    }
}

```

```

    }
}

```

### 1.13. ast.NotZero

```

package ast;

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

public class NotZero extends LCLEExpression {
    private String fVariable;

    public String getVariable() {
        return fVariable;
    }

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

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

    @Override
    public LCLEExpression reduce(Hashtable<String, LCLEExpression> aSymTable) {
        return this;
    }

    @Override
    public LCLEExpression substitute(String aVar, LCLEExpression aExp) {
        if ( getVariable().equals(aVar) ) {
            if ( aExp instanceof LambdaNumber ) {
                LambdaNumber lNumber = (LambdaNumber)aExp;
                if (lNumber.getNumber() == 0) {
                    return new LambdaNumber("0");
                } else {
                    return new LambdaNumber("1");
                }
            }
        }
    }
}

```

```

        } else {
            throw new ArithmeticException( "Illegal argument: " + aExp );
        }
    } else {
        return this;
    }
}

@Override
public String toString() {
    return "notZero(" + fVariable + ")";
}
}

```

## 2. Result

$$((\text{plus } 1) \ 1) = 2$$