# 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:** | 8X, Typed Lambda Calculus |
| **Due date:** | **optional, October 27, 2014, 10:30, on paper** |
| **Lecturer:** | Dr. Markus Lumpe |

---

**Your name:**_____

---

Marker's comments:

| Problem | Marks | Obtained |
|---|---|---|
| 1 | 58 | |
| Total | 58 | |

---

**Extension certification:**

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

Signature of Convener:_____

# Assignment 8X

**COS30023 - Languages in Software Development**

Daniel Parker - 971328X

October 26, 2014

## 1. TypedLambda

### 1.1. TypedLambda.jj

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

PARSER_BEGIN(TypedLambda)
package parser;

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

public class TypedLambda {
  public static void main(String Args[]) throws ParseException {
    try {
      TypedLambda lp = new TypedLambda( new FileInputStream( Args[0] ) );
      ArrayList< TypedLambdaExpression> exprs = lp.CompilationUnit();

      for ( TypedLambdaExpression exp : exprs ) {
        try {
          System.out.println( "Checking: " + exp );
          LambdaType type = exp.typeCheck( new Hashtable<String,LambdaType>() );
          System.out.println( "SUCCESS: " + exp + " has type " + type );
        } catch (RuntimeException e) {
```

```
          System.out.println( "Oops, type error encountered: " + e.getMessage() );
        }
      }
    } catch ( ParseException e ) {
      System.out.println( "Syntax Error : \n" + e.toString() );
    } catch ( FileNotFoundException e ) {
      System.out.println( e.toString() );
    } catch ( RuntimeException e ) {
      System.out.println( "Oops, type error encountered: " + e.getMessage() );
    }
  }
}
PARSER_END(TypedLambda)

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

ArrayList<TypedLambdaExpression> CompilationUnit():
{
  TypedLambdaExpression e;
  ArrayList<TypedLambdaExpression> Results = new ArrayList<TypedLambdaExpression>();
}
{
  ( e = LambdaExp() { Results.add(e); } ) + < EOF >
  { return Results; }
}

TypedLambdaExpression LambdaExp():
{
  TypedLambdaExpression e1;
  TypedLambdaExpression e2;
  LambdaType lType;
  Token t;
```

```
}
{
  t = < NUMBER >
  { return new LambdaNumber( t.image ); }
|
  t = < VARIABLE >
  { return new LambdaVariable( t.image ); }
|
  LOOKAHEAD(2)
  "(" "lambda" t = < VARIABLE > lType = Type() "." e1 = LambdaExp() ")"
  { return new LambdaFunction( t.image, lType, e1 ); }
|
  "(" e1 = LambdaExp() e2 = LambdaExp() ")"
  { return new LambdaApplication( e1, e2 ); }
}

LambdaType Type():
{
  Token s;
  LambdaType t1;
  LambdaType t2;
}
{
  s = "Int"
  { return new IntegerType(); }
|
  "(" t1 = Type() "->" t2 = Type() ")"
  { return new FunctionType( t1, t2 ); }
}

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

## 1.2. ast.TypedLambdaExpression

```
package ast;
```

```
import java.util.Hashtable;

public abstract class TypedLambdaExpression {
  public abstract LambdaType typeCheck( Hashtable<String, LambdaType> aGamma );
  public abstract String toString();
}
```

### 1.3. ast.LambdaType

```
package ast;

public abstract class LambdaType {
  public abstract boolean match( LambdaType aOtherType );
  public abstract String toString();
}
```

### 1.4. ast.IntegerType

```
package ast;

public class IntegerType extends LambdaType {

  public IntegerType() { }

  @Override
  public boolean match(LambdaType aOtherType) {
    if (aOtherType instanceof IntegerType) {
      return true;
    } else {
      return false;
    }
  }

  @Override
  public String toString() {
    return "Int";
  }
}
```

### 1.5. ast.FunctionType

```
package ast;

public class FunctionType extends LambdaType {

  private LambdaType fType1;
  private LambdaType fType2;

  public LambdaType getType1() {
    return fType1;
  }

  public LambdaType getType2() {
    return fType2;
  }

  public FunctionType( LambdaType aType1, LambdaType aType2) {
    fType1 = aType1;
    fType2 = aType2;
  }

  @Override
  public boolean match(LambdaType aOtherType) {
    if (aOtherType instanceof FunctionType) {
      if (((FunctionType) aOtherType).getType1().match(fType1) &&
        ((FunctionType)aOtherType).getType2().match(fType2)) {
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }

  @Override
  public String toString() {
    return "(" + fType1.toString() + " -> " + fType2.toString() + ")";
  }
}
```

### 1.6. ast.LambdaNumber

```
package ast;

import java.util.Hashtable;

public class LambdaNumber extends TypedLambdaExpression{
  private String fValue;

  public LambdaNumber( String aValue ) {
    fValue = aValue;
  }

  @Override
  public LambdaType typeCheck(Hashtable<String, LambdaType> aGamma) {
    return new IntegerType();
  }

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

### 1.7. ast.LambdaVariable

```
package ast;

import java.util.Hashtable;

public class LambdaVariable extends TypedLambdaExpression{
  private String fVariable;

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

  @Override
  public LambdaType typeCheck(Hashtable<String, LambdaType> aGamma) {
    if (aGamma.containsKey(fVariable)) {
      return aGamma.get(fVariable);
    } else {
      throw new RuntimeException(fVariable + " is not contained in the type environment");
```

```
    }
  }

  @Override
  public String toString() {
    return fVariable;
  }
}
```

### 1.8. ast.LambdaFunction

```
package ast;

import java.util.Hashtable;

public class LambdaFunction extends TypedLambdaExpression {
  private String fVariable;
  private LambdaType fType;
  private TypedLambdaExpression fBody;

  public LambdaFunction( String aVariable, LambdaType aType,
                  TypedLambdaExpression aBody ) {
    fVariable = aVariable;
    fType = aType;
    fBody = aBody;
  }

  @Override
  public LambdaType typeCheck(Hashtable<String, LambdaType> aGamma) {
    Hashtable<String, LambdaType>lGamma = (Hashtable<String, LambdaType>) aGamma.clone();
    lGamma.put(fVariable, fType);
    return new FunctionType( fType, fBody.typeCheck(lGamma));
  }

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

### 1.9. ast.LambdaApplication

```
package ast;

import java.util.Hashtable;

public class LambdaApplication extends TypedLambdaExpression{

  private TypedLambdaExpression fFunction;
  private TypedLambdaExpression fArgument;

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

  @Override
  public LambdaType typeCheck(Hashtable<String, LambdaType> aGamma) {
    LambdaType lArgumentType = fArgument.typeCheck(aGamma);

    LambdaType lFunction = fFunction.typeCheck(aGamma);

    if (lFunction instanceof FunctionType) {
      if (((FunctionType)lFunction).getType1().match(lArgumentType)){
        return ((FunctionType)lFunction).getType2();
      } else {
        throw new RuntimeException(fFunction.toString() +
            " parameter type does not match argument type " + fArgument );
      }
    } else {
      throw new RuntimeException("Function type expected for '" + fFunction + "'");
    }
  }

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

## 2. Results

### 2.1. error.lam

```
Checking: (lambda x (Int -> Int) .
            (lambda y (Int -> Int) . (lambda z Int . ((x z) (y z)))))
Oops, type error encountered: Function type expected for `(x z)'
```

### 2.2. one_plus.lam

```
Checking: (((lambda m ((Int -> Int) -> (Int -> Int)) .
              (lambda n ((Int -> Int) -> (Int -> Int)) .
                (lambda s (Int -> Int) . (lambda z Int .
                  ((m s) ((n s) z))))))
                ((lambda n ((Int -> Int) -> (Int -> Int)) .
                  (lambda s (Int -> Int) . (lambda z Int . (s ((n s) z)))))
                (lambda s (Int -> Int) . (lambda z Int . z))))
                  ((lambda n ((Int -> Int) -> (Int -> Int)) .
                    (lambda s (Int -> Int) . (lambda z Int . (s ((n s) z)))))
                  (lambda s (Int -> Int) . (lambda z Int . z))))
SUCCESS: (((lambda m ((Int -> Int) -> (Int -> Int)) .
              (lambda n ((Int -> Int) -> (Int -> Int)) .
                (lambda s (Int -> Int) . (lambda z Int .
                  ((m s) ((n s) z))))))
                ((lambda n ((Int -> Int) -> (Int -> Int)) .
                  (lambda s (Int -> Int) . (lambda z Int . (s ((n s) z)))))
                (lambda s (Int -> Int) . (lambda z Int . z))))
                  ((lambda n ((Int -> Int) -> (Int -> Int)) .
                    (lambda s (Int -> Int) . (lambda z Int . (s ((n s) z)))))
                  (lambda s (Int -> Int) . (lambda z Int . z))))
has type ((Int -> Int) -> (Int -> Int))
```