An Interactive Interpreter for Expressions in the Lambda Calculus

Kim Mason

Department of Computer Science

THE UNIVERSITY OF ADELAIDE

Submitted in partial fulfilment of the
requirements of the Honours degree in
Computer Science

November 1997

Contents

Acknowledgments

Many thanks to my honours supervisor, Mr Brad Alexander, for invaluable advice and guidance, to the staff and honours students in the Adelaide University Computer Science department for their help and comradeship, and to my friends and family for support.

An Interactive Interpreter for Expressions in the Lambda Calculus

ABSTRACT

This paper introduces an interactive interpreter for the lambda calculus. The lambda calculus is a formalism that provides a theoretical foundation for functional programming. Many students currently have difficulty understanding the lambda calculus, possibly due to trivial mistakes made while reducing lambda expressions using the lambda calculus reduction rules, and also possibly due to a lack of interactive feedback and explanation while working through examples and exercises.

The learning tool is designed to give students an easily useable interpreter for the lambda calculus with which they can evaluate any expressions they wish. The tool displays intermediate results from lambda reductions, and offers an explanation to the student if required. The motivation to produce a lambda calculus teaching tool is discussed, as well as the functional requirements of such a tool.

An implementation of the tool in Java 1.1 is presented, with implementation details, theoretical and practical support for design decisions, and possible future modifications which could be made to improve the tool.

1. Introduction

The purpose of this paper is to introduce the development of an educational tool for interpreting and reducing expressions in the lambda calculus. One of the main goals of the project was that the tool must enable students to gain a deeper understanding of the lambda calculus and its reduction rules in a shorter period of time, and with less effort than current learning methods allow.

The lambda calculus is a simple formalism that can be used to represent functions and applications of functions to arguments. The lambda calculus provides a theoretical foundation for functional programming. Computation takes place in this calculus through a process of reducing lambda expressions to progressively less complex forms. Both the number of constructs and the number of rules to reduce expressions in the lambda calculus is extremely small. This makes it a good medium for proving mathematical properties and exploring the fundamental concepts of functional computational processes.

A strong mapping exists between many aspects of functional programming and the lambda calculus. This means that properties that are shown to hold for the lambda calculus can be shown to hold for many aspects of functional programming. This mapping also means that the λ calculus is often used as a kind of 'functional machine code' when compiling or interpreting functional languages. All functional constructs and applications can be represented in pure lambda calculus, including numbers, arithmetic operators, logical operators, recursion, and others.

The lambda calculus (and its variants) are widely used both during the development of functional languages, and in the compilation process. An understanding of the lambda calculus provides an important base for understanding the inner workings and underlying principles of functional programming.

It is felt that some type of interactive interpreter is required for a learning aid because of the difficulties that students display in learning the lambda calculus. It is not that the lambda calculus is particularly difficult or complex, but that it is conceptually unfamiliar to new students of functional programming.

There are three main concepts to grasp while learning the lambda calculus:

1. How functional languages can map to the lambda calculus.
2. How this mapping aids the development and compilation of functional programming languages.
3. How the lambda calculus can be reduced using its reduction rules.

It is not the aim of this teaching tool to enhance a student's understanding of how lambda calculus is related to functional programming (point 2). This is a conceptual area that requires an understanding of functional programming and the development of functional languages, as well as an understanding of the lambda calculus. The area in which an automated tool can offer a great deal of help is in aiding students in the understanding of the process of reducing lambda expressions using the lambda calculus reduction rules, and in understanding the mapping of the lambda calculus to functional constructs.

2. Introduction to the Lambda Calculus

The pure lambda calculus (λ calculus) is a formal system for representing functions (abstractions), and functional applications. It allows naming of values, representation of functions, application of functions to values, and applications of functions to functions (higher order functions). The lambda calculus was invented by Alonzo Church and presented in 1941. The λ calculus is a good low-level representation of functional constructs because:

1. It is a very simple language, with few syntactic constructs, and simple semantics. Because of these properties, the λ calculus is very good for reasoning about the correctness of functional programming.

2. The λ calculus is highly expressive. It has been shown that any computable function can be represented in the λ calculus[1].

3. Many functional constructs in functional languages map strongly onto the λ calculus. This means that the λ calculus is useful for examining the properties of functional languages.

2.1 The Syntax of the λ Calculus

The λ calculus has a very simple syntax, as given below in BNF:

<exp> ::= <constant> Built-in constants

[1] Page 9 of [Jones87]

| <variable> Variable names

| (<exp>)

| <application> | <abstraction>

<application> ::= <exp> <exp> Functional application

<abstraction> ::= λ<variable>.<exp> λ abstraction

2.1.1 Lambda Abstractions

The two main constructs in the above definition are the λ abstraction, and functional application. A λ abstraction can be thought of as a newly defined function, where the variable before the period can be seen as the formal parameter. The <variable> part of the abstraction is called the bound variable, and the <exp> part of the abstraction is called the body. The body of the λ abstraction is what is returned by the function. A λ abstraction can be thought of like this:

```
(λ        x   .   ( +    x    1 ) )

↑         ↑   ↑     ↑   ↑    ↑
```

That function of x which adds x to 1

Note that λ abstractions often require brackets to show when the abstraction ends. It is possible to remove some of these brackets if conventions about function precedence are established. These conventions are explained in section 2.1.3 (λ applications and currying).

2.1.2 Bound and Free Variables

A variable is bound if it is contained within a λ abstraction (or multiple nested λ abstractions), and one of the containing abstractions “binds” that

variable (has the variable after the ‘λ’ and before the ‘.’ in an abstraction). For example, consider the λ expression:

(λ x . (+ x y))

In order to evaluate this expression completely, we require all values within this expression. We don’t have a problem with the variable x, because it is ‘bound’ in a λ expression (x is like a formal parameter). However, the variable y is unbound. In this case, y is like an undefined global variable within a conventional function.

This problem can be overcome if y is bound as well. Binding multiple variables is achieved by nesting λ abstractions:

(λ y. (λ x . (+ x y)))

In the above, both x and y are now bound, so the above expression can be evaluated. A precise definition of bound and free variables follows[2]:

[2] This definition occurs in Figure 2.2, [Jones87]

An occurrence of a variable must be either free or bound.

Definition of 'occurs free'

x occurs free in x (but not in any other variable or constant)

x occurs free in (E F) ⇔ x occurs free in E

or x occurs free in F

x occurs free in λy.E ⇔ x and y are different variables

and x occurs free in E

Definition of 'occurs bound'

x occurs bound in (E F) ⇔ x occurs bound in E

or x occurs bound in F

x occurs bound in λy.E ⇔ (x and y are the same variable

and x occurs free in E)

or x occurs bound in E

Note: '⇔' means 'if and only if'

Definition of free and bound variables

2.1.3 Lambda Applications and Currying

A λ application is simply a λ function applied to one or more arguments. Functional application is simply represented using

juxtaposition. For instance, the expression (λ x . x) applied to the argument 3 is simply:

(λ x . x) 3

To represent an application with multiple arguments, the functions and arguments are juxtaposed with left associativity. This device allows us to think of all functions as only having one argument, and is called currying. For instance, a λ abstraction applied to 3 arguments can be written as:

((((λx.(λy.(λz.(<expr>)))) (3)) (2)) (1))

This can be seen as the function (λx.(λy.(λz.(<expr>)))) applied to the argument 3. The result of this application is then applied to 2, and the result of that application is then applied to 1. Note that in the λ calculus, functions can return other functions.

If left associativity is assumed, some of the brackets can be dropped, and the application re-written as:

(λx.(λy.(λz.(<expr>)))) 3 2 1

When the body of a λ abstraction is not an application, brackets surrounding the body can also be dropped, as it is not ambiguous where the body of the abstraction ends. The above can then be re-written as:

(λx.λy.λz.<expr>) 3 2 1

This is often shortened to (λx y z.<expr>) 3 2 1

2.2 The Operational Semantics of the Lambda Calculus

The syntax of the λ calculus has been described, but now we need to see how to ‘calculate’ with it. This section will introduce the reduction rules required to ‘reduce’ a λ expression.

2.2.1 Beta Reduction

Now that we have described how to represent functional applications within the λ calculus, we will now describe how to substitute the argument into the ‘formal parameter’ of the function. This action is called β-reduction, because it ‘reduces’ the λ expression. Only functional applications can be β reduced. The rule for β reduction is simple:

The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which free occurrences of the formal parameter in the body are replaced with copies of the argument.

For example, the λ application:

(λ x . + x x) 6

reduces to:

(+ 6 6)

Note that for β reduction to occur, the expression being reduced must be an application, and the left sub-expression of that application must be a λ abstraction, or a built-in function.

2.2.2 Eta Reduction

Consider the λ expressions:

(λ x . + 3 x)

and

(+ 3)

When applied to any argument, these two expressions behave in exactly the same way. This equivalence can be expressed by using η (Eta) reduction. The rule for η reduction is:

An expression of the form (λ x . M x) can be reduced to M, provided that x does not occur free in M and that M is a λ abstraction (or built-in function).

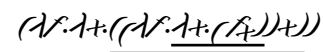
The condition that M denotes an abstraction or built-in function is necessary to prevent false conversions involving built-in constants. Only λ abstractions in the form given above can be η-reduced.

2.2.3 Alpha Conversion

Alpha conversion is a rule that allows us to rename the bound variable (formal parameter) of a λ abstraction, as long as we do so consistently. The newly introduced name must not occur free in the body of the original λ abstraction. Alpha conversion is required to convert equivalent abstractions into identical representations, and to overcome the problem of name capture, the description of which follows. Note that α conversion isn't a reduction, in the sense that it doesn't reduce the expression being operated on to a less complex form. Alpha reduction is only necessary to convert between equivalent representations of expressions.

2.2.4 The Name Capture Problem

There is one situation where β reductions aren't as trivial as the process described above. Consider the λ application:



This λ expression is valid, and all variables are bound. Now consider an attempt to β reduce the underlined (long underline) sub-expression:

$$(\lambda f.\lambda x.(f\ x))\ x$$

We will replace the formal parameter f with the actual parameter x. However, x is already used as another formal parameter in the body (short underline). It is wrong to reduce this inner body to:

$$(\lambda x.\ x\ x)$$

because the x substituted for the f would be 'captured' by the inner λx abstraction. This can be overcome by first α converting the λx abstraction, so that is no longer clashes with the actual parameter.

A general solution to this problem is to rename any abstractions that clash with any free variables being substituted into the current expression, and then perform the substitution on the renamed abstraction.

The name capture problem illustrates that β conversion isn't as simple as first indicated. A precise definition of the λ reduction rules is necessary to implement a λ calculus interpreter.

2.3 A Precise Definition of the λ Reduction Rules

As can be seen from the name capture problem, a precise definition of the α, β and η reduction rules are required to implement an interpreter. This requires an extra piece of notation. The notation:

E[M/x]

represents the expression E with a copy of M substituted for all free occurrences of x. A precise definition of E[M/x] follows:

x[M/x]	=	M
c[M/x]	where c is any variable or constant other than x	
	=	c
(E F)[M/x]	=	(E[M/x] F[M/x])
(λx.E)[M/x]	=	λx.E
(λy.E)[M/x]	where y is any variable other than x	
	=	λy.E[M/x] if x does not occur free in E or if y does not occur free in M
	=	λz.(E[z/y])[M/x] otherwise, where z is a new variable name which does not occur

Definition of E[M/x]

This definition will remove any name-clashes while renaming variables or substituting expressions for variables in λ expressions. We can now define β reduction as:

$$(\lambda x.E) \Leftrightarrow E[M/x]$$

This definition of β conversion delivers consistent renaming, and removes all clashes. The definition of E[M/x] is also useful for α conversion.

2.4 Reduction Orders

When reducing any λ expression, an arbitrary reducible sub-expression can be chosen from the expression for reduction. When a λ expression contains no more reducible expressions, it is said to be in ***normal form***. This gives rise to several different generalised reduction schemes, or reduction orders.

It is impossible for two different reduction sequences to lead to different normal forms. This is stated in the Church-Rosser Theorem 1 as:

> If E1 $\Leftrightarrow$ E2, then there exists an expression E, such that
>
> E1 $\Leftrightarrow$ E and E2 $\Leftrightarrow$ E.

The Church-Rosser Theorem 1

It should be noted that the above theorem does not guarantee reduction termination. It only guarantees that if a reduction does terminate for a given expression, the final expression will always be the same.

2.4.1 Normal Reduction Order

The simplest reduction order is called normal reduction order. To normal reduce a λ expression, the leftmost outermost reducible expression is chosen, and then reduced. A reducible expression is any expression that can be β reduced or η reduced. To completely reduce a λ expression, this reduction sequence is repeated until the expression contains no reducible sub-expressions. The expression will then be in normal form. Normal order guarantees a reduction sequence to normal form, if a normal form exists. This condition is stated in the Church-Rosser Theorem 2.

2.4.2 Applicative Reduction Order

Another simple reduction order is applicative reduction order. All functions in the λ calculus can be viewed as having only one argument due to currying. Applicative reduction order attempts to reduce any outermost reducible expression first, with the exception of functional applications. The argument of a functional application is applicatively reduced, followed by the β reduction of the application itself. Applicative order reduction can cause non-termination problems in cases where normal reduction order would not, due to the fact that it infinitely unrolls recursive constructs[3]. Applicative order reduction can also cause name capture problems in expressions that have no globally free variables. These expressions would completely reduce under normal reduction, without the name capture problem.

2.5 Lambda Syntax Used in the Teaching Tool

As mentioned earlier in this section, it is possible to supply built-in constants in the λ calculus. These constants can represent functions or values, and the functions can be higher order functions (able to return a function). These constants are simply syntactic constructs that can be represented in the λ calculus. The addition of extra built-in constants for the tool was considered, but all built-in constants except natural numbers, +, -, * and / were omitted.

---

[3] For an explanation of combinators and recursive lambda expressions, the reader is referred to [Michaelson89]. Examples can be found in Appendix A.

2.5.1 The Advantages of Built-in Constants

A large advantage that built-in constants have in the λ calculus is their readability. For example, the representations of natural numbers in the λ calculus are very long, and difficult to read[4]. Using built-in numbers simplifies the reading of λ expressions by humans. Using built-in constants can also simplify the representation of other functions. Functions can be written in the λ calculus to convert λ representations of a construct into the built-in representation. Some constants considered for inclusion were if-statements, numerical operators, numerical comparisons, booleans, boolean operators, and boolean comparisons.

A second advantage of built-in constants is that if those constants are supported by the machine (for instance, natural numbers and numerical operations), the reduction of expressions containing these constants can be greatly sped up. Expressions that contain the λ equivalents of these constants can be converted to the built-in constants by using conversion functions (which can be written in the λ calculus).

2.5.2 The Disadvantages of Built-in Constants

A reason for omitting built-in constants and functions is that a goal of the tool is to teach students the pure λ calculus, with no built-in constants, and to demonstrate how reductions are performed in the pure λ calculus. Built-in constants wouldn't aid this understanding. Another goal of the tool is to teach the mapping between the λ calculus and functional constructs. If built-in constants are supplied, students may use the built-in constants

---

[4] See Appendix A for examples of natural numbers in the λ calculus.

without understanding how they can be represented in the λ calculus. This could make the tool less effective in teaching the mapping between λ expressions and functional constructs.

A third reason to omit built-in functions and constants is the number of functions and constants required to support constructs such as the if-statement. If an if-statement is built-in, and there are any other built-in constants, the if-statement must be written to support a given definition of ***true*** and ***false*** in the λ calculus (there are many possible definitions) in the conditional part of the if-statement, or be written to support a built-in boolean representation. For any other built-in constants, comparison operators are required to convert the conditional part of the if-statement to the correct λ expression. For example, consider the expression:

```
λx.λy.( if (> (x) (y))    then * (x) (x)
                          else * (y) (y) )
```

The built-in function '>' must return a boolean value. This must either be a built-in boolean, or a lambda calculus representation of ***true*** and ***false***. If the if-construct supports only built-in booleans, this limits the use of the λ expressions true and false in the if-statement (unless a boolean conversion function is used to convert λ booleans to built-in booleans). If the if-construct supports only λ calculus booleans, it is restricted in the boolean representations that it can recognise.

Due to the number of comparison operators required, as well as the possible confusion created for the student over which representation of booleans the if-construct can handle, the if-construct and booleans were omitted.

2.5.3 Syntax Used by the λ Teaching Tool

The previous 2 paragraphs introduced the constructs that the λ tool can use. The syntax used by the tool is given below in extended[5] BNF:

<binding> ::= let <variable> = <expression>

<variable> :== <letter> [<letter> | <number>]*

<expression> ::= <variable> | <number > |

<abstraction> |

<application> |

<numerical operator> <expression> <expression>

<numerical operator> ::= + | - | * | /

<application> = <expression> <expression>

<abstraction> = \ [variable]+ . <expression>

A <binding> represents the binding of a name to a λ expression. This feature is used so that named λ expressions can be referred to in later λ expressions. Note that numerical operators are not curried in the above definition (they require 2 arguments). They can be easily curried by placing them inside a λ abstraction – eg (λx.λy.+ x y). Another feature of the above syntax is that λ abstractions can be written as:

---

[5] [ ]+ denotes “one or more occurrences of”

[ ]* denotes “zero or more occurrences of”

λ<var1><var2><var3>.<expression>

This representation is equivalent to:

λ<var1>.λ<var2>.λ<var3>.<expression>

3. Project Background, Motivation and Goals

As stated in the introduction, the aim of this project was to create a tool to aid a student's understanding of the syntax, semantics, and reduction rules of the λ calculus, as well as the mapping between the λ calculus and functional constructs. The tool must assist students in learning the λ calculus faster and more easily than is currently achieved with existing methods. This section will go on to outline the motivation for producing a learning tool, the current learning methods and their deficiencies, and ways that the λ teaching tool will address the deficiencies within these methods. A theoretical background of interactive learning will also be introduced.

3.1 Why Create a Learning Tool?

As explained in the introduction, the λ calculus provides an important theoretical basis for understanding the development and compilation of functional languages, and it provides a useful mathematical tool for reasoning about functional languages. Due to the mapping between the λ calculus and functional constructs, and the simplicity of the λ calculus, the λ calculus (or a modified form of it) is often used as a 'functional machine code' for compilation of functional languages.

An understanding of the λ calculus is important to a general understanding of functional programming, and vital to an understanding of the development and compilation of functional languages. There are three main learning areas associated with the λ calculus. These are:

1. Learning how the λ calculus maps to functional programming.

2. Learning how this mapping aids the development and compilation of functional programming languages.

3. Learning the process of reducing the λ calculus using the reduction rules. This requires an understanding of the λ calculus reduction rules, and the order in which they are applied.

The primary aim of this teaching tool is to teach the process of reducing expressions using the λ calculus' reduction rules, and also to aid an understanding of the mapping of the λ calculus to functional constructs. The aim is not to teach how λ calculus aids the development and compilation of functional languages (point 2). To understand this, it is necessary to already have an understanding of the syntax, semantics, and reduction rules of the λ calculus, as well as an understanding of functional languages, and their mapping to the λ calculus.

3.1.1 Knowledge Areas the Tool Aims to Address

McGill & Volet produced a conceptual framework for analysing a student's knowledge of various elements of a programming language[6]:

	Declarative Knowledge	Procedural Knowledge
Syntactic Knowledge	Knowledge of syntactic facts related to a particular language.	Ability to apply rules of syntax when programming.
Conceptual Knowledge	Understanding of and ability to explain the semantics of the actions that take place	Ability to design solutions to programming problems

[6] Refer to [Volet3].

	as a program executes	
Strategic/Conditional Knowledge – the ability to design, code, and test a program to solve a novel problem.		

This tool aims to boost the learning ease of conceptual declarative knowledge of the λ calculus (understanding of the semantics of the λ calculus, and its reduction rules). Most undergraduate courses don't require conceptual procedural knowledge (being able to produce solutions to simple problems) of λ calculus, however the tool has the possibility of boosting this area of knowledge because it is an interpreter, and reasonably complex programs can be written for it. An understanding of the mapping between λ calculus and higher-level functional constructs should provide conceptual procedural knowledge of the λ calculus, because higher-level constructs can be written in the λ calculus, and used to solve simple programming problems.

3.2 Why Students Encounter Difficulty Learning the λ Calculus

Students encounter difficulty understanding the process of understanding and reducing λ expressions, as well as understanding the mapping between the λ calculus and functional constructs. There are several reasons for this:

1. Students have difficulty in understanding the applicability of the λ calculus, and so lose motivation. This is linked to understanding how λ calculus is related to functional programming, and is perhaps better left to explanation, or another non-process based teaching method. However, an understanding of the mapping between the λ calculus and functional constructs can aid understanding of the applicability of the λ calculus, and the

understanding of this mapping can be aided by the λ teaching tool.

2. The λ calculus is unfamiliar and low-level, and its reduction rules are difficult to apply. This causes students to experience difficulty when trying to learn the λ calculus semantics and reduction rules. Students often become lost in the textual complexity of λ expressions. Because the λ calculus is low-level, students also have problems seeing the connection between the λ calculus and higher-level functional constructs.

We will now examine the current learning methods, and outline how these problems are not well addressed with current learning methods. Methods for overcoming these problems will then be proposed.

3.3 Problems with Current Teaching Methods

The current learning methods used to teach student the reduction processes of λ calculus mainly involve examples and worked exercises. This section will outline the shortcomings of both of these approaches, and introduce some tool features that can be used to overcome these problems.

The drawbacks of examples are:

1. The student may not understand every step, possibly due to poor explanation.

2. The student might not be able to find an example of a specific reduction that they don't understand.

3. The student may be confused by the textual complexity of the λ calculus.

A way to overcome the first 2 limitations is to allow the tool to completely drive the reduction of any expression that the student enters, with an explanatory facility offered at every step. The third limitation can be overcome by allowing naming of λ expressions, thus bridging some of the semantic gap.

The drawbacks of worked exercises are:

1. The student may become ‘stuck’ at one point in an exercise, and require assistance and an explanation.

2. The student may make trivial mistakes during reductions that are better automated.

3. The student may be confused by the textual complexity of the λ calculus. This is strongly connected to point #1, as they both stem from λ expressions being low-level, and difficult to read and understand.

Problem 1 can be overcome by allowing the student to drive reductions (choose the next sub-expression to be reduced), with the tool providing assistance if the student can make no further progress with the reduction scheme they are currently using. The tool must explain the criteria it uses to select the next sub-expression for reduction.

Problem 2 can be overcome by having the tool completely automate the reduction of a sub-expression after the sub-expression has been chosen. The tool can explain the reduction to the student, why that particular reduction was chosen, and give the result of the reduction.

Problem 3 can be overcome by allowing the naming of λ expressions, in the same way as problem 3 in the drawbacks of examples.

Examples are also used to teach the mapping between the λ calculus and functional constructs. Some of the limitations of these type of examples are:

1. They aren't very dynamic. Higher-level constructs in the λ calculus may be presented, but it is difficult to demonstrate to the student that these constructs can be reduced using the λ calculus' reduction rules.

2. The student has no way of building small programs from λ calculus constructs, and then reducing them using the λ calculus rules.

These problems can be overcome by allowing the λ reduction tool to have a library of named λ functions available, and then new expressions can be built from the named ones. The teacher or student using the tool can then reduce the new expressions. One important property that the tool must display is the preservation of named expressions for as long as possible. The tool must be able to preserve a named expression until the expression is modified. Only then is the name discarded.

3.4 Three Ways to use the Tool

This section will outline the three main approaches to using the tool. These are teaching the reduction process, teaching the mapping between the λ calculus and functional constructs interactively, and using the tool as a demonstration device. Explanations of these follow.

3.4.1 Process Based Teaching of the Reduction Process[7]

As previously stated, a goal of the λ tool is to teach the ***process*** of reducing λ expressions. Before using the tool, students need an introduction to λ calculus, and need be familiar with at least the syntax of the λ calculus (under Volet's scheme, they require at least syntactic-declarative knowledge). When a student understands the process of reducing λ expressions, they will also understand the syntax and semantics of the pure λ calculus. An interactive tool should be well suited to teaching processes to students, due to their ability to demonstrate the process being learned, and to aid the student at any point in the process.

3.4.2 The Mapping Between the λ Calculus and Functional Languages

Due to the simplicity of the mapping between λ calculus and many functional languages, it is possible for the tool to also aid understanding of the mapping between the λ calculus and functional languages by allowing the naming of λ expressions. This is due to the fact that many functional constructs can be created from lambda expressions using 'syntactic sugar', or simple expression renaming and syntax modification. When the tool provides naming of λ expressions, complex functions can be built up, and the λ calculus begins to look quite similar to a functional language. An understanding of the semantics and reduction rules of λ calculus in association with an understanding of the mapping between the λ calculus

---

[7] Note that this does not refer to process based teaching as it has meaning in the field of psychology, but does refer to the teaching of a defined process or sequence of actions.

and functional languages promotes an understanding of the underlying functional theory of functional languages.

3.4.3 The Usefulness of the Tool as a Demonstration Device

The tool can also be very useful for a lecturer as a demonstration device. If the tool allows naming of lambda expressions, and allows functions to be built up in those expressions, the tool can be used to demonstrate complex reduction sequences. The demonstrator will be able to drive the reduction process, or the tool can drive all reductions. This will aid the students in understanding how to build up complex functional constructs from the lambda calculus. This will be examined later when discussing the functional requirements of the tool.

3.5 Previous Research Into Interactive Learning

There has been a large amount of research into interactive learning using computers. However, most of the research has been based on presenting the student with a course that is presented using hypertext or some other method of presenting information to the student in an easily accessible computerised form. Much of this research has been inconclusive, and the question of how effective interactive teaching methods are is still open.

Russell produced a paper entitled “The No Significant Difference Phenomenon” in which he asserts that there is no significant difference between the success of interactive and conventional teaching methods. He cites 248 supporting case studies. Conversely, many case studies can be found which support interactive learning, and many more are inconclusive. Most papers currently available focus on the learning of static or declarative concepts. The conclusions drawn from these papers may not be applicable to the learning of a process.

The question of whether tools that aid the learning of a process are effective is still open. More research is required using different tools to

teach different fields to gauge the possible effectiveness of tools as process learning aids.

Intuition would suggest that when learning the reduction process, an interactive tool may not be better than a good set of examples, even though it is possible to demonstrate examples using the interpreter. This is because the learning of examples is aided by the type of examples chosen, and the explanations provided with the examples. A good set of written examples can have a very complete set of explanations.

A tool could be far superior to worked exercises, due to the textual complexity of the lambda calculus. A tool should also be ideal for demonstrating the mapping of the lambda calculus to functional languages, due to its ability to name lambda expressions, use those names in the definition of new lambda expressions, and reduce the complex expressions step by step. This is currently difficult to demonstrate to students, due to the complexity of reducing large named λ expressions.

3.6 Summary of Tool Goals

The tool aims to teach:

1. The process of reducing lambda expressions using the λ calculus reduction rules.

2. The mapping from the λ calculus to functional language constructs.

To achieve this, the tool will provide automated expression reduction, an explanatory facility at every step, and the naming of λ expressions. The specifics of the features included in the tool will be described in section 4.

4 Features and Overview of the λ Teaching Tool

This chapter introduces a summary of the features and usage of the λ calculus teaching tool. A list of features required to teach the λ calculus will first be introduced, followed by an introduction to the features of the tool. An outline of how the tool will be used is then presented.

4.1 A List of Features

This section introduces a list of features that can be used to teach the three main learning areas addressed by the tool, and how these features can be used. These learning areas are:

1. Teaching the process of λ reduction.
2. Teaching the mapping between λ expressions and functional constructs.
3. Using the λ tool as a demonstration device for both the process of λ reduction and the mapping between λ expressions and functional constructs.

Many of the features are shared between the three learning areas.

4.1.1 Features for Teaching the Process of λ Reduction

An approach that can be used to teach λ reduction to students is to allow either the tool, or the student to drive reduction of a λ expression. This allows the tool to act as both a provider of examples, and an automation tool for exercises in the λ calculus. An explanation of features that can be used for this follows.

Reading Input Files and Expressions

The first thing that the tool must allow is the reading in of files with pre-declared λ expressions. This is so that example λ expressions can be provided to teach specific reductions. The naming of λ expressions is necessary within this file so that the student may refer to the named expressions, and also to hide unnecessary complexity from the student.

For example, expressions can be declared:

Let true = \x.\y.x

Let identity = \x.x

Students can then refer to named expressions within other expressions (we say the expressions name is ***bound*** to the named expression). For example, students can write a new expression as:

true identity

The above expression is equivalent to:

(\x.\y.x) (\x.x)

Note that the tool supplies the required brackets to separate the two expressions. A facility to remove name bindings selected by the student, or to remove all name bindings is provided.

Example expressions can be specifically written to illustrate individual concepts in the reduction of λ expressions. For example, a good expression to illustrate the name capture problem is:

Let namecapture = (\x.\c.x c) c

A good expression to illustrate β reduction is:

Let betaexample = (\x.\y. x y) (\z.z)

The student can then refer to these expressions, and allow the expressions to be reduced.

Choosing the Sub-Expression to be Reduced

To reduce an expression, the tool can either choose the next sub-expression to be reduced according to a reduction scheme chosen by the student, or the student can select the sub-expression themselves, using a sub-expression chooser provided by the tool. The reduction schemes provided by the tool are normal and applicative reduction. If the tool chooses the sub-expression to be reduced, it must explain its choice to the student in terms of reduction orders.

Reducing the Expression

The student can't select the type of reduction to apply to the sub-expression that they or the tool chose. This is because only one type of reduction (if any) can ever be applied to the whole of any selected sub-expression.

As a learning aid, it is best to reduce the expressions a step at a time, and explain the reduction to the student. There are three main points to explain about any reduction:

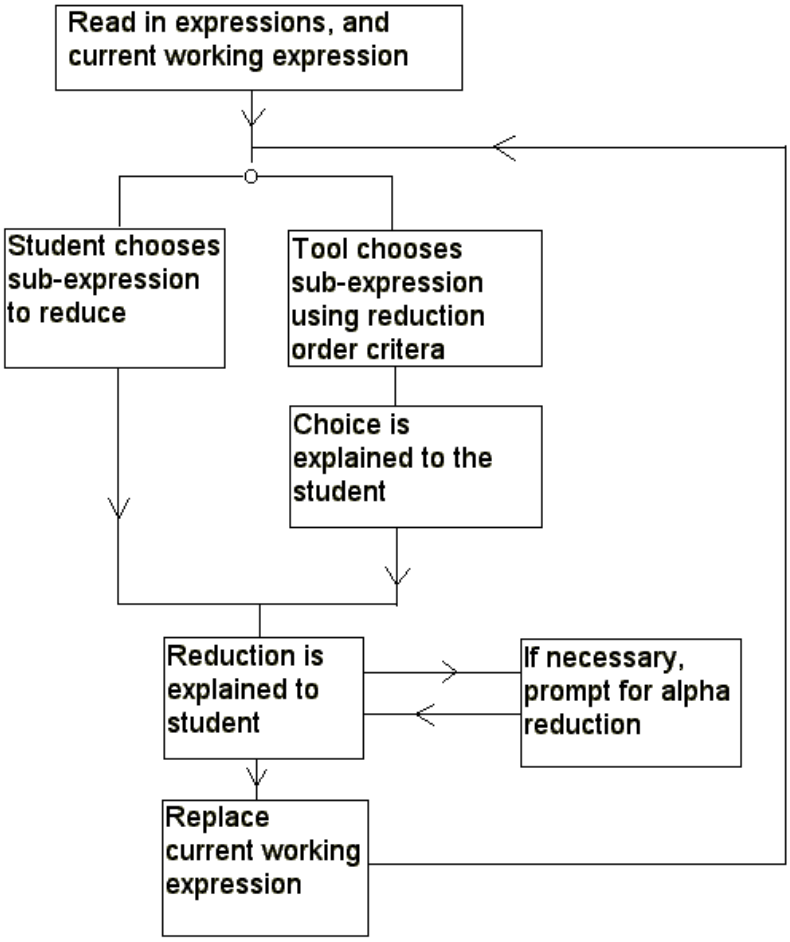
1. Why that particular expression was chosen for reduction. This is concerned with the reduction order being used. If the student chose the sub-expression to be reduced, this explanation is not necessary.

2. What type of reduction will occur on the chosen sub-expression – no reduction, β or η reduction, and the properties of the expression which make it able to be reduced by the chosen reduction method.

3. The details of the reduction. For example, β reduction can be explained as similar to a function application. The function, argument and result can be presented to the student.

One detail to note is that the name capture problem only occurs in β reduction in this implementation. The student never chooses to α reduce a λ abstraction. If α reduction is necessary, the tool alerts the student, and asks the student to enter a new name for the bound variable in the λ abstraction being renamed. The reason for the name-clash is explained to the student when a name-clash occurs.

The cycle of choosing sub-expressions to reduce, and then reducing them with explanation continues until the expression is completely reduced (which is detected by the tool). This cycle is simplified in the following diagram.



4.1.2 Features for Teaching the Mapping of the λ Calculus to Functional Constructs

Entering Expressions

When teaching the mapping from λ calculus to higher-level functional constructs, we wish to be able to name λ expressions, and read these definitions in from a file. Large expressions can then be created from the pre-defined expressions, or new expressions can be named and added to the list of definitions. The ability to create these high-level expressions demonstrates the mapping of the λ calculus to functional constructs for the student. The student may wish to see that these constructs can be reduced to a solution.

Reducing Expressions

The tool can drive the reduction of these higher-level expressions using the built-in reduction orders (normal and applicative), or the student can choose sub-expressions to reduce. It is useful to be able to turn off the explanatory facility associated with reductions when using the tool to demonstrate reductions of complex constructs.

Preservation of Expression Names

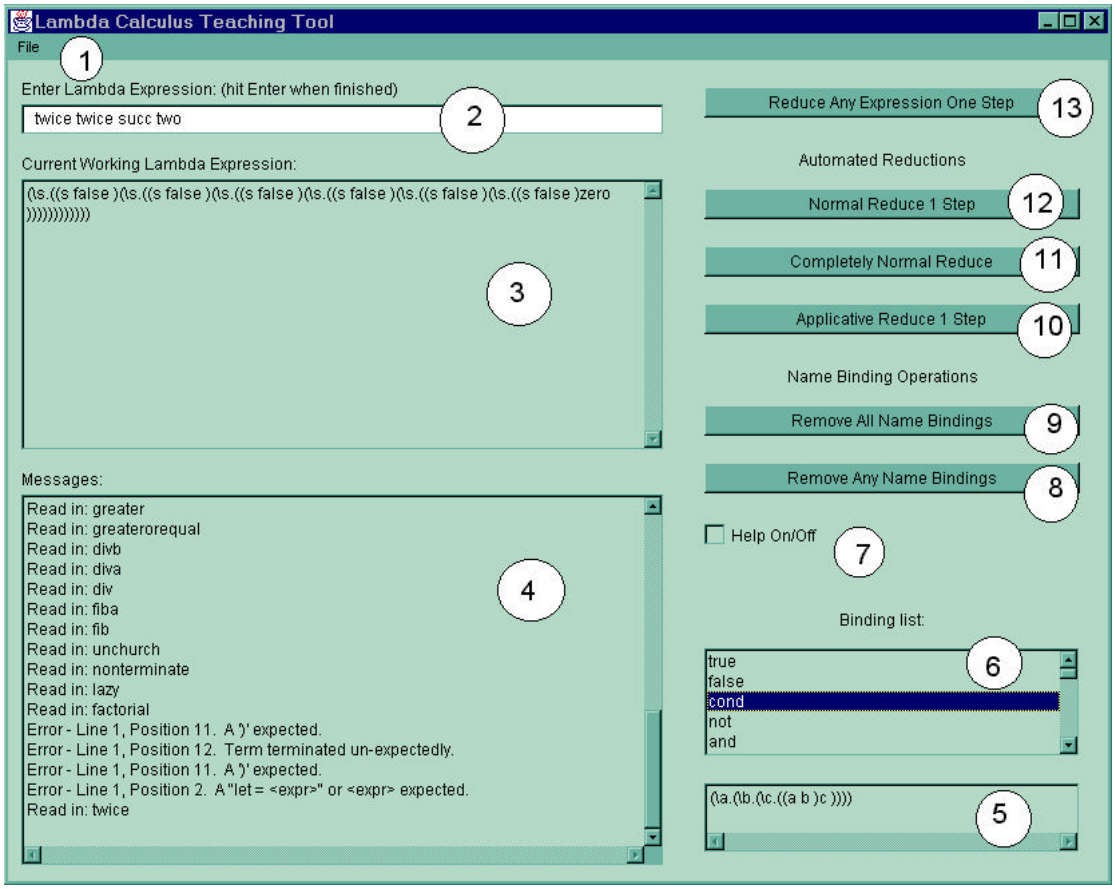
When using the tool to demonstrate a high-level mapping, it is useful for the tool to preserve the name of an expression until that expression is modified. When that expression is modified, the name is discarded. This has the advantage keeping λ expressions as simple as possible.

4.1.3 Features for Using the Tool as a Demonstration Device

The features for using the tool as a demonstration device are the same as those for demonstrating the mapping between λ calculus and functional constructs. Complex expressions can be built up by a teacher for the students, and then reduced by the teacher with explanations. It is important that the tools explanation feature can be turned off for this. The teacher can pursue any reduction order they want, or allow the tool to choose expressions for reduction.

4.2 An Overview of the Tool's Interface

The λ teaching tool created for this project implements all of the features discussed in sections 3 and 4. This section will explain how each interface element of the tool provides the features discussed.



Item #1 is a menu that allows the tool to be exited, or allows an input file to be opened. When an input file is opened, any bindings (named expressions) declared within it are added to the tools environment, and listed in the binding list (6). Each binding is also listed in the Message Window (4) as it is read in.

Item #2 is a text field in which expressions can be entered. These expressions follow the format at the end of section 2. Named expressions can be referred to in these expressions, or new bindings can be declared (note that new bindings can refer to old bindings).

Item #3 displays the current λ expression being operated on. If a binding is present within this expression, the binding name, not its expression is displayed. When an expression is modified, its binding name is discarded.

Item #4 is a message window which displays messages about expressions as they are read in. If an incorrect expression is fed in, an error message is produced which outlines the line number of the input, and the position of the input that the error occurred, and what type of error occurred.

Items #5 and #6 are a list of currently recognised name bindings. When a binding is selected in item #5, its contents are displayed in item #6. The contents of item #6 can then be read into the cut-buffer.

Item #7 turns of explanations of reductions. The only dialogs that will appear during reductions are name capture warning dialogs, which request a new name for the clashing variable.

Item #8 pops up a sub-expression selection dialog box, which can be used to choose a name binding to remove. This discards the name, and instead displays the expression attached to the name. If a sub-expression that isn't a name binding is chosen, an error dialog box appears, and no action is taken. Item #9 removes all name bindings within an expression.

Item #10 applicatively reduces the current working expression one step. If help is turned off, the reduction is simply performed, and the new result

appears in the current working expression text area. If help is turned on, a dialog box explaining applicative reduction order first appears, which also gives the sub-expression chosen for reduction. A second dialog box explaining the actual reduction being performed appears. This explains why that reduction was chosen, and properties of the reduction. The reduction is then performed, and the current working expression replaced with the new reduced expression.

Item #11 completely normal reduces the current λ expression. The only dialog box that appears is an explanation dialog which outlines what normal reduction order is, and name capture dialogs which ask for variables to be renamed. This item is somewhat dangerous because when used on complex recursive or non-terminating expressions, the tool will lock up while it waits for the reduction to terminate.

Item #12 is the same as item #10, except it performs normal reduction.

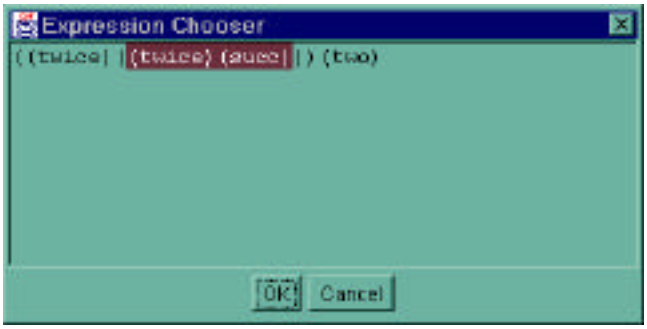
Item #13 allows the student to select any sub-expression they wish for reduction. A dialog box is displayed which asks the student to select a sub-expression to reduce (the student clicks on the sub-expression they wish to reduce, and it is highlighted). If help is on, the tool then explains the reduction (β or η), and performs the reduction. If the selected expression is not reducible, this is explained to the student.

4.2.1 Dialog Boxes Used by the λ Tool

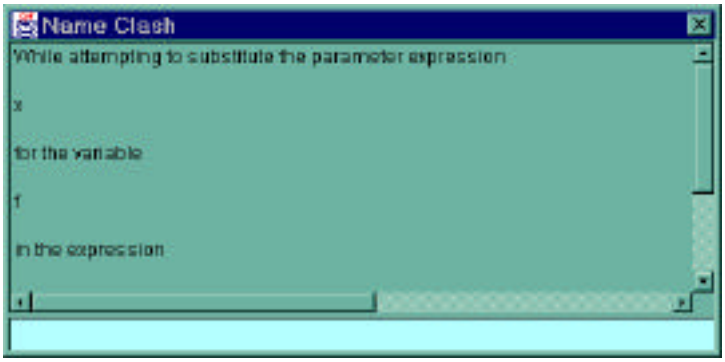
There are three major types of dialog box used by the λ tool. They are all modal (ie. the only part of the tool that can be interacted with while the dialog is up is the dialog itself).

The first of these is a dialog to select a sub-expression. This dialog box is used when the student wants to drive the reduction, or select a name binding to remove. The expression is chosen by clicking on the desired expression. The chosen expression is highlighted. The student can

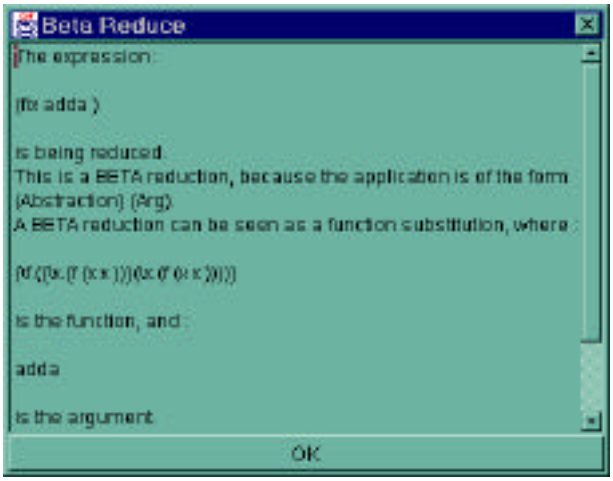
change their mind and select another expression before selecting ‘ok’. Selecting ‘cancel’ cancels the operation that brought the dialog box up. The expression chooser dialog appears below.



The second of these dialog boxes is brought up when name capture occurs. The name capture is explained to the student in the scrolling text area, and the student is prompted to enter a new name for the clashing bound variable. If the new name-clashes, the student is prompted again. The name capture dialog appears below.



The final type of dialog box is a simple dismissible explanatory dialog box which is used to explain why the tool selected a given sub-expression for reduction, and also used to explain the specific reduction performed on an expression. An example of this dialog appears below.



4.3 Summary of Tool Features

All of the features described in this section can be used by a student to reduce expressions of varying complexity. Expressions can be named, and the names preserved to hide complexity. In the sense of choosing what to reduce, the tool is a blank slate. The expressions that can be given to the students in an input file, and the instructions for using those expressions are left to the teachers discretion. A set of example input expressions are provided in Appendix A. The expressions in Appendix A are well suited for demonstrating the mapping between λ expressions and functional constructs.

5 Implementation Details of the λ Calculus Tool

This chapter outlines the implementation of the λ calculus teaching tool. The architecture of the tool as a whole is explained, as well as implementation details (for instance, a description of the tools reduction engine).

5.1 Implementation Language and Target Platform

A requirement for the λ calculus teaching tool is that it must run on a range of platforms. This is because universities may change the architecture of the students' machines at any time. The λ calculus teaching tool is implemented in Java 1.1. Java is a platform independent interpreted object-oriented programming language. Java's platform independence is achieved in several ways:

1. All data types in Java are specified to be machine independent.
2. Java provides a standard library of I/O, networking, utility, and windowing classes which are guaranteed to be available on all platforms. The consistency and functionality of these classes across platforms is also specified.
3. Java programs are compiled into a machine independent form (byte-codes), which are then interpreted by a virtual machine on the platform of choice at runtime.

This platform independence makes Java an ideal language for implementing an interactive teaching tool. Java also has the advantage of being an object-oriented language, which increases the tools modularity, and makes implementation easier.

5.1.1 Java Applet or Java Application?

In the Java programming language, there are two main types of executable program that can be developed. These are the Java applet and the Java application. A Java application consists of a group of compiled Java classes, one of which contains a "main" method where execution begins. A Java application is run on a Java interpreter.

A Java applet is a class that is run by an already running Java application (usually inside a web browser or applet viewer). Java applets are designed to be loaded over a network. This raises many security issues. For this reason, most applet viewers prohibit file system access. The λ calculus teaching tool is implemented as a Java application due to its need to load files that contain definitions of λ expressions. Under Java 1.1, it is possible for applets to be digitally signed and trusted to access the file system. However, at the time of writing most web browsers aren't Java 1.1 compliant.

5.1.2 Java 1.0 or Java 1.1?

The initial versions of the tool were written in Java 1.0. The tool was later converted to Java 1.1 when it was realised that modal dialog boxes are difficult to manipulate under the Java 1.0 event model. Java 1.1 offers a more flexible event model than Java 1.0, as well as a wider range of library classes.

5.2 Architecture of the Tool

The λ calculus teaching tool can be seen as having four main parts. These parts are:

1. The graphical user interface (GUI), which is implemented separately to the reduction engine.

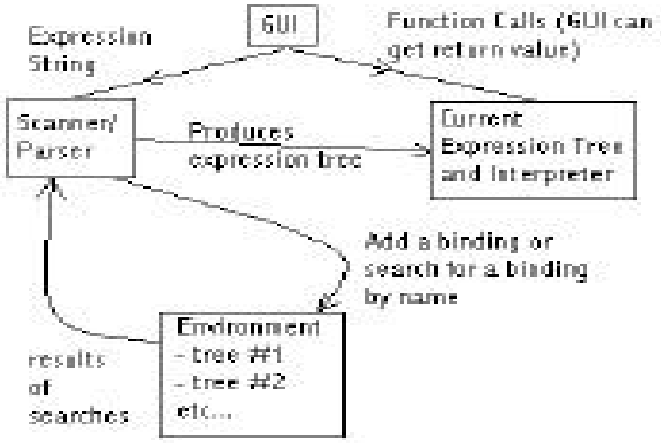
2. A reduction engine, which consists of an object oriented representation of an expression tree.

3. A scanner and parser, which convert text input into expression trees.

4. An environment, which holds named λ expressions (bindings).

   The parser can refer to the environment. The environment is simply a searchable linked list of expression trees, indexed by name.

The main interactions between the components of the λ teaching tool are represented below:



The GUI can be seen as the top layer, which drives the operation of the scanner/parser and the reduction engine. The tool can be seen as having three major actions associated with it.

1. Reading in λ expressions.

2. Manipulating and reducing a current working λ expression.

3. Reporting actions taken to the user.

5.2.1 Reading in λ expressions

Lambda expressions can be read in from a file or a text field in the GUI. This is achieved using a simple scanner and recursive-descent parser. The expressions are converted to an expression tree (explained in section 5.3), and if they are a name-binding, they are stored in the environment. When names are referred to in future, the environment can be consulted, and the retrieved expression substituted for the name in the current working expression. The expression read in is then made the current working expression.

5.2.2 Manipulating and Reducing the Current Expression

The code for handling expression trees defines operations for selecting sub-expressions for reduction, extracting information about expressions, and reducing and manipulating expressions. By using these operations, the GUI can choose a sub-expression for reduction, and display information about both the reduction order and the reduction details to the student. The expression tree can then be made to reduce the chosen sub-expression, and the new resulting expression can be displayed.

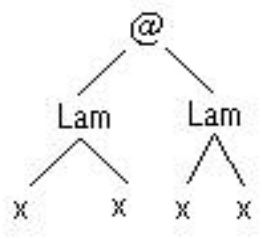
5.2.3 Reporting Actions to the User

In many cases, reporting actions to the user is trivial, because the GUI contains all of the information it needs. For example, if the student pressed the normal reduce button, it is trivial for the GUI to explain normal reduction. The GUI can extract specific details about the current expression, and the reduction being performed by calling functions provided by the implementation of the expression tree.

5.3 The λ Expression Representation and Reduction Engine

The syntax of the λ calculus used by the teaching tool can be found at the end of section 2. Any expression in this syntax is easily converted into a

syntax tree[8]. For example, the expression (λx.x) (λx.x) can be represented as:



The '@' represents a functional application, 'Lam' represents a λ abstraction, and the x's are variables. The connections in the tree represent pointers to data structures. The λ teaching tool represents the current λ expression as a syntax tree.

5.3.1 Precise Definition of λ Expressions in the Tool

Because of Java's object oriented properties, a super-class can be defined, which I have called ***Expression***. This class represents a generic λ expression, and the operations that can be performed on any λ expression are defined within this class. Operations defined for this class include:

- Un-parse the expression to a string
- Functions to reduce the current expression. The reductions provided include β, η, and the reduction of a chosen sub-expression. To reduce a chosen sub-expression, the current expression is searched until the chosen sub-expression is found,

[8] Readers are referred to [Jones87] for a more complete explanation of syntax trees

and then the chosen sub-expression is reduced using the reduction rule appropriate to the expression.

- Functions to check if the expression is reducible for each reduction type (β, and η). These functions return booleans. A function to check if this expression contains reducible sub-expressions is also provided (to check if the expression has reached normal form).

- Functions to find the next sub-expression for normal order reduction and applicative order reduction.

- Functions to check if the current expression contains unbound variables, and which unbound variables it contains.

- A function to perform name substitution on the current expression. A variable name is passed as a parameter, as well as an expression to be substituted for it. Un-bound occurrences of the variable to be replaced have a copy of the input expression substituted for them.

- A function to evaluate any built-in functions within the current expression.

- A function to clone the current expression and all sub-expressions.

These operations are all that is necessary to display information to the user about what reductions are occurring, and perform reductions and expression manipulations.

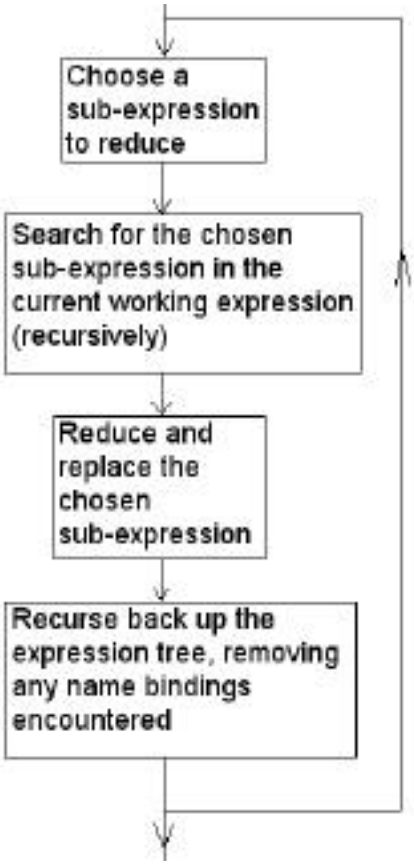
Elements of the λ calculus syntax, such as applications, abstractions, variables, expression bindings, and so on, can be represented as subclasses of the class ***Expression***. This ensures that generic expression operations are available for all classes that represent a λ syntax element.

5.3.1 Reducing the Expression Tree

There are four steps in reducing any expression tree. These steps are:

1. Choose the sub-expression to reduce
2. Recursively search the parent expression until the sub-expression chosen for reduction is found.
3. Reduce the sub-expression chosen for reduction, and place the newly reduced expression into the tree.
4. Climb back up the tree (using recursion), and remove any name-bindings encountered.

A diagram of these four steps is shown below.



The first step in reducing an expression is to choose the sub-expression to reduce. This is done by either using functions provided by expressions to find the normal reducible or applicative reducible sub-expression, or by using the expression chooser dialog box which allows a student to choose a sub-expression.

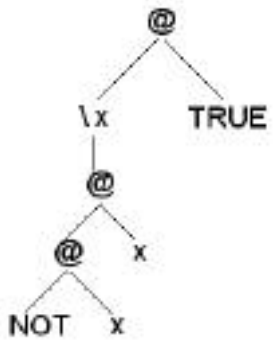
Once the expression to reduce is chosen, information can be extracted by the GUI to display to the student. The sub-expression can then be reduced using a function contained in the ***Expression*** class which takes an expression as a parameter, and looks through the current expression until it finds the input expression, which it then reduces, and places back into the expression tree. This search is done recursively. The advantage of this is that once the reduction has been performed, the new reduced expression can replace the old sub-expression, and any name bindings can be removed from higher up in the tree. Name bindings must be removed when one of their sub-expressions has been modified.

5.3.2 Method Used to Reduce Expressions

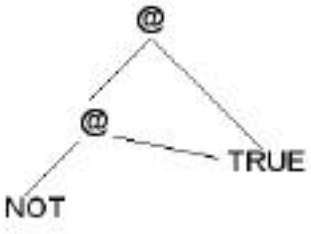
When an expression has been selected to be reduced, there are several possible reduction strategies. The main two are known as graph reduction and tree reduction.

When an application is β reduced using graph reduction, a pointer to the argument is ***substituted*** into every occurrence of the bound variable in the abstractions body. When an application is β reduced using tree reduction (or string reduction), the argument itself is ***copied*** into each occurrence of the abstraction's bound variable in the body. For example, consider the expression:

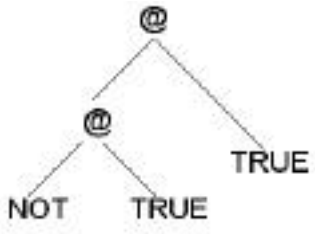
(λx. (NOT x) x) TRUE:



The result of a β reduction using graph reduction on this expression is:



The result of a β reduction using tree reduction is:



Now if the student modifies the TRUE, which is in the leftmost functional application, different things will happen to the tree-reduced expression and the graph reduced expression. In the graph-reduced version, any modifications made to the TRUE will affect both applications (because they both point to the same TRUE). In the tree reduced version, only the TRUE in the leftmost application will be modified.

Graph reduction is good for implementing fast reduction, because expressions are evaluated only once. However, when used in a demonstration device, the student may have difficulty understanding why reducing or modifying one expression causes many other expressions to

change. Tree reduction was used in the λ teaching tool for this reason. The major trade off is that tree reduction is very slow.

6 Future Work and Conclusions

This section will examine the effectiveness of the λ tool when used with undergraduate students, and possible modifications to the tool are suggested for future work. A different approach to teaching the process of learning λ reductions is presented., and suggestions for a study of the effectiveness of the tool are made.

6.1 The Tools Effectiveness as a Teaching Device

A tutorial was set for second year undergraduate students, to gain feedback on how effective the λ teaching tool is. The main problem is getting the students to use the tool, due to the non-compulsory nature of tutorials. Some students who used the tool found the number of brackets presented in λ expressions to be confusing. This problem can be rectified by implementing a more complex expression un-parser that takes expression context into account, and removes non-necessary brackets. Another feature that some people wished for is the ability for the tool to be able to display previous reductions. This can be easily implemented by copying the expression tree before a reduction, and storing the copy. The students can then browse the previous expressions.

Some preliminary feedback suggests that the tool is better at demonstrating the mapping between the λ calculus and higher order functional constructs then it is at teaching details of the reduction process. This is valuable, as many students fail to understand the λ calculus' applicability to functional programming. If the tool can demonstrate the mapping effectively, this problem will have been addressed. The tools comparatively poor performance while teaching the process and details of λ reduction may be because the tool doesn't guide the student in reductions very effectively. The tool can only give examples of normal and applicative

order reductions. Other than that, the student is left to drive reductions, with the tools explanations being their only aid.

6.2 A New Way of Teaching the λ Reduction Process

One problem with the tool is that it doesn't guide the student enough in the reduction process. It either demonstrates a reduction, or the student is left to drive the reduction.

This problem can be overcome by asking the students different types of questions, and letting them attempt to answer the question. If the student answers wrongly, the correct answer can be presented, and then another question of a similar type asked.

For example, the tool could present the student with an expression and ask them which sub-expression would be reduced for normal order reduction. If the student answers incorrectly, the tool could explain the correct answer. Another question of that type could then be asked. This allows the student to attempt to answer questions, but allows progress if they become stuck. The tool could also ask questions like:

- What type of reduction would occur on the expression λx.(\z.z) x.
- Please reduce the expression (λx.λz. z) 2

The programming involved to implement features such as this would be reasonably complex. It may be possible to write a scripting language that the teacher could use to write questions for the λ teaching tool to ask. This would tie the wording of the question to the internal functionality of the tool (for instance, in the first example, the tool must know it is attempting to normal reduce an expression).

This feature could be best implemented as an addition to the tool. That way, the tool can retain its functionality as an effective demonstration

device for teaching the mapping between the λ calculus and functional constructs.

6.3 Suggestions for a Study of the Tools Effectiveness

As explained in section 6.1, the tool was provided to students to evaluate, but no study has yet been made into the tools effectiveness. The tools effectiveness in several area's of knowledge could be gauged using McGill and Volet's knowledge framework (see section 3.1.1). This can be done by testing students on separate aspects of knowledge such as syntactic procedural and declarative conceptual knowledge after using the tool. By implementing tutorials using the tool, and using the tool as a demonstration device, possible correlations between the students' understanding of the λ calculus and use of the tool can be investigated.

6.4 Summary and Tool Usage Suggestions

The λ calculus teaching tool can be seen as a clean slate when it comes to teaching the process of λ reductions, and the mapping from the λ calculus to functional constructs. Students need to be guided while using the tool. They need a supply λ expressions tailored to the area being learned, and instructions on how to use the tool on those expressions. Using the tool as a demonstration device could be very useful in making the students feel comfortable with the tool before they use the tool. It is hoped that the tool will help the students build a mental model of how the λ calculus reduction process works, and how the λ calculus relates to functional constructs. A case study, and possible modification of the tool is required before the tools effectiveness can be tested.

Appendix A: Example λ Expressions

The following λ expressions can be fed into the teaching tool to aid students in their understanding of the mapping between the λ calculus and functional constructs.

Boolean Representation

```
let true  = \x . \y . x
let false = \x . \y . y
{  cond is the equivalent of an if-statement  }
{  The use is cond <condition> <alternative1> <alternative2>  }
let cond  = \a. \b. \c . a b c
```

Boolean Operations

```
let not = \x . ( ( ( cond x ) false ) true )
let and = \x . \y . x y false
let or = \x . \y . x true y
```

Pairs

```
let pair = \x . \y . \z . z x y
let fst  = \n. n true
let snd  = \n. n false

{ The twice function }
let twice = \f. \x. f (f x)
```

A Fixed Point Combinator

```
let fix     = \f. (\x. f (x x))(\x. f (x x))
```

Natural Numbers, Operations and Comparisons

```
let zero    = \z . z
let succ    = \n . \s. ((s false) n)
let one     = succ zero
let two     = succ one

let iszero  = \n . (n true)

let preda   = \n. (n false)
let pred    = \n.(((cond (iszero n)) zero) (preda n))

let adda    = \f. \x. \y. (cond (iszero y) x (f (succ x) (pred y)))
```

```
let add      = fix adda

let multa    = \f.\x.\y. cond (iszero y) zero (add x (f x (pred y)))
let mult     = fix multa

let suba     = \f.\x.\y. cond (iszero y) x (f (pred x) (pred y))
let sub      = fix suba

let powera   = \f.\x.\y. cond (iszero y) one (mult x (f x (pred y)))
let power    = fix powera

let absdiff = \x.\y. add (sub x y) (sub y x)

let equal = \x.\y. iszero (absdiff x y)
let greater = \x.\y. not (iszero (sub x y))
let greaterorequal  = \x.\y.iszero(sub y x)


let divb = \f.\x.\y. cond (greater y x) zero (succ (diva (sub x y) y))
let diva = fix divb
let div = \x.\y.cond (iszero y) zero (diva x y)
```

Miscellaneous Functions

```
{  A fibonacci function  }
let fiba = \f. \n. cond (iszero n) one (cond (equal n one) one (add (f
(sub n one)) (f (sub n two))))
let fib = fix fiba

{ unchurch can be used to turn a church   }
{ numeral into a built-in number.              }
let unchurch = \n . n (\x.(+1 x)) 0

{ A non-terminating expression }
let nonterminate = (\x . x x) (\x . x x)
{ lazy does terminate under normal order reduction }
let lazy  = (\x . 3) nonterminate

{ Example of using fixpoint to define recursive functions, eg factorial }
let factorial = fix (\f . \n . cond (iszero n) one (mult n (f (pred n))))
```

References

[Church41]
Church, A.
The Calculi of Lambda Conversion
Princeton University Press, 1941

[GHT84]
Glaser, H., Hankin, C., and Till, D.
Principles of Functional Programming
Prentice-Hall, 1984

[Cornell97]
Cornell, G., Horstmann, C.
Core Java
Second Edition
Prentice-Hall PTR, 1997

[Jones87]
Peyton Jones, S.L.
The Implementation of Functional Programming Languages
Prentice-Hall International Series in Computer Science, 1987

[Michaelson89]
Michaelson, G.
An Introduction to Functional Programming Through Lambda Calculus
International Computer Science Series, Addison-Wesley, 1989

[Rosser82]
Rosser, J.B.
Highlights of the History of the Lambda Calculus
Proceedings of the A.C.M. Symposium on LISP and Functional Programming, August 1982, pp. 216 - 225

[Russell]
Russell, T.
The No Significant Difference Phenomenon – Russell, T.

[Volet1]
Volet, S., McGill, T., Pears, H.
Implementing Process Based Instruction in Regular University Teaching - Conceptual, Methodological and Practical Issues
European Journal of Psychology of Education

[Volet2]
Volet, S.
Process Oriented Instruction - A Discussion
European Journal of Psychology of Education

[Volet3]
Volet, S. E., McGill, T. J.
A Conceptual Framework for Analysing Students' Knowledge of Programming
Journal of Research on Computing in Education