Text is in Times Roman. **New terms and important concepts are in bold Times Roman**.

`Programs and definitions are in Courier`.

Greek letters are used in naming λ calculus concepts:

α - alpha
β - beta
λ - lambda
η - eta

Syntactic constructs are defined using BNF **rules.** Each rule has a **rule name** consisting of one or more words within angle brackets `<` and `>`. A rule associates its name with a **rule body** consisting of a sequence of **symbols** and rule names. If there are different possible rule bodies for the same rule then they are separated by `|`s.

For example, binary numbers are based on the digits `1` and `0`:

```
<digit> ::= 1 | 0
```

and a binary number may be either a single digit or a digit followed by a number:

```
<binary> ::= <digit> | <digit> <binary>
```

1.13. Summary

In this chapter we have discussed the differences between imperative and functional programming and seen that:

- imperative languages are based on assignment sequences whereas functional langugaes are based on nested function calls
- in imperative languages, the same name may be associated with several values whereas in functional languages a name is only associated with one value
- imperative languages have fixed evaluation orders whereas functional languages need not
- in imperative languages, new values may be associated with the same name through command repetition whereas in functional languages new names are associated with new values through recursive function call nesting
- functional languages provide explicit data structure representations
- in functional languages, functions are values

We have also seen that:

- functional languages originate in mathematical logic and the theory of computing, in recursive function theory and λ calculus

2. LAMBDA CALCULUS

2.1. Introduction

In this chapter we are going to meet the λ calculus, which will be used as a basis for functional programming in the rest of the book.

To begin with, we will briefly discuss abstraction as generalisation through the introduction of names in expressions and specialisation through the replacement of names with values. We will also consider the role of abstraction in programming languages.

Next we will take an overview of the λ calculus as a system for abstractions based on functions and function applications. We will discuss informally the rules for constructing and evaluating λ expressions in some detail.

We will introduce a notation for defining named functions and look at how functions may be constructed from other functions.

We will then consider functions for manipulating pairs of values. These will be used as building blocks in subsequent chapters to add syntax for booleans, numbers, types and lists to λ calculus.

Lastly, we will take a slightly more formal look at λ expression evaluation.

2.2. Abstraction

Abstraction is central to problem solving and programming. It involves generalisation from concrete instances of a problem so that a general solution may be formulated. A general, abstract, solution may then be used in turn to solve particular, concrete instances of the problem.

The simplest way to specify an instance of a problem is in terms of particular concrete operations on particular concrete objects. Abstraction is based on the use of names to stand for concrete objects and operations to generalise the instances. A generalised instance may subsequently be turned into a particular instance by replacing the names with new concrete objects and operations.

We will try to get a feel for abstraction through a somewhat contrived example. Consider buying 9 items at 10 pence each. The total cost is:

```
10*9
```

Here we are carrying out the concrete operation of `multiplication` on the concrete values `10` and `9`. Now consider buying 11 items at 10 pence each. The total cost is:

```
10*11
```

Here we are carrying out the concrete operation of `multiplication` on the concrete values `10` and `11`.

We can see that as the number of items changes so the formula for the total cost changes at the place where the number of items appears. We can **abstract over** the number of items in the formula by introducing a name to stand for a general number of items, say `items`:

```
10*items
```

We might make this abstraction explicit by preceding the formula with the name used for abstraction:

```
REPLACE items IN 10*items
```

Here, we have abstracted over an operand in the formula. To evaluate this abstraction we need to supply a value for the name. For example, for 84 items:

```
REPLACE items WITH 84 IN 10*items
```

which gives:

```
10*84
```

We have made a **function** from a formula by replacing an object with a name and identifying the name that we used. We have then **evaluated** the function by replacing the name in the formula with a new object and evaluating the resulting formula.

Let us use abstraction again to generalise our example further. Suppose the cost of items goes up to 11 pence. Now the total cost of items is:

```
REPLACE items IN 11*items
```

Suppose the cost of items drops to 9 pence. Now the total cost of items is:

```
REPLACE items IN 9*items
```

Because the cost changes, we could also introduce a name to stand for the cost in general, say `cost:`

```
REPLACE cost IN
 REPLACE items IN cost*items
```

Here we have abstracted over two operands in the formula. To evaluate the abstraction we need to supply two values. For example, 12 items at 32 pence will have total cost:

```
REPLACE cost WITH 32 IN
 REPLACE items WITH 12 IN cost*items
```

which is:

```
REPLACE items WITH 12 IN 32*items
```

which is:

```
32*12
```

For example, 25 items at 15 pence will have total cost:

```
REPLACE cost WITH 15 IN
 REPLACE items WITH 25 IN cost*items
```

which is:

```
REPLACE items WITH 25 IN 15*items
```

which is:

```
15*25
```

Suppose we now want to solve a different problem. We are given the total cost and the number of items and we want to find out how much each item costs. For example, if 12 items cost 144 pence then each item costs:

```
144/12
```

If 15 items cost 45 pence then each item costs:

```
45/12
```

In general, if `items` items cost `cost` pence then each item costs:

```
REPLACE cost IN
 REPLACE items IN cost/items
```

Now, compare this with the formula for finding a total cost:

```
REPLACE cost IN
 REPLACE items IN cost*items
```

They are the same except for the operation `/` in finding the cost of each item and `*` in finding the cost of all items. We have two instances of a problem involving applying an operation to two operands. We could generalise these instances by introducing a name, say `op,` where the operation is used:

```
REPLACE op IN
 REPLACE cost IN
  REPLACE items IN cost op items
```

Now, finding the total cost will require the replacement of the operation name with the concrete `multiplication` operation:

```
REPLACE op WITH * IN
 REPLACE cost IN
  REPLACE items IN cost op items
```

which is:

```
REPLACE cost IN
 REPLACE items IN cost * items
```

Similarly, finding the cost of each item will require the replacement of the operation name with the concrete `division` operation:

```
REPLACE op WITH / IN
 REPLACE cost  IN
  REPLACE item IN cost op items
```

which is:

```
REPLACE cost IN
 REPLACE items IN cost / items
```

Abstraction is based on generalisation through the introduction of a name to replace a value and specialisation through the replacement of a name with another value.

Note that care must be taken with generalisation and replacement to ensure that names are replaced by objects of appropriate types. For example, in the above examples, the operand names must be replaced by numbers and the operator name must be replaced by an operation with two number arguments. We will look at this in slightly more detail in chapter 5.

2.3. Abstraction in programming languages

Abstraction lies at the heart of all programming languages. In imperative languages, variables as name/value associations are abstractions for computer memory locations based on specific address/value associations. The particular address for a variable is irrelevant so long as the name/value association is consistent. Indeed, on a computer with memory management, a variable will correspond to many different concrete locations as a program's data space is swapped in and out of different physical memory areas. The compiler and the run time system make sure that variables are implemented as consistent, concrete locations.

Where there are abstractions there are mechanisms for introducing them and for specialising them. For example, in Pascal, variables are introduced with declarations and given values by statements for use in subsequent statements. Variables are then used as abstractions for memory addresses on the left of assignment statements or in READ

statements, and as abstractions for values in expressions on the right of assignment statements or in `WRITE` statements.

Abstractions may be subject to further abstraction. This is the basis of hierarchical program design methodologies and modularity. For example, Pascal procedures are abstractions for sequences of statements, named by procedure declarations, and functions are abstractions for expressions, named by function declarations. Procedures and functions declare formal parameters which identify the names used to abstract in statement sequences and expressions. Simple, array and record variable formal parameters are abstractions for simple, array and record variables in statements and expressions. Procedure and function formal parameters are abstractions for procedures and functions in statements and expressions. Actual parameters specialise procedures and functions. Procedure calls with actual parameters invoke sequences of statements with formal parameters replaced by actual parameters. Similarly, function calls with actual parameters evaluate expressions with formal parameters replaced by actual parameters.

Programming languages may be characterised and compared in terms of the abstraction mechanisms they provide. Consider, for example, Pascal and BASIC. Pascal has distinct `INTEGER` and `REAL` variables as abstractions for integer and real numbers whereas BASIC just has numeric variables which abstract over both. Pascal also has `CHAR` variables as abstractions for single letters. Both Pascal and BASIC have arrays which are abstractions for sequences of variables of the same type. BASIC has string variables as abstractions for letter sequences whereas in Pascal an array of `CHAR`s is used. Pascal also has records as abstractions for sequences of variables of differing types. Pascal has procedures and functions as statement and expression abstractions. Furthermore, procedure and function formal parameters abstract over procedures and functions within procedures and functions. The original BASIC only had functions as expression abstractions and did not allow function formal parameters to abstract over functions in expressions.

In subsequent chapters we will see how abstraction may be used to define many aspects of programming languages.

2.4. λ calculus

The λ calculus was devised by Alonzo Church in the 1930's as a model for computability and has subsequently been central to contemporary computer science. It is a very simple but very powerful language based on pure abstraction. It can be used to formalise all aspects of programming languages and programming and is particularly suited for use as a 'machine code' for functional languages and functional programming.

In the rest of this chapter we are going to look at how λ calculus expressions are written and manipulated. This may seem a bit disjointed at first: it is hard to introduce all of a new topic simultaneously and so some details will be rather sketchy to begin with.

We are going to build up a set of useful functions bit by bit. The functions we introduce in this chapter to illustrate various aspects of λ calculus will be used as building blocks in later chapters. Each example may assume previous examples and so it is important that you work through the material slowly and consistently.

2.5. λ expressions

The λ calculus is a system for manipulating **λ expressions**. A λ expression may be a **name** to identify an abstraction point, a **function** to introduce an abstraction or a **function application** to specialise an abstraction:

```
<expression> ::= <name> | <function> | <application>
```

A name may be any sequence of non-blank characters, for example:

```
fred  legs-11  19th_nervous_breakdown  33  +  -->
```

A λ function is an abstraction over a λ expression and has the form:

```
<function> ::= λ<name>.<body>
```

where

```
<body> ::= <expression>
```

for example:

```
λx.x  λfirst.λsecond.first  λf.λa.(f a)
```

The λ precedes and introduces a name used for abstraction. The name is called the function's **bound variable** and is like a formal parameter in a Pascal function declaration. The `.` separates the name from the expression in which abstraction with that name takes place. This expression is called the function's **body.**

Notice that the body expression may be any λ expression including another function. This is far more general than, for example, Pascal which does not allow functions to return functions as values.

Note that functions do not have names! For example, in Pascal, the function name is always used to refer to the function's definition.

A function application has the form:

```
<application> ::= (<function expression> <argument expression>)
```

where

```
<function expression> ::= <expression>
<argument expression> ::= <expression>
```

for example:

```
(λx.x λa.λb.b)
```

A function application specialises an abstraction by providing a value for the name. The function expression contains the abstraction to be specialised with the argument expression.

In a function application, also known as a **bound pair**, the function expression is said to be **applied to** the argument expression. This is like a function call in Pascal where the argument expression corresponds to the actual parameter. The crucial difference is that in Pascal the function name is used in the function call and the implementation picks up the corresponding definition. The λ calculus is far more general and allows function definitions to appear directly in function calls.

There are two approaches to evaluating function applications. For both, the function expression is evaluated to return a function. Next, all occurences of the function's bound variable in the function's body expression are replaced by *either*

the value of the argument expression

or

the unevaluated argument expression

Finally, the function body expression is then evaluated.

The first approach is called **applicative order** and is like Pascal 'call by value': the actual parameter expression is evaluated before being passed to the formal parameter.

The second approach is called **normal order** and is like 'call by name' in ALGOL 60: the actual parameter expression is not evaluated before being passed to the formal parameter.

As we will see, normal order is more powerful than applicative order but may be less efficient. For the moment, all function applications will be evaluated in normal order.

The syntax allows a single name as a λ expression but in general we will restrict single names to the bodies of functions. This is so that we can avoid having to consider names as objects in their own right, like LISP or Prolog literals for example, as it complicates the presentation. We will discuss this further later on.

We will now look at a variety of simple λ functions.

2.6. Identity function

Consider the function:

```
λx.x
```

This is the identity function which returns whatever argument it is applied to. Its bound variable is:

```
x
```

and its body expression is the name:

```
x
```

When it is used as the function expression in a function application the bound variable `x` will be replaced by the argument expression in the body expression `x` giving the original argument expression.

Suppose the identity function is applied to itself:

```
(λx.x λx.x)
```

This is a function application with:

```
λx.x
```

as the function expression and:

```
λx.x
```

as the argument expression.

When this application is evaluated, the bound variable:

```
x
```

for the function expression:

```
λx.x
```

is replaced by the argument expression:

```
λx.x
```

in the body expression:

```
x
```

giving:

```
λx.x
```

An identity operation always leaves its argument unchanged. In arithmetic, adding or subtracting `0` are identity operations. For any number `<number>`:

```
<number> + 0 = <number>
<number> - 0 = <number>
```

Multiplying or dividing by `1` are also identity opeartions:

```
<number> * 1 = <number>
<number> / 1 = <number>
```

The identity function is an identity operation for λ functions.

We could equally well have used different names for the bound variable, for example:

```
λa.a
```

or:

```
λyibble.yibble
```

to define other versions of the identity function. We will consider naming in more detail later but note just now that we can consistently change names.

2.7. Self application function

Consider the rather odd function:

```
λs.(s s)
```

which applies its argument to its argument. The bound variable is:

```
s
```

and the body expression is the function application:

```
(s s)
```

which has the name:

```
s
```

as function expression and the same name:

```
s
```

as argument expression.

Let us apply the identity function to it:

```
(λx.x λs.(s s))
```

In this application, the function expression is:

```
λx.x
```

and the argument expression is:

```
λs.(s s)
```

When this application is evaluated, the function expression bound variable:

```
x
```

is replaced by the argument:

```
λs.(s s)
```

in the function expression body:

```
x
```

giving:

```
λs.(s s)
```

which is the original argument.

Let us apply this self application function to the identity function:

```
(λs.(s s) λx.x)
```

Here, the function expression is:

```
λs.(s s)
```

and the argument expression is:

```
λx.x
```

When this application is evaluated, the function expression bound variable:

```
s
```

is replaced by the argument:

```
λx.x
```

in the function expression body:

```
(s s)
```

giving a new application:

```
(λx.x λx.x)
```

with function expression:

```
λx.x
```

and argument expression:

```
λx.x
```

This is now evaluated as above giving the final value:

```
λx.x
```

Consider the application of the self application function to itself:

```
(λs.(s s) λs.(s s))
```

This application has function expression:

```
λs.(s s)
```

and argument expression:

```
λs.(s s)
```

To evaluate it, the function expression bound variable:

```
s
```

is replaced by the argument:

```
λs.(s s)
```

in the function expression body:

```
(s s)
```

giving a new application:

```
(λs.(s s) λs.(s s))
```

with function expression:

```
λs.(s s)
```

and argument expression:

```
λs.(s s)
```

which is then evaluated. The function expression bound variable:

```
s
```

is replaced by the argument:

```
λs.(s s)
```

in the function expression body:

```
(s s)
```

giving the new application:

```
(λs.(s s) λs.(s s))
```

which is then evaluated...

Each application evaluates to the original application so this application never terminates!

We will use a version of this self application function to construct recursive functions in chapter 4. Here, we will note that not all expression evaluations terminate. In fact, as we will see in chapter 8, there is no way of telling whether or not an expression evaluation will ever terminate!

2.8. Function application function

Consider the function:

```
λfunc.λarg.(func arg)
```

This has bound variable:

```
func
```

and the body expression is another function:

```
λarg.(func arg)
```

which has bound variable:

```
arg
```

and a function application:

```
(func arg)
```

as body expression. This in turn has the name:

```
func
```

as function expression and the name:

```
arg
```

as argument expression.

When used, the whole function returns a second function which then applys the first function's argument to the second function's argument. For example, let us use it to apply the identity function to the self application function:

```
((λfunc.λarg.(func arg) λx.x) λs.(s s))
```

In this example of an application, the function expression is itself an application:

```
(λfunc.λarg.(func arg) λx.x)
```

which must be evaluated first. The bound variable:

```
func
```

is replaced by the argument:

```
λx.x
```

in the body:

```
λarg.(func arg)
```

giving:

```
λarg.(λx.x arg)
```

which is a new function which applies the identity function to its argument. The original expression is now:

```
(λarg.(λx.x arg) λs.(s s))
```

and so the bound variable:

```
arg
```

is replaced by the argument:

```
λs.(s s)
```

in the body:

```
(λx.x arg)
```

giving:

```
(λx.x λs.(s s))
```

which is now evaluated as above. The bound variable:

```
x
```

is replaced by the argument:

```
λs.(s s)
```

in the body

```
x
```

giving:

```
λs.(s s)
```

2.9. Introducing new syntax

As our λ expressions become more elaborate they become harder to work with. To simplify working with λ expressions and to construct a higher level functional language we will allow the use of more concise notations. For example, in this and subsequent chapters we will introduce named function definitions, infix operations, an `IF` style conditional expression and so on. This addition of higher level layers to a language is known as **syntactic sugaring** because the representation of the language is changed but the underlying meaning stays the same.

We will introduce new syntax for commonly used constructs through substitution rules. The application of these rules won't involve making choices. Their use will lead to pure λ exprtessions after a finite number of steps involving simple substitutions. This is to ensure that we can always 'compile' completely a higher level representation into λ calculus before evaluation. Then we only need to refer to our original simple λ calculus rules for evaluation. In this way we won't need to modify or augment the λ calculus itself and, should we need to, we can rely on the existing

theories for λ calculus without developing them further.

Furthermore, we are going to use λ calculus as an time order independent language to investigate time ordering. Thus, our substitution rules should also be time order independent. Otherwise, different substitution orders might lead to different substitutions being made and these might result in expressions with different meanings. The simplest way to ensure time order independence is to insist that all substitutions be made statically or be capable of being made statically. We can then apply them to produce pure λ calculus before evaluation starts.

We won't always actually make all substitutions before evaluation as this would lead to pages and pages of incomprehensible λ expressions for large higher level expressions. **We will insist, however, that making all substitutions before evaluation always remains a possibility.**

2.10. Notation for naming functions and application reduction

It is a bit tedious writing functions out over and over again. We will now name functions using:

```
def <name> = <function>
```

to define a name/function association.

For example, we could name the functions we looked at in the previous sections:

```
def identity = λx.x

def self_apply = λs.(s s)

def apply = λfunc.λarg.(func arg)
```

Now we can just use the `<name>` in expressions to stand for the `<function>`.

Strictly speaking, all defined names in an expression should be replaced by their definitions before the expression is evaluated. However, for now we will only replace a name by its associated function when the name is the function expression of an application. We will use the notation:

```
(<name> <argument>) == (<function> <argument>)
```

to indicate the replacement of a `<name>` by its associated `<function>`.

Formally, the replacement of a bound variable with an argument in a function body is called **β reduction (beta reduction)**. In future, instead of spelling out each β reduction blow by blow we will introduce the notation:

```
(<function> <argument>) => <expression>
```

to mean that the `<expression>` results from the application of the `<function>` to the `<argument>`.

When we have seen a sequence of reductions before or we are familiar with the functions involved we will omit the reductions and write:

```
=> ... =>
```

to show where they should be.

2.11. Functions from functions

We can use the self application function to build versions of other functions. For example, let us define a function with the same effect as the identity function:

```
def identity2 = λx.((apply identity) x)
```

Let us apply this to the identity function:

```
(identity2 identity) ==

(λx.((apply identity) x) identity) =>

((apply identity) identity) ==

((λfunc.λarg.(func arg) identity) identity) =>

(λarg.(identity arg) identity) =>

(identity identity) => ... =>

identity
```

Let us show that `identity` and `identity2` are equivalent. Suppose:

```
<argument>
```

stands for any expression. Then:

```
(identity2 <argument>) ==

(λx.((apply identity) x) <argument>) =>

((apply identity) <argument>) => ... =>

(identity <argument>) => ... =>

<argument>
```

so `identity` and `identity2` have the same effect.

We can use the function application function to define a function with the same effect as the function application function itself. Suppose:

```
<function>
```

is any function. Then:

```
(apply <function>) ==

(λf.λa.(f a) <function>) =>

λa.(<function> a)
```

Applying this to any argument:

```
<argument>
```

we get:

```
(λa.(<function> a) <argument>) =>

(<function> <argument>)
```

which is the application of the original function to the argument. Using `apply` adds a layer of β reduction to an application.

We can also use the function application function slightly differently to define a function with the same effect as the self application function:

```
def self_apply2 = λs.((apply s) s)
```

Let us apply this to the identity function:

```
(self_apply2 identity) ==

(λs.((apply s) s) identity) =>

((apply identity) identity) => ... =>

(identity identity) => ... =>

identity
```

In general, applying `self_apply2` to any argument:

```
<argument>
```

gives:

```
(self_apply2 <argument>) ==

(λs.((apply s) s) <argument>) =>

((apply <argument>) <argument>) => ... =>

(<argument> <argument>)
```

so `self_apply` and `self_apply2` have the same effect.

2.12. Argument selection and argument pairing functions

We are now going to look at functions for selecting arguments in nested function applications. We will use these functions a great deal later on to model boolean logic, integer arithmetic and list data structures.

2.12.1. Selecting the first of two arguments

Consider the function:

```
def select_first = λfirst.λsecond.first
```

This function has bound variable:

```
first
```

and body:

```
λsecond.first
```

When applied to an argument, it returns a new function which when applied to another argument returns the first argument. For example:

```
((select_first identity) apply) ==

((λfirst.λsecond.first identity) apply) =>

(λsecond.identity apply) =>

identity
```

In general, applying `select_first` to arbitrary arguments:

```
<argument1>
```

and:

```
<argument2>
```

returns the first argument:

```
((select_first <argument1>) <argument2>) ==

((λfirst.λsecond.first <argument1>) <argument2>) =>

(λsecond.<argument1> <arguement2>) =>

<argument1>
```

2.12.2. Selecting the second of two arguments

Consider the function:

```
def select_second = λfirst.λsecond.second
```

This function has bound variable:

```
first
```

and body:

```
λsecond.second
```

which is another version of the identity function. When applied to an argument `select_second` returns a new function which when applied to another argument returns the other argument. For example:

```
((select_second identity) apply) ==

((λfirst.λsecond.second identity) apply) =>

(λsecond.second apply) =>

apply
```

The first argument `identity` was lost because the bound variable `first` does not appear in the body `λsecond.second`.

In general, applying `select_second` to arbitrary arguments:

```
<argument1>
```

and:

```
<argument2>
```

returns the second arguemt:

```
((select_second <argument1>) <argument2>) ==

((λfirst.λsecond.second <argument1>) <argument2>) =>

(σecond.second <argument2>) =>

<argument2>
```

We can show that `select_second` applied to anything returns a version of `identity.` As before, we will use:

```
<argument>
```

to stand for an arbitrary expression, so:

```
(select_second <argument>) ==

(λfirst.λsecond.second <argument>) =>

λsecond.second
```

If `second` is replaced by `x` then:

```
λsecond.second
```

becomes:

```
λx.x
```

Notice that `select_first` applied to `identity` returns a version of `select_second`:

```
(select_first identity) ==

(λfirst.λsecond.first identity) =>

λsecond.identity ==

λsecond.λx.x
```

If `second` is replaced by `first` and `x` by `second` then this becomes:

```
λfirst.λsecond.second
```

2.12.3. Making pairs from two arguments

Consider the function:

```
def make_pair = λfirst.λsecond.λfunc.((func first) second)
```

with bound variable:

```
first
```

and body:

```
λsecond.λfunc.((func first) second)
```

This function applies argument `func` to argument `first` to build a new function which may be applied to argument `second`. Note that arguments `first` and `second` are used before argument `func` to build a function:

```
λfunc.((func first) second)
```

Now, if this function is applied to `select_first` then argument `first` is returned and if it is applied to `select_second` then argument `second` is returned.

For example:

```
((make_pair identity) apply) ==

((λfirst.λsecond.λfunc.((func first) second)
 identity) apply) =>

(λsecond.λfunc.((func identity) second) apply) =>

λfunc.((func identity) apply)
```

Now, if this function is applied to `select_first`:

```
(λfunc.((func identity) apply) select_first) ==

((select_first identity) apply) ==

((λfirst.λsecond.first identity) apply) =>

(λsecond.identity apply) =>

identity
```

and if it is applied to `select_second`:

```
(λfunc.((func identity) apply) select_second) ==

((select_second identity) apply) ==

((λfirst.λsecond.second identity) apply) =>

(λsecond.second apply) =>

apply
```

In general, applying `make_pair` to arbitrary arguments:

```
<argument1>
```

and

```
<argument2>
```

gives:

```
((make_pair <argument1>) <argument2>) ==
((λfirst.λsecond.λfunc.((func first) second)) <argument1>) <argument2>) =>
(λsecond.λfunc.((func <argument1>) second) <argument2>) =>
λfunc.((func <argument1>) <argument2>)
```

Thereafter, applying this function to `select_first` returns the first argument:

```
(λfunc.((func <argument1>) <argument2>) select_first) =>
((select_first <argument1>) <argument2>) ==
((λfirst.λ second.first <argument1>) <argument2>) =>
(λsecond.<argument1> <argument2>) =>
<argument1>
```

and applying this function to `select_second` returns the second argument:

```
(λfunc.((func <argument1>) <argument2>) select_second) =>
((select_second <argument1>) <argument2>) ==
((λfirst.λ second.second <argument1>) <argument2>) =>
(λsecond.second <argument2>) =>
<argument2>
```

2.13. Free and bound variables

We are now going to consider how we ensure that arguments are substituted correctly for bound variables in function bodies. If all the bound variables for functions in an expression have distinct names then there is no problem.

For example, in:

```
(λf.(f λx.x) λs.(s s))
```

there are three functions. The first has bound variable `f,` the second has bound variable `x` and the third has bound variable `s.` Thus:

```
(λf.(f λx.x) λs.(s s)) =>
(λs.(s s) λx.x) =>
```

```
(λx.x λx.x) =>

λx.x
```

It is possible, however, for bound variables in different functions to have the same name. Consider:

```
(λf.(f λf.f) λs.(s s))
```

This should give the same result as the previous expression. Here, the bound variable `f` should be replaced by:

```
λs.(s s)
```

Note that we should replace the first `f` in:

```
(f λf.f)
```

but not the `f` in the body of:

```
λf.f
```

This is a new function with a new bound variable which just happens to have the same name as a previous bound variable.

To clarify this we need to be more specific about how bound variables relate to variables in function bodies. For an arbitrary function:

```
λ<name>.<body>
```

the bound variable `<name>` may correspond to occurrences of `<name>` in `<body>` and nowhere else. Formally, the **scope** of the bound variable `<name>` is `<body>`.

For example, in:

```
λf.λs.(f (s s))
```

the bound variable `f` is in scope in:

```
λs.(f (s s))
```

In:

```
(λf.λg.λa.(f (g a)) λg.(g g))
```

the leftmost bound variable `f` is in scope in:

```
λg.λa.(f (g a))
```

and nowhere else. Similarly, the rightmost bound variable `g` is in scope in:

```
(g g)
```

and nowhere else.

Note that we have said *may correspond*. This is because the re-use of a name may alter a bound variable's scope, as we will see.

Now we can introduce the idea of a variable being **bound** or **free** in an expression. A variable is said to be bound to occurrences in the body of a function for which it is the bound variable provided no other functions within the body

introduce the same bound variable. Otherwise it is said to be free.

Thus, in the expression:

```
λx.x
```

the variable `x` is bound but in the expression:

```
x
```

the variable `x` is free. In:

```
λf.(f λx.x)
```

the variable `f` is bound but in the expression:

```
(f λx.x)
```

the variable `f` is free.

In general, for a function:

```
λ<name>.<body>
```

`<name>` refers to the same variable throughout `<body>` except where another function has `<name>` as its bound variable. References to `<name>` in the new function's body then correspond to the new bound variable and not the old.

In formal terms, all the free occurrences of `<name>` in `<body>` are references to the same bound variable `<name>` introduced by the original function. `<name>` is in scope in `<body>` wherever it may occur free; that is except where another function introduces it in in a new scope.

For example, in the body of:

```
λf.(f λf.f)
```

which is:

```
(f λf.f)
```

the first `f` is free so it corresponds to the original bound variable `f` but subsequent `f`s are bound and so are distinct from the original bound variable. The outer `f` is in scope except in the scope of the inner `f.`

In the body of:

```
λg.((g λh.(h (g λh.(h λg.(h g))))) g)
```

which is:

```
(g λh.(h (g λh.(h λg.(h g))))) g
```

the first, second and last occurrences of `g` occur free so they correspond to the outer bound variable `g.` The third and fourth `g`s are bound and so are distinct from the original `g.` The outer `g` is in scope in the body except in the scope of the inner `g.`

Let us tighten up our definitions. A variable is bound in an expression if:

i) the expression is an application:

```
(<function> <argument>)
```

and the variable is bound in `<function>` or `<argument>`

For example, `convict` is bound in:

```
(λconvict.convict fugitive)
```

and in:

```
(λprison.prison λconvict.convict)
```

ii) the expression is a function:

```
λ<name>.<body>
```

and either the variable's name is `<name>` or it is bound in `<body>`.

For example, `prisoner` is bound in:

```
λprisoner.(number6 prisoner)
```

and in:

```
λprison.λprisoner.(prison prisoner)
```

Similarly, a variable is free in an expression if:

i) the expression is a single name:

```
<name>
```

and the variable's name is `<name>`

For example, `truant` is free in:

```
truant
```

ii) the expression is an application:

```
(<function> <argument>)
```

and the variable is free in `<function>` or in `<argument>`

For example, `escaper` is free in:

```
(λprisoner.prisoner escaper)
```

and in:

```
(escaper λjailor.jailor)
```

iii) the expression is a function:

```
λ<name>.<body>
```

and the variable's name is not `<name>` and the variable is free in `<body>`.

For example, `fugitive` is free in:

```
λprison.(prison fugitive)
```

and in:

```
λshort.λsharp.λshock.fugitive
```

Note that a variable may be bound and free in different places in the same expression.

We can now define β reduction more formally. In general, for the β reduction of an application:

```
(λ<name>.<body> <argument>)
```

we replace all free occurrences of `<name>` in `<body>` with `<argument>`. This ensures that only those occurrences of `<name>` which actually correspond to the bound variable are replaced.

For example, in:

```
(λf.(f λf.f) λs.(s s))
```

the first occurrence of `f` in the body:

```
(f λf.f)
```

is free so it gets replaced:

```
(λs.(s s) λf.f) =>

(λf.f λf.f) =>

λf.f
```

In subsequent examples we will use distinct bound variables.

2.14. Name clashes and α conversion

We have restricted the use of names in expressions to the bodies of functions. This may be restated as the requirement that there be no free variables in a λ expression. Without this restriction names become objects in their own right. This eases data representation: atomic objects may be represented directly as names and structured sequences of objects as nested applications using names. However, it also makes reduction much more complicated.

For example consider the function application function:

```
def apply = λfunc.λarg.(func arg)
```

Consider:

```
((apply arg) boing) ==

((λfunc.λarg.(func arg) arg) boing)
```

Here, `arg` is used both as a function bound variable name and as a free variable name in the leftmost application. These are two distinct uses: the bound variable will be replaced by β reduction but the free variable stays the same. However, if we carry out β reduction literally:

```
((λfunc.λarg.(func arg) arg) boing) =>

(λarg.(arg arg) boing) =>

(boing boing)
```

which was not intended at all. The argument `arg` has been substituted in the scope of the bound variable `arg` and appears to create a new occurrence of that bound variable.

We can avoid this using consistent renaming. Here we might replace the bound variable `arg` in the function with, say, `arg1`:

```
((λfunc.λarg1.(func arg1) arg) boing) =>

(λarg1.(arg arg1) boing) =>

(arg boing)
```

A name clash arises when a β reduction places an expression with a free variable in the scope of a bound variable with the same name as the free variable. Consistent renaming, which is known as **α conversion (alpha conversion)**, removes the name clash. For a function:

```
λ<name1>.<body>
```

the name `<name1>` and all free occurrences of `<name1>` in `<body>` may be replaced by a new name `<name2>` provided `<name2>` is not the name of a free variable in `λ<name1>.<body>`. Note that replacement includes the name at:

```
λ<name1>
```

In subsequent examples we will avoid name clashes.

2.15. Simplification through eta reduction

Consider an expression of the form:

```
λ<name>.(<expression> <name>)
```

This is a bit like the function application function above after application to a function expression only. This is equivalent to:

```
<expression>
```

because the application of this expression to an arbitrary argument:

```
<argument>
```

gives:

```
(λ<name>.(<expression> <name>) <argument>) =>

(<expression> <argument>)
```

This simplification of:

```
λ<name>.(<expression> <name>)
```

to:

```
<expression>
```

is called η *reduction (eta reduction)*. We will use it in later chapters to simplify expressions.

2.16. Summary

In this chapter we have:

- considered abstraction and its role in programming languages, and the λ calculus as a language based on pure abstraction
- met the λ calculus syntax and analysed the structure of some simple expressions
- met normal order β reduction and reduced some simple expressions, noting that not all reductions terminate
- introduced notations for defining functions and simplifying familiar reduction sequences
- seen that functions may be constructed from other functions
- met functions for constructing pairs of values and selecting from them
- formalised normal order β reduction in terms of substitution for free variables
- met α conversion as a way of removing name clashes in expressions
- met η reduction as a way of simplifying expressions

Some of these topics are summarised below.

Lambda calculus syntax

```
<expression> ::= <name> | <function> | <application>

<name> ::= non-blank character sequence

<function> ::= λ <name> . <body>
 <body> ::= <expression>

<application> ::= ( <function expression> <argument expression> )
 <function expression> ::= <expression>
 <argument expression> ::= <expression>
```

Free variables

i) `<name>` is free in `<name>`.

ii) `<name>` is free in λ`<name1>.<body>`
if `<name1>` is not `<name>`
and `<name>` is free in `<body>`.

iii) `<name>` is free in `(<function expression> <argument expression>)`
if `<name>` is free in `<function expression>`
or `<name>` is free in `<argument expression>`.

Bound variables

i) <name> is bound in λ<name1>.<body>
if <name> is <name1>
or <name> is bound in <body>.

ii) <name> is bound in (<function expression> <argument expression>)
if <name> is bound in <function expression>
or <name> is bound in <argument expression>.

Normal order β reduction

For (<function expression> <argument expression>)

i) normal order β reduce <function expression> to <function value>

ii) if <function value> is λ<name>.<body>
then replace all free occurences of <name> in <body> with <argument expression>
and normal order β reduce the new <body>

or

iii) if <function value> is not a function
then normal order β reduce <argument expression> to <argument value>
and return (<function value> <argument value>)

Normal order reduction notation

=> - normal order β reduction

=> ... => - multiple normal order β reduction

Definitions

```
def <name> = <expression>
```

Replace all subsequent occurences of <name> with <expression> before evaluation.

Replacement notation

== - defined name replacement

α conversion

To rename <name1> as <name2> in λ<name1>.<body>
if <name2> is not free in λ<name1>.<body>
then replace all free occurences of <name1> in <body> with <name2>
and replace <name1> in λ.<name1>

η reduction

```
(λ<name>.(<expression> <name>) <argument>) =>

<expression> <argument>
```

2.17. Exercises

1) Analyse each of the following lambda expressions to clarify its structure. If the expression is a function, identify the bound variable and the body expression, and then analyse the body expression. If the expression is an application, identify the function and argument expressions, and then analyse the function and argument expressions:

```
i)   λa.(a λb.(b a))
ii)  λx.λy.λz.((z x) (z y))
iii) (λf.λg.(λh.(g h) f) λp.λq.p)
iv)  λfee.λfi.λfo.λfum.(fum (fo (fi fee)))
v)   (λp.(λq.p λx.(x p)) λi.λj.(j i))
```

2) Evaluate the following lambda expressions:

```
i)   ((λx.λy.(y x) λp.λq.p) λi.i)
ii)  (((λx.λy.λz.((x y) z) λf.λa.(f a)) λi.i) λj.j)
iii) (λh.((λa.λf.(f a) h) h) λf.(f f))
iv)  ((λp.λq.(p q) (λx.x λa.λb.a)) λk.k)
v)   (((λf.λg.λx.(f (g x)) λs.(s s)) λa.λb.b) λx.λy.x)
```

3) For each of the following pairs, show that function `a)` is equivalent to the function resulting from expression `b)` by applying both to arbitrary arguments:

```
i)   a) identity
     b) (apply (apply identity))
ii)  a) apply
     b) λx.λy.(((make_pair x) y) identity)
iii) a) identity
     b) (self_apply (self_apply select_second))
```

4) Define a function:

```
def make_triplet = ...
```

which is like `make_pair` but constructs a triplet from a sequence of three arguments so that any one of the arguments may be selected by the subsequent application of a triplet to a selector function.

Define selector functions:

```
def triplet_first = ...
def triplet_second = ...
def triplet_third = ...
```

which will select the first, second or third item from a triplet respectively.

Show that:

```
make_triplet <item1> <item2> <item3> triplet_first => ... => <item1>
make_triplet <item1> <item2> <item3> triplet_second => ... => <item2>
make_triplet <item1> <item2> <item3> triplet_third => ... => <item3>
```

for the arbitrary arguments:

```
<item1> <item2> <item3>
```

5) Analyse each of the following lambda expressions to identify its free and bound variables, and those in its sub-expressions:

```
i)   λx.λy.(λx.y λy.x)
ii)  λx.(x (λy.(λx.x y) x))
iii) λa.(λb.a λb.(λa.a b))
iv)  (λfree.bound λbound.(λfree.free bound))
v)   λp.λq.(λr.(p (λq.(λp.(r q)))) (q p))
```

6) Remove any name clashes in the expressions in exercise 5 above.

3. CONDITIONS, BOOLEANS AND INTEGERS

3.1. Introduction

In this chapter we are going to start to add layers to the λ calculus to develop a higher level functional notation.

First of all we will use the pair functions from chapter 1 to represent conditional expressions with truth values `true` and `false.` We will then use these to develop boolean operations like `not, and` and `or.`

Next we will use the pair functions to represent natural numbers in terms of the value `zero` and the successor function.

Finally, we will introduce notations for simplifying function definitions and λ expressions, and for an 'if .. then ... else' form of conditional expression.

For the moment we will be looking at untyped representations of truth values and functions. We will develop typed representations in chapter 5.

3.2. Truth values and conditional expression

Boolean logic is based on the truth values `TRUE` and `FALSE` with logical operations `NOT, AND, OR` and so on.

We are going to represent `TRUE` by `select_first` and `FALSE` by `select_second`, and use a version of `make_pair` to build logical operations. To motivate this, consider the C conditional expression:

```
<condition>?<expression>:<expression>
```

If the `<condition>` is `TRUE` then the first `<expression>` is selected for evaluation and if the `<condition>` is `FALSE` then the second `<expression>` is selected for evaluation.

For example, to set `max` to the greater of `x` and `y:`

```
max = x>y?x:y
```

or to set `absx` to the absolute value of `x:`

```
absx = x<0?-x:x
```

We can model a conditional expression using a version of the make pair function:

```
def cond = λe1.λe2.λc.((c e1) e2)
```

Consider `cond` applied to the arbitrary expressions `<expression1>` and `<expression2>`:

```
((cond <expression1>) <expression2>) ==
```