```
i)  fun_sum_step double five two
ii) fun_sum_step double four two
```

5) Define functions to test whether or not a number is less than, or less than or equal to another number:

```
def less x y = ...

def less_or_equal x y = ...
```

Evaluate:

```
i)   less three two
ii)  less two three
iii) less two two
iv)  less_or_equal three two
v)   less_or_equal two three
vi)  less_or_equal two two
```

6) Define a function to find the remainder on dividing one number by another:

```
def mod x y = ...
```

Evaluate:

```
i)   mod three two
ii)  mod two three
iii) mod three zero
```

5. TYPES

5.1. Introduction

In this chapter we are going to consider how types can be added to our functional notation to ensure that only meaningful arguments are passed to functions.

To begin with, we will consider the role of types in programming in general and how types may be characterised. We will then introduce functions for constructing and manipulating typed values, using the pair manipulation functions to represent typed objects as type/value pairs.

Next, we will introduce the error type for error objects which are returned after type errors. We will then develop typed representations for booleans, numbers and characters.

Finally, we will introduce new notations to simplify function definitions through case definitions and structure matching.

5.2. Types and programming

We are working with a very simple language. As we exclude single names as expressions, the only objects are functions which take function arguments and return function results. (For the moment, we won’t consider non-terminating applications.) We have constructed functions which we can interpret as boolean values, boolean operations, numbers, arithmetic operations and so on but particular functions have no intrinsic interpretations other than in terms of their effects on other functions. Because functions are so general, there is no way to restrict the application of functions to specific other functions, for example we cannot restrict ‘arithmetic’ functions to ‘numeric’ operands. We can carry out function applications which are perfectly valid but have results with no relevant meaning within our intended interpretations.

For example, consider the effect of:

```
iszero true ==

λn.(n select_first) true =>

true select_first ==

λfirst.λsecond.first select_first =>

λsecond.select_first ==

λsecond.λfirst.λsecond.first
```

This produces a perfectly good function for selecting the second argument in an application with three nested arguments but we expect `iszero` to return a boolean.

Using these functions is analogous to programming in machine code. In most CPUs, the sole objects are undifferentiated bit patterns which have no intrinsic meanings but are interpreted in different ways by the machine code operations applied to them. For example, different machine code instructions may treat a bit pattern as a signed or unsigned integer, decimal or floating point number or a data or instruction address. Thus, a machine code program may twos-complement an address or jump to a floating point number.

The single typed systems programming language BCPL, a precursor of C, is similarly free and easy. Although representations are provided for a variety of objects, operations are used without type checks on the assumption that their operands are appropriate. Early versions of C provided type checking but were somewhat lax when operations were carried out on almost appropriate types, allowing indirection on integers or arithmetic on pointers, for example.

It is claimed that this ‘freedom’ from types makes languages more flexible. It does ease implementation dependent programming where advantage is taken of particular architectural features on particular CPUs through bit or word level manipulation but this in turn leads to a loss of portability because of gross incompatibilities between architectures. For example, many computer memories are based on 8 bit bytes so 16 bit words require 2 bytes. However, computers differ in the order in which these bytes are used: some put the top 8 bits in the first byte but others put them in the second byte. Thus, programs using ‘clever’ address arithmetic which involves knowing the byte order won’t work on some computers. ‘Type free’ programming also increases incomprehensible errors through dodgy low-level subterfuges. For example, ‘cunning’ address manipulations to access the fields of a data structure may cause the corruption of other fields or of a completely different data structure which is close to the requisite one in memory, through byte mis-alignments.

5.3. Type as objects and operations

Types are introduced into languages to control the use of operations on objects so as to ensure that only meaningful combinations are used. As we saw in chapter 2, variables in programming languages are used as a primary abstraction mechanism. In ‘typeless’ languages there are no restrictions on object/operation combinations and any variable may be associated with any object. Here, variables just abstract over objects in general. In weakly typed languages, like LISP and Prolog, objects are typed but variables are not. There are restrictions on object/operation combinations but not on variable/object associations. Thus, variables do not abstract over specific types. In strongly typed languages like ML and Pascal variables are specified as being of a specific type and have the same restrictions on use as objects of that type.

More formally, a type specifies a class of objects and associated operations. Object classes may be defined by listing their values, for example for booleans:

```
TRUE is a boolean
FALSE is a boolean
```

or by specifying a base object and a means of constructing new objects from the base, for example for natural numbers:

```
0 is a number

SUCC N is a number
if N is a number
```

Thus, we can show that:

```
SUCC (SUCC (SUCC 0))
```

is a number because:

```
0

SUCC 0

SUCC (SUCC 0)
```

are all numbers.

Operations may be specified exhaustively with a case for each base value, for example for booleans, negation:

```
NOT TRUE = FALSE
NOT FALSE = TRUE
```

and conjunction:

```
AND FALSE FALSE = FALSE
AND FALSE TRUE = FALSE
AND TRUE FALSE = FALSE
AND TRUE TRUE = TRUE
```

Operations may also be specified constructively in terms of base cases for the base objects and general cases for the constructive objects. For example for natural numbers, the predecessor function:

```
PRED 0 = 0
PRED (SUCC X) = X
```

and addition:

```
ADD X 0 = X
ADD X (SUCC Y) = ADD (SUCC X)  Y
```

and subtraction:

```
SUB X 0 = X
SUB (SUCC X) (SUCC Y) = SUB X Y
```

and multiplication:

```
MULT X 0 = 0
MULT X (SUCC Y) = ADD X (MULT X Y)
```

Note that we have just formalised the informal descriptions from the last chapter. We will look at the relationship between exhaustive and case definitions, and our conditional style of definition later in this chapter.

Sometimes it may be necessary to introduce conditions into case definitions because the form of the object definition may not provide enough information to discriminate between cases. For example for division:

```
DIV X 0 = NUMBER ERROR
DIV X Y = 0
          if (GREATER Y X)
DIV X Y = SUCC (DIV (SUB X Y) Y)
          if NOT (GREATER Y X)
```

the `if`s are needed because the **values** rather than the **structures** of `X` and `Y` determine how they are to be processed.

Operations may map objects of a type to objects of the same type or to objects of another type. A common requirement is for **predicates** which are used to test properties of objects and return booleans. For example for numbers:

```
EQUAL 0 0 = TRUE
EQUAL (SUCC X) 0 = FALSE
EQUAL 0 (SUCC X) = FALSE
EQUAL (SUCC X) (SUCC Y) = EQUAL X Y
```

We are not going to give a full formal treatment of types here.

5.4. Representing typed objects

We are going to construct functions to represent typed objects. In general, an object will have a type and a value. We need to be able to:

i) construct an object from a value and a type

ii) select the value and type from an object

iii) test the type of an object

iv) handle type errors

We will represent an object as a type/value pair:

```
def make_obj type value = λs.(s type value)
```

For an arbitrary object of type:

```
<type>
```

and value:

```
<value>
```

is represented by:

```
make_obj <type> <value> => ... =>

λs.(s <type> <value>
```

Thus, the type is selected with `select_first`:

```
def type obj = obj select_first
```

and the value is selected with `select_second`:

```
def value obj = obj select_second
```

We will use numbers to represent types and numeric comparison to test the type:

```
def istype t obj = equal (type obj) t
```

We are going to define typed objects and operations in terms of untyped objects and operations. In general, our approach will be:

i) check argument types

ii) extract untyped values from typed arguments

iii) carry out untyped operations on untyped values

iv) construct typed result form untyped result

We must distinguish definitions from the subsequent uses of typed objects. When defining types we cannot avoid using untyped operations: we have to start somewhere. Once types are defined, however, we should only manipulate typed objects with typed operations to ensure that the type checks aren't overridden.

In general, we will use `UPPER CASE LETTERS` for typed constructs and `lower case letters` for untyped constructs.

We should show that our representation of a type satisfies the formal definition of that type but we won't do this rigorously or religiously.

5.5. Errors

Whenever we detect a type error we will return an appropriate error object. Such an object will have type `error_type`, represented as `zero:`

```
def error_type = zero
```

We need a function to construct error objects:

```
def MAKE_ERROR = make_obj error_type
```

This definition expands as:

```
make_obj error_type ==

λtype.λvalue.λs.(s type value) error_type =>

λvalue.λs.(s error_type value)
```

An error object's value should indicate the sort of error the object represents. For example, for a type error the corresponding error object might have the expected type as value.

We will define a universal error object of type `error_type`:

```
def ERROR = MAKE_ERROR error_type
```

This definition expands as:

```
    λvalue.λs.(s error_type value) error_type =>

    λs.(s error_type error_type)
```

so the error object `ERROR` has type `error_type` and value `error_type`.

We can test for an object of type error by using `istype` to look for `error_type`:

```
    def iserror = istype error_type
```

so `iserror`'s definition expands as:

```
    istype error_type ==

    λt.λobj.(equal (type obj) t) error_type =>

    λobj.(equal (type obj) error_type)
```

For example, to test that `ERROR` is of type error:

```
    iserror ERROR ==

    λobj.(equal (type obj) error_type) ERROR =>

    equal (type ERROR) error_type
```

Now:

```
    type ERROR
```

expands as:

```
    λobj.(obj select_first) ERROR =>

    ERROR select_first ==

    λs.(s error_type error_type) select_first =>

    select_first error_type error_type => ... =>

    errortype
```

Thus:

```
    equal (type ERROR) error_type -> ... ->

    equal error_type error_type => ... =>

    true
```

Our formal type definitions should be extended to show how error objects are accommodated. We won't do so rigorously. In general, if an operation expects an argument of one type and does not receive one then it will return an error object corresponding to the required type. Thus if an operation is passed an error object as the result of a previous operation then the error object will not be of the required type and a new error object will be returned.

5.6. Booleans

We will represent the boolean type as `one`:

```
def bool_type = one
```

Constructing a boolean type involves preceding a boolean value with `bool_type`:

```
def MAKE_BOOL = make_obj bool_type
```

which expands as:

```
λvalue.λs.(s bool_type value)
```

We can now construct the typed booleans `TRUE` and `FALSE` from the untyped versions by:

```
def TRUE = MAKE_BOOL true
```

which expands as:

```
λs.(s bool_type true)
```

and:

```
def FALSE = MAKE_BOOL false
```

which expands as:

```
λs.(s bool_type false)
```

As with the error type, the test for a boolean type involves checking for `bool_type`:

```
def isbool = istype bool_type
```

This definition expands as:

```
λobj.(equal (type obj) bool_type)
```

A boolean error object will be an error object with type `bool_type`:

```
def BOOL_ERROR = MAKE_ERROR bool_type
```

which expands as:

```
λs.(s error_type bool_type)
```

The typed function `NOT` should either return an error if the argument is not a boolean or extract the value from the argument, use untyped `not` to complement it and make a new boolean from the result:

```
def NOT X =
 if isbool X
 then MAKE_BOOL (not (value X))
 else BOOL_ERROR
```

Similarly, the typed function `AND` should either return an error if either argument is non boolean or make a new boolean from ‘and’ing the values of the arguments:

```
def AND X Y =
 if and (isbool X) (isbool Y)
 then MAKE_BOOL (and (value X) (value Y))
 else BOOL_ERROR
```

We will now consider how these definitions bolt together by looking at:

```
AND TRUE FALSE
```

After definition replacement and initial bound variable substitution we have:

```
 if and (isbool TRUE) (isbool FALSE)
 then MAKE_BOOL (and (value TRUE) (value FALSE))
 else BOOL_ERROR
```

First of all:

```
isbool TRUE ==

λobj.(equal (type obj) bool_type) TRUE =>

equal (type TRUE) bool_type ==

equal (λobj.(obj select_first) TRUE) bool_type ->

equal (TRUE select_first) bool_type ==

equal (λs.(s bool_type true) select_first) bool_type -> ... ->

equal bool_type bool_type => ... =>

true
```

Similarly:

```
isbool FALSE => ... =>

true
```

Thus:

```
and (isbool TRUE) (isbool FALSE) -> ... ->

and true (isbool FALSE) -> ... ->

and true true => ... =>

true
```

We now evaluate:

```
MAKE_BOOL (and (value TRUE) (value FALSE))
```

For the `and`:

```
value TRUE ==

λobj.(obj select_second) TRUE =>
```

```
TRUE select_second ==

λs.(s bool_type true) select_second => ... =>

true
```

and:

```
value FALSE => ... =>

false
```

so:

```
MAKE_BOOL (and (value TRUE) (value FALSE)) ->

MAKE_BOOL (and true (value FALSE)) ->

MAKE_BOOL (and true false) ->

MAKE_BOOL false ==

λvalue.λs.(s bool_type value) false =>

λs.(s bool_type false) ==

FALSE
```

5.7. Typed conditional expression

We need a typed conditional expression to handle both typed booleans and type errors in a condition:

```
def COND E1 E2 C =
 if isbool C
 then
  if value C
  then E1
  else E2
 else BOOL_ERROR
```

Note that this typed conditional function will return `BOOL_ERROR` if the condition is not a boolean.

We will now write:

```
IF <condition>
THEN <expression1>
ELSE <expression2>
```

instead of:

```
COND <expression1> <expression2> <condition>
```

We also need typed versions of the type testers for use with `IF` because `iserror` and `isbool` return the untyped `true` or `false` instead of the typed `TRUE` or `FALSE`:

```
def ISERROR E = MAKE_BOOL (iserror E)
```

```
    def ISBOOL B = MAKE_BOOL (isbool B)
```

5.8. Numbers and arithmetic

We will represent the number type as `two:`

```
    def numb_type = two
```

and a number will be a pair starting with `numb_type`:

```
    def MAKE_NUMB = make_obj numb_type
```

`MAKE_NUMB`s definition expands to:

```
    λvalue.λs.(s numb_type value)
```

We need an error object for arithmetic type errors:

```
    def NUMB_ERROR = MAKE_ERROR numb_type
```

which expands to:

```
    λs.(s error_type numb_type)
```

We also need a type tester:

```
    def isnumb = istype numb_type
```

which expands to:

```
    λobj.(equal (type obj) numb_type)
```

from which we can define a typed type tester:

```
    def ISNUMB N = MAKE_BOOL (isnumb N)
```

Next we can construct a typed `zero:`

```
    def 0 = MAKE_NUMB zero
```

which expands as:

```
    λs.(s numb_type zero)
```

We now need a typed successor function:

```
    def SUCC N =
     if isnumb N
     then MAKE_NUMB (succ (value N))
     else NUMB_ERROR
```

to define numbers:

```
    def 1 = SUCC 0

    def 2 = SUCC 1
```

```
def 3 = SUCC 2
etc.
```

For example, `1` expands as:

```
SUCC 0 => ... =>

if isnumb 0
then MAKE_NUMB (succ (value N))
else NUMB_ERROR
```

First of all:

```
isnumb 0 ==

equal (type 0) numb_type ==

equal (λobj.(obj select_first) 0) numb_type =>

equal (0 select_first) numb_type ==

equal (λs.(s numb_type zero) select_first) numb_type -> ... ->

equal numb_type numb_type => ... =>

true
```

Thus, we next evaluate:

```
MAKE_NUMB (succ (value 0)) ==

MAKE_NUMB (succ (λobj.(obj select_second) 0)) ->

MAKE_NUMB (succ (0 select_second)) ==

MAKE_NUMB (succ (λs.(s numb_type zero) select_second) -> ... ->

MAKE_NUMB (succ zero) ==

MAKE_NUMB one ==

λvalue.λs.(s numb_type value) one =>

λs.(s numb_type one)
```

In general, a typed number is a pair with the untyped equivalent as value.

We can now redefine the predecessor function to return an error for zero:

```
def PRED N =
 if isnumb N
 then
  if iszero (value N)
  then NUMB_ERROR
  else MAKE_NUMB ((value N) select_second)
 else NUMB_ERROR
```

Note that we return `NUMB_ERROR` for a non-number argument and a zero number argument. We could construct more elaborate error objects to distinguish such cases but we won’t pursue this further here.

We will need a typed test for zero:

```
def ISZERO N =
 if isnumb N
 then MAKE_BOOL (iszero (value N))
 else NUMB_ERROR
```

Now we can redefine the binary arithmetic operations. They all need to test that both arguments are numbers so we will introduce an auxiliary function to do this:

```
def both_numbs X Y = and (isnumb X) (isnumb Y)
```

Now for addition based on our earlier definition:

```
def + X Y =
 if both_numbs X Y
 then MAKE_NUMB (add (value X) (value Y))
 else NUMB_ERROR
```

and multiplication:

```
def * X Y =
 if both_numbs X Y
 then MAKE_NUMB (mult (value X) (value Y))
 else NUMB_ERROR
```

and division to take account of a zero divisor:

```
def / X Y =
 if both_numbs X Y
 then
  if iszero (value Y)
  then NUMB_ERROR
  else MAKE_NUMB (div1 (value X) (value Y))
 else NUMB_ERROR
```

and equality:

```
def EQUAL X Y =
 if both_numbs X Y
 then MAKE_BOOL (equal (value X) (value Y))
 else NUMB_ERROR
```

5.9. Characters

Let us now add characters to our types. Character values are specified exhaustively:

```
’0’ is a character
’1’ is a character
...
’9’ is a character

’A’ is a character
’B’ is a character
```

```
...
'Z' is a character

'a' is a character
'c' is a character
...
'z' is a character

'.' is a character
',' is a character
...
```

It is useful to have orderings on sub-sequences of characters for lexicographic purposes:

```
'0' < '1'
'1' < '2'
...
'8' < '9'

'A' < 'B'
'B' < 'C'
...
'Y' < 'Z'

'a' < 'b'
'b' < 'c'
...
'y' < 'z'
```

where the ordering relation has the usual transitive property:

```
X < Z
if X < Y and Y < Z
```

It simplifies character manipulation if there is a uniform ordering overall. For example, in the ASCII character set:

```
'9' < 'A'
'Z' < 'a'
```

and most punctuation marks appear before `'0'` in the ordering.

We will introduce a new type for characters:

```
def char_type = four

def CHAR_ERROR = MAKE_ERROR char_type

def ischar = istype char_type

def ISCHAR C = MAKE_BOOL (ischar C)

def MAKE_CHAR = make_obj char_type
```

A character object will have type `char_type`. To provide the ordering, characters will be mapped onto the natural numbers so the value of a character will be an untyped number. We will use the ASCII values:

```
def '0' = MAKE_CHAR forty_eight
def ’1’ = MAKE_CHAR (succ (value ’0’))

 ...

def ’9’ = MAKE_CHAR (succ (value ’8’))

def ’A’ = MAKE_CHAR sixty_five

def ’B’ = MAKE_CHAR (succ (value ’A’))

 ...

def ’Z’ = MAKE_CHAR (succ (value ’Y’))

def ’a’ = MAKE_CHAR ninety_seven

def ’b’ = MAKE_CHAR (succ (value ’a’))

 ...

def ’z’ = MAKE_CHAR (succ (value ’y’))
```

Now we can define character ordering:

```
def CHAR_LESS C1 C2 =
 if and (ischar C1) (ischar C2)
 then MAKE_BOOL (less (value C1) (value C2))
 else CHAR_ERROR
```

and conversion from character to number:

```
def ORD C =
 if ischar C
 then MAKE_NUMB (value C)
 else CHAR_ERROR
```

and vice-versa:

```
def CHAR N =
 if isnumb N
 then MAKE_CHAR (value N)
 else NUMB_ERROR
```

For example, to find `’A’`s numeric equivalent:

```
ORD ’A’ => ... =>

MAKE_NUMB (value ’A’) ==

MAKE_NUMB (value λs.(s char_type sixty_five)) -> ... ->

MAKE_NUMB sixty_five => ... =>

λs.(numb_type sixty_five) ==

65
```

Similarly, to construct a character from the number `98`:

```
CHAR 98 => ... =>

MAKE_CHAR (value 98) ==

MAKE_CHAR (value λs.(s numb_type ninety_eight)) -> ... ->

MAKE_CHAR ninety_eight => ... =>

λs.(char_type ninety_eight) ==

'b'
```

Because we have used numbers as character values we can base character comparison on number comparison:

```
def CHAR_EQUAL C1 C2 =
 if and (ischar C1) (ischar C2)
 then MAKE_BOOL (equal (value C1) (value C2))
 else CHAR_ERROR
```

5.10. Repetitive type checking

Once we have defined typed `TRUE`, `FALSE`, `ISBOOL` and `IF` we could define typed versions of all the other boolean operations from them, for example:

```
def NOT X =
 IF X
 THEN FALSE
 ELSE TRUE

def AND X Y =
 IF ISBOOL Y
 THEN
  IF X
  THEN Y
  ELSE FALSE
 ELSE BOOL_ERROR
```

Note that for `NOT` we do not need to check explicitly that `X` is a boolean because the `IF` does so anyway. In the same way in `AND` we do not need to check that `X` is a boolean as the second `IF` will.

With typed boolean operations and having defined typed `0`, `SUCC`, `PRED`, `ISNUMB` and `ISZERO`, we could define the other arithmetic operations using only typed operations, for example:

```
def ADD X Y =
 IF AND (ISNUMB X) (ISNUMB Y)
 THEN AND1 X Y
 ELSE NUMB_ERROR

rec ADD1 X Y =
 IF ISZERO Y
 THEN X
 ELSE ADD1 (SUCC X) (PRED Y)
```

Here we have defined an outer non-recursive function to check the arguments and an auxiliary recursive function to carry out the operation without argument checks. We could avoid the explicit check that `Y` is a number as `ISZERO` will do so. However, for a non numeric argument, `ISZERO` (and thence the two `IF`s) will return a `BOOL_ERROR` instead of a `NUMB_ERROR`.

As definitions these seem satisfactory but they would be appallingly inefficient if used as the basis of an implementation because of repetitive type checking. Consider, for example(*):

```
ADD 1 2
```

First of all in:

```
IF AND (ISNUMB 1) (ISNUMB 2)
```

both:

```
ISNUMB 1
```

and:

```
ISNUMB 2
```

are checked and return booleans. Next:

```
AND (ISNUMB 1) (ISNUMB 2)
```

checks that both `ISNUMB`s return booleans and then itself returns a boolean. Then:

```
IF AND (ISNUMB 1) (ISNUMB 2)
```

checks that `AND` returns a boolean.

Secondly, after `ADD1` is called:

```
IF ISZERO 2
```

calls:

```
ISZERO 2
```

to check that `2` is a number and return a boolean. Next:

```
IF ISZERO 2
```

checks that `ISZERO` returns a boolean.

Now, `ADD1` is called recursively through:

```
ADD1 (SUCC 1) (PRED 2)
```

so:

```
IF ISZERO (PRED 2)
```

calls:

(*) This example also highlights repetitive argument evaluation due to naive normal order evaluation. We will consider different approaches to argument evaluation in chapter 8.

```
ISZERO (PRED 2)
```

to check that `(PRED`2) is a number and return a boolean and then:

```
IF ISZERO (PRED 2)
```

checks that `ISZERO` returns a boolean.

Once again, `ADD1` is called recursively in:

```
ADD1 (SUCC (SUCC 1)) (PRED (PRED 2))
```

and evaluation, and type checking, continue.

5.11. Static and dynamic type checking

Clearly, there is a great deal of unnecessary type checking here. Arguably, we 'know' that the function types match so we only need to test the outer level arguments once. This is the approach we have used above in defining typed operations where the arguments are checked before untyped operations are carried out. But how do we 'know' that the types match up? It is all very well to claim that we are using types consistently in relatively small definitions but in developing large expressions, type mismatches will inevitably slip through, just as they do during the development of large programs. The whole point of types was to detect such inconsistencies.

As we saw above, using untyped functions is analogous to programming in a language without typed variables or type checking like machine code or BCPL. Similarly, using our fully typed functions is analogous to programming in a language where variables are untyped but objects are checked dynamically by each operation while a program runs as in Prolog and LISP.

The alternative is to introduce types into the syntax of the language and then check type consistency symbolically before running programs. With symbolic checking, run time checks are redundant. Typing may be made explicit with typed declarations, as in C and Pascal, or deduced from variable and operation use, as in ML and PS-algol, though in the last two languages types may and sometimes must be specified explicitly as well.

There are well developed theories of types which are used to define and manipulate typed objects and typed languages. Some languages provide for user defined types in a form based on such theories. For example, ML and Miranda provide for user defined types through abstract data types which, in effect, allow functional abstraction over type definitions. We won't consider these further. We will stick with defining 'basic' typed functions from untyped functions and subsequently using the typed functions. What constitutes a 'basic' function will be as much a matter of expediency as theory! For pure typed functions though, the excessive type checking will remain.

5.12. Infix operators

In our notation the function always precedes the arguments in a function application. This is known as **prefix** notation and is used in `LISP.` Most programming languages follow traditional logic and arithmetic and allow **infix** notation as well for some binary function names. These may appear between their arguments.

We will now allow the names for logical and arithmetic functions to be used as infix operators so, for example:

```
<expression1> AND <expression2> == AND <expression1> <expression2>
<expression1> OR <expression2> == OR <expression1> <expression2>
<expression1> + <expression2> == + <expression1> <expression2>
<expression1> - <expression2> == - <expression1> <expression2>
<expression1> * <expression2> == * <expression1> <expression2>
<expression1> / <expression2> == / <expression1> <expression2>
```

To simplify the presentation we won't introduce operator precedence or implicit associativity. Thus, strict bracketing is still required for function application arguments. For example, we write:

```
    7 + (8 * 9) == + 7 (8 * 9) == + 7 (* 8 9)
```

rather than the ambiguous:

```
    7 + 8 * 9
```

Some languages allow the programmer to introduce new infix binary operators with specified precedence and associativity. These include Algol 68, ML, POP-2 and Prolog.

5.13. Case definitions and structure matching

In this chapter we have introduced formal definitions based on the structure of the type involved. Thus, booleans are defined by listing the values `TRUE` and `FALSE` so boolean function definitions have explicit cases for combinations of `TRUE` and `FALSE.` Numbers are defined in terms of `0` and the application of the successor function `SUCC.` Thus, numeric functions have base cases for `0` and recursion cases for non-zero numbers.

In general, for multi-case definitions we have written:

```
    <name> <names1> = <expression1>
    <name> <names2> = <expression2>
     ...
```

where `<names>` is a structured sequence of bound variables, constants and constructors. When a function with a multi-case definition is applied to an argument, the argument is **matched** against the structured bound variable, constant and constructor sequences to determine which case applies. When a match succeeds for a particular case, then that case's bound variables are associated with the corresponding argument sub-structures for subsequent use in the case's right hand side expression. This is known as **structure matching**.

In our functional notation, however, we have to use conditional expressions explicitly to determine the structure of an object and hence which case of a definition should be used to process it. We then use explicit selection of sub-structures from structured arguments.

Some languages allow the direct use of case definitions and structure matching, for example Prolog, ML and Miranda. We will extend our notation in a similar manner so a function may now take the form:

```
    λ<names1>.<expression1>
     or <names2>.<expression2>
     or ...
```

and a definition simplifies to:

```
    def <name> <names1> = <expression1>
     or <name> <names2> = <expression2>
     or ...
```

For recursive functions, `rec` is used in place of `def.`

Note that the effect of matching depends on the order in which the cases are tried. Here, we will match cases from first to last in order. We also assume that each case is distinct from the others so at most only one match will succeed.

When a case defined function is applied to an argument, if the argument matches `<names1>` then the result is `<expression1>`; if the argument matches `<names2>` then the result is `<expression2>` and so on.

For boolean functions, we will allow the use of the constants `TRUE` and `FALSE` in place of bound variables. In general, for:

```
def <name> <bound variable> =
 IF <bound variable>
 THEN <expression1>
 ELSE <expression2>
```

we will now write:

```
def <name> TRUE = <expression1>
 or <name> FALSE = <expression2>
```

Thus, negation is defined by:

```
NOT TRUE = FALSE
NOT FALSE = TRUE
```

but written as:

```
def NOT X =
 IF X
 THEN FALSE
 ELSE TRUE
```

We will now write:

```
def NOT TRUE = FALSE
 or NOT FALSE = TRUE
```

Similarly, implication is defined by:

```
IMPLIES TRUE Y = Y
IMPLIES FALSE Y = TRUE
```

but written as:

```
def IMPLIES X Y =
 IF X
 THEN Y
 ELSE TRUE
```

We will now write:

```
def IMPLIES TRUE Y = Y
 or IMPLIES FALSE Y = TRUE
```

For numbers, we will allow the use of the constant `0` and bound variables qualified by nested `SUCC`s in place of bound variables. In general, for:

```
rec <name> <bound variable> =
 IF ISZERO <bound variable>
 THEN <expression1>
 ELSE <expression2 using (PRED <bound variable>)>
```

we will now write:

```
rec <name> 0 = <expression1>
 or <name> (SUCC <bound variable>) = <expression2 using <bound variable>>
```

Thus, the predecessor function is defined by:

```
PRED 0 = 0
PRED (SUCC X) = X
```

but written as:

```
def PRED X =
 IF ISZERO X
 THEN 0
 ELSE MAKE_NUMB (pred (value X))
```

We will now write:

```
def PRED 0 = 0
 or PRED (SUCC X) = X
```

Similarly the power function is defined by:

```
POWER X 0 = 1
POWER X (SUCC Y) = X*(POWER X Y)
```

but written as:

```
rec POWER X Y =
 IF ISZERO Y
 THEN 1
 ELSE X*(POWER X (PRED Y))
```

We will now write:

```
rec POWER X 0 = 1
 or POWER X (SUCC Y) = X*(POWER X Y)
```

5.14. Summary

In this chapter we have:

- considered the role of types in programming
- considered informally types as operations on objects
- introduced a representation for typed objects using type/value pairs
- developed an error type
- developed a boolean type with typed boolean operations
- developed typed conditional expressions and an ‘IF ... THEN ... ELSE ...’ notation for them
- developed a number type with typed numeric operations
- developed a character type with typed character operations

- considered repetitive type checking, and static and dynamic typing
- introduced notation for strictly bracketed infix expressions
- introduced notation for case definitions and structure matching

Some of these topics are summarised below.

IF ... THEN ... ELSE ...

```
IF <condition>
THEN <expression1>
ELSE <expression2> ==

COND <expression1> <expression2> <condition>
```

Infix operators

```
<expression1> <operator> <expresion2> ==

<operator> <expression1> <expression2>
```

Note that strict bracketing is required for nested infix expressions.

Boolean case definition

```
def <name> TRUE = <expression1>
 or <name> FALSE = <expression2> ==

def <name> <bound variable> =
 IF <bound variable>
 THEN <expression1>
 ELSE <expression2>
```

Number case definition

```
rec <name> 0 = <expression1>
 or <name> (SUCC <bound variable>) =
      <expression2 using ‘<bound variable>’> ==

rec <name> <bound variable> =
 IF ISZERO <bound variable>
 THEN <expression1>
 ELSE <expression2 using ‘PRED <bound variable>’>
```

5.15. Exercises

1) Evaluate fully the following expressions:

```
i)   ISBOOL 3
ii)  ISNUMB FALSE
iii) NOT 1
iv)  TRUE AND 2
v)   2 + TRUE
```

2) Signed numbers might be introduced as a new type with an extra layer of ‘pairing’ so that a numbers’s value is preceded by a boolean to indicate whether or not the number is positive or negative:

```
def signed_type = ...
def SIGN_ERROR = MAKE_ERROR signed_type
def POS = TRUE
def NEG = FALSE
def MAKE_SIGNED N SIGN = make_obj signed_type (make_obj SIGN N)
```

So:

```
+<number> == MAKE_SIGNED <number> POS
-<number> == MAKE_SIGNED <number> NEG
```

For example:

```
+4 == MAKE_SIGNED 4 POS
-4 == MAKE_SIGNED 4 NEG
```

Note that there are two representations for 0:

```
+0 == MAKE_SIGNED 0 POS
-0 == MAKE_SIGNED 0 NEG
```

i) Define tester and selector functions for signed numbers:

```
def issigned N = ...       - true if N is a signed number
def ISSIGNED N = ...       - TRUE if N is a signed number
def sign N = ...           - N’s sign as an untyped number
def SIGN N = ...           - N’s sign as a typed number
def sign_value N = ...     - N’s value as an unsigned number
def VALUE N = ...          - N’s value as a signed number
def sign_iszero N = ...    - true if N is 0
```

Show that your functions work for representative positive and negative values, and 0.

ii) Define signed versions of `ISZERO`, `SUCC` and `PRED:`

```
def SIGN_ISZERO N = ...
def SIGN_SUCC N = ...
def SIGN_PRED N = ...
```

Show that your functions work for representative positive and negative values, and 0.

iii) Define a signed versions of ‘+’:

```
def SIGN_+ X Y = ...
```

Show that your function works for representative positive and negative values, and 0.

6. LISTS AND STRINGS

6.1. Introduction

In this chapter we are going to look at the list data structure which is used to hold variable length sequences of values.

To begin with, we will discuss list construction and list element access. We will then use pair functions and the type representation techniques to add lists to our notation.