```
<expression> ::= (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 <number>
```

might be represented by a nested list structure so:

```
(<expression1> + <expression2>) == [<expression1>,"+",<expression2>]
(<expression1> - <expression2>) == [<expression1>,"-",<expression2>]
(<expression1> * <expression2>) == [<expression1>,"*",<expression2>]
(<expression1> / <expression2>) == [<expression1>,"/",<expression2>]
<number> == <number>
```

For example:

```
3 == 3
(3 * 4) == [3,"*",4]
((3 * 4) - 5) == [[3,"*",4],"-",5]
((3 * 4) - (5 + 6)) == [[3,"*",4],"-",[5,"+",6]]
```

Write a function which evaluates a nested list representation of an arithmetic expression. For example:

```
EVAL 3 => ... => 3
EVAL [3,"*",4] => ... => 12
EVAL [[3,"*",4],"-",5] => ... => 7
EVAL [[3,"*",4],"-",[5,"+",6]] => ... => 11
```

8. EVALUATION

8.1. Introduction

In this chapter we are going to look at evaluation order in more detail.

First of all we will consider the relative merits of applicative and normal order evaluation and see that applicative order is generally more efficient than normal order. We will also see that applicative order evaluation may lead to non-terminating evaluation sequences where normal order evaluation terminates, with our representations of conditional expressions and recursion.

We will then see that the halting problem is undecideable in undecidable so it is not possible to tell whether or not the evaluation of an arbitrary λ expression terminates. We will also survey the Church-Rosser theorems which show that normal and applicative order evaluation order are equivalent but that normal order is more likely to terminate.

Finally, we will look at lazy evaluation which combines the best features of normal and applicative orders.

8.2. Termination and normal form

A lambda expression which cannot be reduced any further is said to be in **normal form**. Our definition of β reduction in chapter 2 implied that evaluation of an expression terminates when it is no longer a function application. This won't reduce an expression to normal form. Technically we should go on evaluating the function body until it contains no more function applications. Otherwise, expressions which actually reduce to the same normal form appear to be different. For example, consider:

```
λx.x λa.(a a)
```

and:

```
λf.λa.(f a) λs.(s s)
```

The first reduces as:

```
λx.x λa.(a a) =>

λa.(a a)
```

and the second as:

```
λf.λa.(f a) λs.(s s) =>

λa.(λs.(s s) a)
```

Evaluation using our definition of β reduction terminates with two different final forms. If, however, we continue to evaluate the body of the second:

```
λa.(λs.(s s) a) =>

λa.(a a)
```

we can see that they are actually identical.

To be more formal, a reducible function application expression is called a **redex**. An expression is in normal forms when it contains no more redexes.

We will still tend to stop evaluation when we reach a recognisable function or function application. Thus, for lists we will continue to leave:

```
<value1>::<value2>
```

as it is instead of translating to the function application:

```
CONS <value1> <value2>
```

and continuing with evaluation.

8.3. Normal order

Normal order β reduction requires the evaluation of the leftmost redex in a expression. For a function application, this will evaluate the function expression and then carry out the substitution of the unevaluated argument. Normal order evaluation has the effect of delaying the evaluation of applications which are in turn arguments for other applications and may result in the multiple evaluation of expressions. Consider, for example:

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

Now, if we evaluate:

```
ADD 1 (ADD 1 2) => ... =>
```

```
IF ISZERO (ADD 1 2)
THEN 1
ELSE ADD (SUCC 1) (PRED (ADD 1 2)) => ... =>

ADD (SUCC 1) (PRED (ADD 1 2)) => ... =>

IF ISZERO (PRED (ADD 1 2))
THEN SUCC 1
ELSE ADD (SUCC (SUCC 1)) (PRED (PRED (ADD 1 2))) => ... =>

ADD (SUCC (SUCC 1)) (PRED (PRED (ADD 1 2))) => ... =>

IF ISZERO (PRED (PRED (ADD 1 2)))
THEN SUCC (SUCC 1)
ELSE ADD (SUCC (SUCC (SUCC 1))) (PRED (PRED (PRED (ADD 1 2)))) => ... =>

ADD (SUCC (SUCC (SUCC 1))) (PRED (PRED (PRED (ADD 1 2)))) => ... =>

IF ISZERO (PRED (PRED (PRED (ADD 1 2))))
THEN SUCC (SUCC (SUCC 1))
ELSE ... => ... =>

SUCC (SUCC (SUCC 1)) ==

4
```

is returned.

`ADD 1 2` has been evaluated 4 times even though it only appeared once originally. In the initial call to `ADD` the argument `ADD 1 2` is not evaluated because it is not a leftmost application. Instead it completely replaces the bound variable `Y` throughout `ADD`'s body. Thereafter it is evaluated in the `IF` condition but when `ADD` is called recursively it becomes an argument for the call so evaluation is again delayed. We also had to repeatedly evaluate other applications, for example `PRED (ADD 1 2)` in the condition, although we did not highlight these.

In general, for normal order evaluation, an unevaluated application argument will replace all occurrences of the associated bound variable in the function body. Each replacement is the site of initiation of potential additional evaluation. Clearly, the more often the bound variable appears in the body, the more argument evaluation may multiply.

8.4. Applicative order

Applicative order β reduction of an application requires the evaluation ofs both the function and the argument expressions. More formally, this involves the evaluation of the left most redex free of internal redexes. For a function application, this will result in each argument only being evaluated once.

For example, evaluating our previous example in applicative order:

```
ADD 1 (ADD 1 2) ->

ADD 1 3 -> ... ->

IF ISZERO 3
THEN 1
ELSE ADD (SUCC 1) (PRED 3) -> ... ->

ADD (SUCC 1) (PRED 3) ->
```

```
ADD 2 2 ->

IF ISZERO 2
THEN 2
ELSE ADD (SUCC 2) (PRED 2) -> ... ->

ADD (SUCC 2) (PRED 2) ->

ADD 3 1 -> ... ->

IF ISZERO 1
THEN 3
ELSE ADD (SUCC 3) (PRED 1) -> ... ->

ADD (SUCC 3) (PRED 1) ->

ADD 4 0 -> ... ->

IF ISZERO 0
THEN 4
ELSE ... -> ... ->

4
```

Here argument evaluation appears to be minimised. For example, `ADD 1 2` and `PRED (ADD 1 2)` are only evaluated once.

8.5. Consistent applicative order use

We have actually been using applicative order somewhat selectively; in particular, we are still evaluating `IF`s in normal order. Let us look at the previous example again, this time using untyped arithmetic to simplify things:

```
rec add x y =
 if iszero y
 then x
 else add (succ x) (pred y)
```

Now, consider:

```
succ (add one two)
```

First of all, the argument `add one two` is evaluated:

```
add one two  -> ... ->

if iszero two
then one
else add (succ one) (pred two)
```

Now, recall our definition of `if` as a syntactic simplification of:

```
def cond e1 e2 c = c e1 e2
```

Thus, after substitution for `add`s bound variables, `add 1 2` becomes:

```
cond one (add (succ one) (pred two)) (iszero two) ->
```

```
cond one (add (succ one) (pred two)) true
```

so now we have to evaluate:

```
add (succ one) (pred two) -> ... ->

add two one -> ... ->

if iszero one
then two
else add (succ two) (pred one)
```

Again, replacing `if` with `cond` we have:

```
cond two (add (succ two) (pred one)) (iszero one) ->

cond two (add (succ two) (pred one)) true
```

so we have to evaluate:

```
add (succ two) (pred one) -> ... ->

add three zero -> ... ->

if iszero zero
then three
else add (succ three) (pred zero)
```

which translates to:

```
cond three (add (succ three) (pred zero)) (iszero zero) ->

cond three (add (succ three) (pred zero)) true
```

so we have to evaluate:

```
add (succ three) (pred zero) -> ... ->

add four zero
```

and so on.

Evaluation will never terminate! This is not because `pred zero` is defined to be `zero`. The evaluation would still not terminate even if an error were returned from `pred zero`. Rather, the problem lies with consistent applicative order use.

`if` is just another function. When an `if` is used it is just the same as calling a function in an application: all arguments must be evaluated before substitution takes place. Recursive functions are built out of `if`s so if an `if`s argument is itself a recursive function call then, in applicative order, argument evaluation will recurse indefinitely. This does not occur with normal order evaluation because the recursive function call argument to `if` is not evaluated until it is selected in the `if`s body.

We can see the same difficulty more starkly with our construction for recursion. Recall:

```
def recursive f = λs.(f (s s)) λs.(f (s s))
```

and consider, for an arbitrary function `<function>`:

```
recursive <function> ==

λf.(λs.(f (s s)) λs.(f (s s))) <function> ->

λs.(<function> (s s) λs.(<function> (s s) ->

<function>
 (λs.(<function> (s s) λs.(<function> (s s)) ->

<function>
 (<function>
  (λs.(<function> (s s) λs.(<function> (s s))) ->

<function>
 (<function>
  (<function>
   (λs.(<function> (s s) λs.(<function> (s s))))
```

and so on. Again this won't terminate. This does not arise with normal order because argument evaluation is delayed. In this example, the self-application will depend on `<function>` calling itself recursively and is delayed until the recursive call is encountered as the leftmost application.

8.6. Delaying evaluation

With consistent applicative order use we need to find some means of delaying argument evaluation explicitly. One way, as we saw when we discussed recursion, is to add an extra layer of abstraction to make an argument into a function body and then extract the argument with explicit function application to evaluate the body.

For example, for an `if` we need to delay evaluation of the `then` and `else` options until one is selected. We might try changing an `if` to take the form:

```
def cond e1 e2 c = c λdummy.e1 λdummy.e2
```

to delay the evaluation of `e1` and `e2`. Here the idea is that if the condition `c` is `true` then:

```
λdummy.e1
```

is selected and if the condition is `false` then:

```
λdummy.e2
```

is selected. Sadly, this won't work. Remember:

```
def cond e1 e2 c = c λdummy.e1 λdummy.e2
```

is short-hand for:

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

so when `cond` is called the arguments corresponding to `e1` and `e2` are evaluated before being passed to:

```
λdummy.e1
```

and:

```
λdummy.e2
```

Instead, we have to change our notation for `if`. Now:

```
if <condition>
then <true choice>
else <false choice>
```

will be replaced by:

```
cond λdummy.<true choice> λdummy.<false choice> <condition>
```

Thus `<true choice>` and `<false choice>` will be inserted straight into the call to `cond` without evaluation. We have introduced the delay through a textual substitution technique which is equivalent to normal order evaluation.

Alternatively, we could redefine our `def` notation so it behaves like a **macro** definition and then our first attempt would work. A macro is a text substitution function. When a macro is called occurences of its bound variables in its body are replaced by the arguments. Macros are used for abstraction in programming languages, for example in C and in many assembly languages, but result in the addition of in-line text rather than layers of procedure or functions calls.

Here:

```
def <name> <bound variables> = <body>
```

might introduce the macro `<name>` and subsequent occurrences of:

```
<name> <arguments>
```

would require the replacment of `<bound variables>` in `<body>` with the corresponding `<arguments>` followed by the evaluation of the resulting `<body>`

This would introduce macro expansion in a form equivalent to normal order evaluation.

Here we will redefine `if ... then ... else ...`. Now, `true` and `false` must be changed to force the evaluation of the selected option. This suggests:

```
def true x y = x identity

def false x y = y identity
```

For arbitrary expressions `<expression1>` and `<expression2>`:

```
if true
then <expression1>
else <expression2> ==

cond λdummy.<expression1> λdummy.<expression2> true ==

true λdummy.<expression1> λdummy.<expression2> -> ... ->

λdummy.<expression1> identity ->

<expression1>
```

and:

```
if false
then <expression1>
```

```
else <expression2> ==

cond λdummy.<expression1> λdummy.<expression2> false ==

false λdummy.<expression1> λdummy.<expression2> -> ... ->

λdummy.<expression2> identity ->

<expression2>
```

This delayed evaluation approach is similar to the use of **thunks** to implement ALGOL 60 call by name. A thunk is a parameterless procedure which is produced for a call by name parameter to delay evaluation of that parameter until the parameter is encountered in the procedure or function body.

Note that to follow this through, all definitions involving booleans also have to change to accommodate the new forms for `true` and `false.` We won't consider this further.

We could also use abstraction to build an applicative order version of `recursive:` we won't consider this further either.

The effect of using abstraction to delay evaluation is to reintroduce the multiple evaluation associated with normal order. Any expression which is delayed by abstraction must be evaluated explicitly. Thus, if bound variable substitution places a delayed expression in several places then it must be explicitly evaluated in each place.

Clearly, for functional language implementations based on applicative order evaluation some compromises must be made. For example, LISP is usually implemented with applicative order evaluation but the conditional operator `COND` implicitly delays evaluation of its arguments. Thus, the definition of recursive functions causes no problems. LISP also provides the `QUOTE` and `EVAL` operators to delay and force expression evaluation explicitly.

8.7. Evaluation termination, the halting problem, evaluation equivalance and the Church-Rosser theorems

We have tacitly assumed that there is some equivalence between normal and applicative order and we have switched between them with cheery abandon. There are, however, differences. We have seen that normal order may lead to repetitive argument evaluation and that applicative order may not terminate. Of course, normal order may not terminate as well. One of our first examples:

```
λs.(s s) λs.(s s)
```

showed us this.

In general, there is no way of telling whether or not the evaluation of an expression will ever terminate. This was shown originally by Alan Turing who devised a formal model for computing based on what are known as Turing machines. Turing proved that it is impossible to construct a Turing machine to tell whether or not an arbitrary Turing machine halts: in formal terminology, the halting problem for Turing machines is undecidable.

Church's thesis hypothesised that all descriptions of computing are equivalent. Thus any result for one applies to the other as well. In particular, it has been shown that the λ calculus and Turing machines are equivalent: for every Turing machine there is an equivalent λ expression and vice-versa. Thus, the undecidability of the halting problem applies to the λ calculus as well so there is no way to tell if evaluation of an arbitrary λ expression terminates. In principle, we can just go on evaluating individual λ expressions in the hope that evaluation will terminate but there is no way of being sure that it will.

To return to normal and applicative order reduction: two theorems by Church and Rosser show that they are interchangeable but that normal order gives a better guarantee of evaluation termination.

The first Church-Rosser theorem shows that every expression has a unique normal form. Thus, if an expression is reduced using two different evaluation orders and both reductions terminate then they both lead to the same normal form. For example, if normal and applicative order reductions of an expression both terminate then they produce the same final result. This suggests that we can use normal and applicative orders as we choose.

The second Church-Rosser theorem shows that if an expression has a normal form then it may be reached by normal order evaluation. In other words, if any evaluation order will terminate then normal order evaluation is guaranteed to terminate.

These theorems suggest that normal order evaluation is the best bet if finding the normal form for an expression is an over-riding consideration.

As we have seen, there are practical advantages in the selective use of applicative order and in not evaluating function bodies even though the first reduces the likelihood of termination and the second may stop evaluation before a normal form is reached. We will discuss evaluation strategies for real functional languages in subsequent sections.

8.8. Infinite objects

The evaluation delay with normal order evaluation enables the construction of infinite structures. For example, for lists with normal order evaluation, if a `CONS` constructs a list from a recursive call then evaluation of that call is delayed until the corresponding field is selected. To illustrate this, we will use typeless versions of `CONS, HEAD` and `TAIL:`

```
def cons h t s = s h t

def head l = l λx.λy.x

def tail l = l λx.λy.y
```

Now, let us define the list of all numbers:

```
rec numblist n = cons n (numblist (succ n))

def numbers = numblist zero
```

In normal order, `numbers'` definition leads to:

```
numblist zero => ... =>

cons zero (numblist (succ zero) => ... =>

λs.(s zero (numblist (succ zero)))
```

Now:

```
head numbers => ... =>

λs.(s zero (numblist (succ zero))) λx.λy.x => ... =>

zero
```

and:

```
tail numbers => ... =>

λs.(s zero (numblist (succ zero))) λx.λy.y => ... =>
```

```
numblist (succ zero) => ... =>

λs.(s (succ zero) (numblist (succ (succ zero)))
```

so:

```
head (tail numbers) => ... =>

(tail numbers) λx.λy.x => ... =>

λs.(s (succ zero) (numblist (succ (succ zero)))) λx.λy.x => ... =>

(succ zero)
```

and:

```
tail (tail numbers) => ... =>

(tail numbers) λx.λy.y => ... =>

λs.(s (succ zero) (numblist (succ (succ zero)))) λx.λy.y => ... =>

numblist (succ (succ zero)) => ... =>

λs.(s (succ (succ zero)) (numblist (succ (succ (succ zero)))))
```

In applicative order, this definition would not terminate because the call to `numblist` would recurse indefinitely.

In normal order though, we have the multiple evaluation of `succ zero`, `succ (succ zero)` and so on. In addition, the list is recalculated up to the required value every time a value is selected from it.

8.9. Lazy evaluation

Lazy evaluation is a method of delaying expression evaluation which avoids multiple evaluation of the same expression. Thus, it combines the advantages of normal order and applicative order evaluation. With lazy evaluation, an expression is evaluated when its value is needed; that is when it appears in the function position in a function application. However, after evaluation all copies of that expression are updated with the new value. Hence, lazy evaluation is also known as *call by need*.

Lazy evaluation requires some means of keeping track of multiple copies of expressions. We will give each bound pair in an expression a unique subscript. During expression evaluation, when a bound pair replaces a bound variable it retains its subscript but the bound pair containing the variable and copies of it, and the surrounding bound pairs and their copies are given consistent new subscripts.

For example, consider:

$$(\lambda s.(s\ s)_1\ (\lambda x.x\ \lambda y.y)_2)_3$$

To evaluate the outer bound pair 3, the argument bound pair 2 is copied into the function body bound pair 1 which is renumbered 4:

$$((\lambda x.x\ \lambda y.y)_2\ (\lambda x.x\ \lambda y.y)_2)_4$$

Note that bound pair 2 occurs twice in bound pair 4.

To evaluate bound pair 4 first evaluate the function expression which is bound pair 2:

$(\lambda x.x\ \lambda y.y)_2$ =>

$\lambda y.y$

and then replace all occurrences of it in bound pair 4 with the new value and renumber bound pair 4 as 5 to get:

$(\lambda y.y\ \lambda y.y)_5$

Finally, evaluate bound pair 5:

$\lambda y.y$

Note that bound pair 2:

$(\lambda x.x\ \lambda y.y)_2$

has only been evaluated once even though it occurs in two places.

We can now see a substantial saving in normal order evaluation with recursion. To simplify presentation we will only number bound pairs which may be evaluated several times and we won't expand everything into lambda functions. We will also use applicative order to simplify things occasionally.

Consider addition once again:

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

For the evaluation of:

```
ADD 2 2 => ... =>

IF ISZERO 2
THEN 2
ELSE ADD (SUCC 2) (PRED 2)1 => ... =>

ADD (SUCC 2) (PRED 2)1 => ... =>

IF ISZERO (PRED 2)1
THEN (SUCC 1)
ELSE ADD (SUCC (SUCC 2)) (PRED (PRED 2)1)
```

evaluate:

$ISZERO\ (PRED\ 2)_1$

which leads to the evaluation of bound pair 1:

$(PRED\ 2)_1$ => ... => 1

which replaces all other occurrences of bound pair 1:

```
IF ISZERO 1
THEN SUCC 1
ELSE ADD (SUCC (SUCC 2)) (PRED 1)2 => ... =>

ADD (SUCC (SUCC 2)) (PRED 1)2 => ... =>
```

```
IF ISZERO (PRED 1)2
THEN (SUCC (SUCC 2))
ELSE ADD (SUCC (SUCC (SUCC 2))) (PRED (PRED 1)2)
```

so:

```
ISZERO (PRED 1)2
```

is evaluated which involves the evaluation of bound pair 2:

```
(PRED 1)2 => ... => 0
```

which replaces all other occurrences of bound pair 2:

```
IF ISZERO 0
THEN SUCC (SUCC 2)
ELSE ADD (SUCC (SUCC (SUCC 2))) (PRED 0) => ... =>

SUCC (SUCC 2) => ... =>

0
```

Here, the evaluation of arguments is delayed, as for normal order, but an argument is only evaluated once, as for applicative order.

Lazy evaluation avoids repetitive evaluation in infinite lists as the list is extended whenever head or tail evaluation occurs. This makes them useful when the same values are going to be used repeatedly and one wants to avoid recalculating them every time.

For example, we might define a function to calculate squares as:

```
def SQ X = X * X
```

Every time we required the square of a number, SQ would calculate it afresh. We could put the squares into an infinite list:

```
rec SQLIST N = (SQ N)::(SQLIST (SUCC N))

def SQUARES = SQLIST 0
```

so SQUARES is:

```
(SQ 0)::(SQLIST (SUCC 0)1)2
```

Here, we have only labeled the recursive extension of the list.

We now construct functions to select values from the list:

```
rec IFIND N L =
 IF ISZERO N
 THEN HEAD L
 ELSE IFIND (PRED N) (TAIL L)

def SQUARE N = IFIND N SQUARES
```

Now, if a particular square has been selected before then it is already in the list and is just selected. Otherwise, selection forces evaluation until the required value is found. This forced evaluation then leaves new values in the extended list ready for another value selection.

For example, consider:

```
SQUARE 2 => ...=>

IFIND 2 SQUARES => ... =>

IFIND 1 (TAIL SQUARES) => ... =>

IFIND 0 (TAIL (TAIL SQUARES)) => ... =>

HEAD (TAIL (TAIL SQUARES))
```

Now, the inner selection of:

```
TAIL SQUARES ==

TAIL ((SQ 0)::(SQLIST (SUCC 0)1)2)
```

results in the the selection of bound pair 2:

```
(SQLIST (SUCC 0)1)2
```

The next level of selection:

```
TAIL (TAIL SQUARES) => ... =>

TAIL (SQLIST (SUCC 0)1)2
```

results in the forced evaluation of bound pair 2:

```
(SQLIST (SUCC 0)1)2 => ... =>

((SQ (SUCC 0)1)::(SQLIST (SUCC (SUCC 0)1)3)4)5
```

Thus:

```
TAIL (SQLIST (SUCC 0)1)2 => ... =>

TAIL ((SQ (SUCC 0)1)::(SQLIST (SUCC (SUCC 0)1)3)4)5 =>... =>

(SQLIST (SUCC (SUCC 0)1)3)4
```

leads to the selection of bound pair 4.

Note that all occurrences of bound pair 2 were replaced by the new bound pair 5 so SQUARES is now associated with the list:

```
(SQ 0)::((SQ (SUCC 0)1)::(SQLIST (SUCC (SUCC 0)1)3)4)5
```

The final level of selection:

```
HEAD (TAIL (TAIL SQUARES)) => ... =>

HEAD (SQLIST (SUCC (SUCC 0)1)3)4
```

results in the forced evaluation of bound pair 4:

```
(SQLIST (SUCC (SUCC 0)1)3)4 => ... =>

(SQ (SUCC (SUCC 0)1)3)6::(SQLIST (SUCC (SUCC (SUCC 0)1)3)7)8
```

Occurrences of bound pair 4 are replaced by the new bound pair 8 so SQUARES is now:

```
(SQ 0)::(SQ (SUCC 0)1)::(SQ (SUCC (SUCC 0)1)3)6::
 (SQLIST (SUCC (SUCC(SUCC 0)1)3)7)8
```

Thus, evaluation of:

```
HEAD (TAIL (TAIL SQUARES))
```

requires the selection of:

```
(SQ (SUCC (SUCC 0)1)3)6
```

This requires the evaluation of:

```
(SUCC (SUCC 0)1)3
```

which in turn requires the evaluation of:

```
(SUCC 0)1 => ... => 1
```

which replaces all occurrences of bound pair 1. Now, SQUARES is:

```
(SQ 0)::(SQ 1)::(SQ (SUCC 1)3)6::(SQLIST (SUCC (SUCC 1)3)7)8
```

Thus evaluation of:

```
(SUCC (SUCC 0)1)3 => ... =>

(SUCC 1)3
```

gives:

```
2
```

which replaces all occurrences of bound pair 3 so SQUARES is now:

```
(SQ 0)::(SQ 1)::(SQ 2)6::(SQLIST (SUCC 2)7)8
```

Finally, evaluation of:

```
(SQ 2)6
```

gives:

```
4
```

which replaces all occurrences of bound pair 6 so SQUARES is now:

```
(SQ 0)::(SQ 1)::4::(SQLIST (SUCC 2))
```

If we now try:

```
SQUARE 1 => ... =>

IFIND 1 SQUARES => ... =>

IFIND 0 (TAIL SQUARES) => ... =>

HEAD (TAIL SQUARES) ==

HEAD (TAIL ((SQ 0)::(SQ 1)9::4::(SQLIST (SUCC 2)))) -> ... ->

HEAD ((SQ 1)9::4::(SQLIST (SUCC 2))) => ... =>

(SQ 1)9 => ... => 1
```

which replaces all occurrences of bound pair 9 so SQUARES is now:

```
(SQ 0)::1::4::(SQLIST (SUCC 2))
```

Thus, repeated list access evaluates more and more of the infinite list but avoids repetitive evaluation.

Lazy evaluation is used for Miranda lists.

8.10. Summary

In this chapter we have:

- compared normal and applicative order β reduction and seen that normal order reduction may be less efficient
- seen that consitent applicative order β reduction with our conditional expression representation leads to non-termination
- considered ways of delaying applicative order evaluation
- seen that the halting problem is unsolvable
- met the Church-Rosser theorems which suggest that normal order β reduction is most likely to terminate
- seen that normal order β reduction enables the construction of infinite objects
- met lazy evaluation as a way of combining the best aspects of normal and applicative order β reduction

Lazy evaluation is summarised below.

Lazy evaluation

i) number every bound pair

ii) To lazy evaluate (<function expression> <argument expression>)$_i$

a) lazy evaluate <function expression> to <function value>

b) if <function value> is λ<name>.<body>
then replace all free occurences of <name> in <body> with <argument expression>
and renumber consistently all surrounding bound pairs
and replace all occurences of (<function expression> <argument expression>)$_i$
with the new <body>
and lazy evaluate the new <body>

or

d) if <function value> is not a function
then lazy evaluate <argument expression>to <argument value>
and replace all occurences of (<function expression> <argument expression>)$_i$
with (<function value> <argument value>)
and return (<function value> <argument value>)

8.11. Exercises

1) Evaluate the following expressions using normal order, applicative order and lazy evaluation. Explain any differences in the final result and the number of reductions in each case:

```
i) λs.(s s) (λf.λa.(f a) λx.x λy.y)
ii) λx.λy.x λx.x (λs.(s s) λs.(s s))
iii) λa.(a a) (λf.λs.(f (s s)) λx.x)
```

9. FUNCTIONAL PROGRAMMING IN STANDARD ML

9.1. Introduction

ML(Meta Language) is a general purpose language with a powerful functional subset. It is used mainly as a design and implementation tool for computing theory based research and development. It is also used as a teaching language. ML is strongly typed with compile time type checking. Function calls are evaluated in applicative order.

ML originated in the mid 1970's as a language for building proofs in Robin Milner's LCF(Logic for Computable Functions) computer assisted formal reasoning system. SML(Standard ML) was developed in the early 1980's from ML with extensions from the Hope functional language. SML is one of the first programming languages to be based on well defined theoretical foundations.

We won't give a full presentation of SML. Instead, we will concentrate on how SML relates to our approach to functional programming.

SML is defined in terms of a very simple **bare language** which is overlaid with standard **derived forms** to provide a higher level syntax. Here, we will just use these derived forms.

We will explain SML with examples. As the symbol -> is used in SML to represent the types of functions, we will follow SML system usage and show the result of evaluating an expression as:

```
- <expression>;
> <result>
```

9.2. Types

Types are central to SML. Every object and construct is typed. Unlike Pascal, types need not be made explicit but they must be capable of being deduced statically from a program.

SML provides several standard types, for example for booleans, integers, strings, lists and tuples which we will look at below. SML also has a variety of mechanisms for defining new types but we won't consider these here.

When representing objects, SML always displays types along with values: