A Functional Program

```
let
  type intfun = int -> int

  function add(n: int) : intfun =
    let function h(m: int) : int = n+m
     in h
    end

  var addFive : intfun := add(5)
  var addSeven : intfun := add(7)
  var twenty := addFive(15)
  var twentyTwo := addSeven(15)

  function twice(f: intfun) : intfun =
    let function g(x: int) : int = f(f(x))
     in g
    end

  var addTen : intfun := twice(addFive)

  var seventeen := twice(add(5))(7)
  var addTwentyFour := twice(twice(add(6)))

 in addTwentyFour(seventeen)
end
```

PROGRAM 15.1. A Fun-Tiger program.

Closures

- In languages without nested functions (such as C), the run-time representation of a function value can be the address of the machine code for that function.
- When *nested functions* come into the picture, functions are represented as *closures*: records that contain the machine-code pointer and a way to access the necessary non-local variables (*environment*).
- One way of representing environments is using the *static link*. **Disadvantages**: it takes a chain of pointer dereferences to access the outermost variables and the garbage collector becomes less effective.

Heap-Allocated Activation Records

- The use static links in closures means that activation records for "enclosing" functions must not be destroyed upon their return because they serve as environments for other functions.
- So, activation records are stored on the heap instead of the stack. It is then up to the garbage collector to determine that it is safe to reclaim the heap-allocated frames.
- A refinement of this technique is to save on the heap only variables that *escape* (are used by inner-nested functions). Stack frames thus also hold a pointer to the escaping-variable record:
  1. has any local variables that an inner-nested procedure might need;
  2. a static link to the environment provided by the enclosing function.

Pure Functional Programming

Allows *equational reasoning* by prohibiting side-effects of functions:

1. Assignments to variables (except as initializations)
2. Assignments to fields of heap-allocated records
3. Calls to external functions that have visible side-effects (read, print, exit, ...).

Thus, functions return results *without changing the "world"* in any observable way! Instead of updating old values, functions always produce new values. I/O is performed in a *continuation-based* style (interestingly enough, I/O becomes now "visible" to the type-checker).

```
type key = string
type binding = int
type tree = {key: key,
             binding: binding,
             left: tree,
             right: tree}

function look(t: tree, k: key)
                       : binding =
  if k < t.key
       then look(t.left,k)
  else if k > t.key
       then look(t.right,k)
  else t.binding

function enter(t: tree, k: key,
               b: binding) =
  if k < t.key
    then if t.left=nil
      then t.left :=
              tree{key=k,
                   binding=b,
                   left=nil,
                   right=nil}
      else enter(t.left,k,b)
  else if k > t.key
    then if t.right=nil
      then t.right :=
              tree{key=k,
                   binding=b,
                   left=nil,
                   right=nil}
      else enter(t.right,k,b)
  else t.binding := b
```

(a) Imperative

```
type key = string
type binding = int
type tree = {key: key,
             binding: binding,
             left: tree,
             right: tree}

function look(t: tree, k: key)
                       : binding =
  if k < t.key
       then look(t.left,k)
  else if k > t.key
       then look(t.right,k)
  else t.binding

function enter(t: tree, k: key,
               b: binding) : tree =
  if k < t.key
    then
     tree{key=t.key,
          binding=t.binding,
          left=enter(t.left,k,b),
          right=t.right}

  else if k > t.key
    then
     tree{key=t.key,
          binding=t.binding,
          left=t.left,
          right=enter(t.right,k,b)}
  else tree{key=t.key,
            binding=b,
            left=t.left,
            right=t.right}
```

(b) Functional

PROGRAM 15.3. Binary search trees implemented in two ways.
From *Modern Compiler Implementation in ML*,

Types and Functions for Continuation-Based I/O

```
type answer
type stringConsumer = string -> answer
type cont = () -> answer

function getchar(c: stringConsumer) : answer
function print(s: string, c: cont) : answer
function flush(c: cont) : answer
function exit() : answer
```

PROGRAM 15.4. Built-in types and functions for PureFun-Tiger.
From *Modern Compiler Implementation in ML*,

Optimization of Pure Functional Languages

- In general, functional languages can use the same kinds of optimizations as imperative language compilers and more:

```
var a1 := 5
var b1 := 7
var r := record{a := a1, b := b1}
var x := f(r)
var y := r.a + r.b      ⇒      var y := 12
```

- On the other hand, in higher-order functional languages, calculating the control-flow graph can be a bit more complicated, because the control flow may be expressed through calls to function-variables instead of statically defined functions.

Program with Continuation-Based I/O

```
let
type intConsumer = int -> answer

function isDigit(s : string) : int =
      ord(s)>=ord("0") & ord(s)<=ord("9")

function getInt(done: intConsumer) =
 let function nextDigit(accum: int) =
       let function eatChar(dig: string) =
            if isDigit(dig)
             then nextDigit(accum*10+ord(dig))
             else done(accum)
        in getchar(eatChar)
       end
  in nextDigit(0)
 end

function putInt(i: int, c: cont) =
  if i=0 then c()
  else let var rest := i/10
           var dig := i - rest * 10
           function doDigit() = print(chr(dig), c)
        in putInt(rest, doDigit)
       end

function factorial(i: int) : int =
  if i=0 then 1 else i * factorial(i-1)

function loop(i) =
  if i > 12 then exit()
  else let function next() = getInt(loop)
        in putInt(factorial(i), next)
       end
in
   getInt(loop)
end
```

PROGRAM 15.5. PureFun-Tiger program to read *i*, print *i*!.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Generic program to print an int table

```
let
    type list = {head: int, tail: list}
    type observeInt  = (int,cont) -> answer

    function doList(f: observeInt, l: list, c: cont) =
      if l=nil then c()
      else let function doRest() = doList(f, l.tail, c)
            in f(l.head, doRest)
           end

    function double(j: int) : int = j+j

    function printDouble(i: int, c: cont) =
        let function again() = putInt(double(i),c)
         in putInt(i, again)
        end

    function printTable(l: list, c: cont) =
       doList(printDouble, l, c)

    var mylist := ···

 in printTable(mylist, exit)
end
```

PROGRAM 15.6. `printTable` in PureFun-Tiger.
From *Modern Compiler Implementation in ML*,

Inline Expansion

Functional programs tend to use many small functions that get passed from one place to another.

An important optimization technique is *inline expansion* of function calls: replacing a function call with a copy of the function body.

- How to perform inlining ?
- When to perform inlining and when not to ?

Avoiding Variable Capture

Local variables can create "holes" in the scope of outer variables.

For correctness, inlining should first rename (*α-convert*) the formal parameters of inner-nested functions.

```
let var x := 5                          let var x := 5
  function g(y:int): int =                function g(y:int): int =
    y + x                                   y + x
  function f(x:int): int =       ⇒        function f(a:int): int =
    g(1) + x                                (1 + x) + a
 in f(2) + x                             in f(2) + x
end                                     end
```

Program before and after Inlining

```
let
 type list = {head: int,
              tail: list}

 function double(j: int): int =
      j+j

 function printDouble(i: int) =
      (putInt(i);
       putInt(double(i)))

 function printTable(l: list) =
    while l <> nil
       do (printDouble(l.head);
           l := l.tail)

 var mylist := ···

in printTable(mylist)
end
```

(a) As written

```
let
 type list = {head: int,
              tail:list}

 function printTable(l: list) =
    while l <> nil
       do let var i := l.head
           in putInt(i);
              putInt(i+i);
              l := l.tail
          end

 var mylist := ···

in printTable(mylist)
end
```

(b) Optimized

PROGRAM 15.7. Regular Tiger `printTable`.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Inlining of Recursive Functions

To avoid expansion of only the first call to a recursive function (the first iteration of a loop) a *loop-preheader transformation* is used.

The idea is to split a function into:

- a *prelude* called from outside once, and
- a *loop header* which is recursively called from inside

```
function doList(fX:observeInt, lX:list, cX:cont) =
 let function doListX(f:observeInt, l:list, c:cont) =
    if l=nil then c()
    else let function doRest() = doListX(f, l.tail, c)
         in f(l.head, doRest)
         end
 in doListX(fX, lX, cX)
end
```

Inline Expansion & Loop Preheader

(a) When the actual parameters are simple variables $i_1, \ldots, i_n$.
Within the scope of:

`function` $f(a_1, \ldots, a_n) = B$

the expression

$f(i_1, \ldots, i_n)$

rewrites to

$B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$

(b) When the actual parameters are non-trivial expressions, not just variables.
Within the scope of:

`function` $f(a_1, \ldots, a_n) = B$

the expression

$f(E_1, \ldots, E_n)$

rewrites to

`let var` $i_1 := E_1$
$\vdots$
`var` $i_n := E_n$
`in` $B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$
`end`

where $i_1, \ldots, i_n$ are previously unused names.

ALGORITHM 15.8. Inline expansion of function bodies. We assume that no two declarations declare the same name.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

`function` $f(a_1, \ldots, a_n) =$
B

$\rightarrow$

`function` $f(a'_1, \ldots, a'_n) =$
`let function` $f'(a_1, \ldots, a_n) =$
$B[f \mapsto f']$
`in` $f'(a'_1, \ldots, a'_n)$
`end`

ALGORITHM 15.9. Loop-preheader transformation.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Loop-Invariant Hoisting

If every use of f' within `B` is of the form $f'(E_1, \ldots, E_{i-1}, a_i, E_{i+1}, \ldots, E_n)$ such that the ith argument is always a_i, then rewrite

function $f(a'_1, \ldots, a'_n) =$ let function $f'(a_1, \ldots, a_n) = B$ in $f'(a'_1, \ldots, a'_n)$ end	$\rightarrow$	function $f(a'_1, \ldots, a'_{i-1}, a_i, a'_{i+1}, \ldots, a'_n) =$ let function $f'(a_1, \ldots, a_n) = B$ in $f'(a'_1, \ldots, a'_{i-1}, a'_{i+1}, \ldots, a'_n)$ end

where every call $f'(E_1, \ldots, E_{i-1}, a_i, E_{i+1}, \ldots, E_n)$ within B is rewritten as $f'(E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n)$.

ALGORITHM 15.10. Loop-invariant hoisting.
From *Modern Compiler Implementation in ML*,


```
function printTable(l: list, c: cont) =
    let function doListX(l: list) =
          if l=nil then c()
          else let function doRest() =
                         doListX(l.tail)
                    var i := l.head
                    function again() =
                         putInt(i+i,doRest)
                 in putInt(i,again)
               end
   in doListX(l)
  end
```

PROGRAM 15.11. `printTable` as automatically specialized.
From *Modern Compiler Implementation in ML*,

Loop-Invariant Hoisting Tranformation (example)

We can avoid passing around values that are the same in every recursive call (e.g. `f` and `c` in `doListX`) by using a *loop-invariant hoisting* transformation (replace every use of `f` with `fX` and `c` with `cX`).

```
function doList(f:observeInt, lX:list, c:cont) =
 let function doListX(l:list) =
   if l=nil then c()
   else let function doRest() = doListX(l.tail)
     in f(l.head, doRest)
   end
 in doListX(lX)
end
```

Avoiding Code Explosion

If inline expansion is performed indiscriminantly, the size of the program explodes!

There are several heuristics to control code explosion:

1. Expand only *frequent* function-call sites (frequency can be determined either by static estimation [loop-nest depth] or by feedback from an execution profiler);
2. Expand only functions *with very small bodies* (so that the copied function body is not much larger than the instructions that would call the function);
3. Expand functions *called only once* and perform *dead function elimination* to the original program.

Closure Conversion

The aim is to *transform the program so that no function* appears to *access free (non-local) variables*. This is done by turning each free-variable access into a formal-parameter access:

Given a function $f(a_1, \ldots, a_n) = B$ at nesting depth d with escaping local variables (and formal parameters) $x_1, x_2, \ldots, x_n$ and nonescaping variables $y_1, \ldots, y_n$, rewrite into:

$$f(a_0, a_1, \ldots, a_n) = \texttt{let var } r := \{a_0, x_1, x_2, \ldots, x_n\} \texttt{ in } B' \texttt{ end}$$

where the new parameter a_0 is the static link which is now made into an explicit argument, and r is a record containing all the escaping variables *and* the enclosing static link.

Any use of a non-local variable (that comes from nesting depth $< d$) within B must be transformed into an access of some offset within the record a_0. The resulting body is B'.

Efficient Tail Recursion

A function call $f(x)$ within the body of a function $g(y)$ is in a tail position if "calling f is the last thing that g will do before returning".

1. `let var x :=` C_1 `in` B_1 `end`
2. $C_1(C_2)$
3. `if` C_1 `then` B_1 `else` B_2
4. $C_1 + C_2$

Tail calls can be implemented more efficiently than ordinary calls!

```
g(y) = let var x := h(y) in f(x) end
```

The result r returned from `f(x)` will also be the one returned from `g(y)`. Instead of pushing a new return address for `f` to return to, `g` could just give `f` the return address given to `g` and have `f` return directly.

Program after Closure Conversion

```
type mainLink = { ··· }
type printTableLink= {SL: mainLink, cFunc: cont, cSL: ?}
type cont = ? -> answer
type doListXLink1 = {SL: printTableLink, l: list}
type doListXLink2 = {SL: doListXLink1, i: int,
                     doRestFunc: cont, doRestSL: doListXLink1}

function printTable(SL: mainLink, l: list, cFunc: cont, cSL: ?) =
    let var r1 := printTableLink{SL=SL,cFunc=cFunc,cSL=cSL}
        function doListX(SL: printTableLink, l: list) =
          let var r2 := doListXLink1{SL: printTableLink, l=l}
           in if r2.l=nil then SL.cFunc(SL.cSL)
              else let function doRest(SL: doListXLink1) =
                          doListX(SL.SL, SL.l.tail)
                       var i := r2.l.head
                       var r3 := doListXLink2{SL=r2, i=i,
                                   doRestFunc=doRest, doRestSL=r2}
                       function again(SL: doListXLink2) =
                          putInt(SL.SL.SL, SL.i+SL.i,
                                 SL.doRest.func, SL.doRestSL)
                    in putInt(SL.SL,i, again,r3)
                   end
     in doListX(r1,l)
    end
```

PROGRAM 15.12. `printTable` after closure conversion.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

printTable:	allocate record `r1`	printTable:	allocate stack frame
	jump to doListX		jump to whileL
doListX:	allocate record `r2`	whileL:	
	if `l=nil` goto doneL		if `l=nil` goto doneL
	`i := r2.l.head`		`i := l.head`
	allocate record `r3`		
	jump to `putInt`		call `putInt`
again:	add `SL.i+SL.i`		add `i+i`
	jump to `putInt`		call `putInt`
doRest:	jump to doListX		jump to whileL
doneL :	jump to `SL.cFunc`	doneL:	return

(a) Functional program (b) Imperative program

FIGURE 15.13. `printTable` as compiled.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

```
type tree = {key: ()->key,
             binding: ()->binding,
             left: ()->tree,
             right: ()->tree}

function look(t: ()->tree, k: ()->key) : ()->binding =
  if k() < t().key() then look(t().left,k)
  else if k() > t().key() then look(t().right,k)
  else t().binding
```

PROGRAM 15.14. Call-by-name transformation applied to Program 15.3a.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Implementation of Tail Recursion Optimization

A tail call can be implemented more like a `jump` than a `call`:

1. Move actual parameters into argument registers.
2. Restore callee-save registers.
3. Pop the stack frame of the calling function, *if it has one.*
4. Jump to the callee.

In many cases, step 1 is eliminated by the coalescing phase of the compiler. Also, steps 2 and 3 are eliminated because the calling function has no stack frame — any function that can do all its computation in callee-save registers needs no frame.

Thus, a tail call can be as cheap as a `jump` instruction!

Equational Reasoning in Functional Programs

One important principle of equational reasoning is **β-substitution**:

> if $f(x) = B$, then any application $f(E)$ to an expression E is equivalent to B with every occurrence of x replaced with E.

`let`
`  function loop(z:int): int =`
`    if z>0 then z`
`           else loop(z)`
`  function f(x:int): int =`
`    if y>8 then x`
`           else -y`
` in`
`   f(loop(y))`
`end`
	`let`
	`  function loop(z:int): int =`
	`    if z>0 then z`
	`           else loop(z)`
	`  function f(x:int): int =`
	`    if y>8 then x`
	`           else -y`
	` in if y>8 then loop(y)`
	`           else -y`
	`end`

Lazy Evaluation

- In pure functional languages, if a program A is obtained using β-substitutions from B, then both programs will never give different results *if they both halt*; however, A and B are not necessarily equivalent as they might not halt on the same inputs!
- To remedy this (partial) failure of equational reasoning, we can introduce *lazy evaluation* into the programming language.
- Under lazy evaluation, *an expression is not evaluated unless its value is demanded by some other part of the computation.*
- In contrast, *strict languages* (ML, C, Java, Erlang) evaluate each expression as the control flow of the program reaches it.

Call-by-Name Evaluation

Most languages pass function arguments using *call-by-value*:

> e.g. upon a call to `f(g(x))`, first `g(x)` is computed and the result is passed to `f`. The computation is unnecessary if `f` does not need to use its argument!

Call-by-name evaluation avoids this problem. Under this evaluation scheme, each variable is not a simple value but a *thunk*: a function that computes the value of the variable on demand.

```
let                        let
   var a := 5+7               function a() = 5+7
 in                  ⇒     in
   a + 10                     a() + 10
end                        end
```

The problem with call-by-name is that each thunk may be executed many times, repeatedly producing the same result.

Call-by-Need (Lazy Evaluation)

- It is a modification of call-by-name that never evaluates the same thunk twice.
- Each thunk is equipped with a *memo slot* that stores its value. Each evaluation of the thunk checks the memo slot: if full, the *memoized* value is returned; if empty, the thunk function is called.
- Thunks can be represented as two-element records of the form

$$\langle \textit{thunk\_function}, \textit{memo\_slot} \rangle$$

  An *unevaluated* thunk contains an arbitrary thunk function, and the memo slot is a static link to be used in calling the thunk function. An *evaluated* thunk has the previously computed value in its memo slot, and its thunk function just returns the memo-slot value.

Optimization of Lazy Functional Programs

Lazy functional languages can use the same kinds of optimizations as imperative or strict functional languages and more! For example:

Invariant hoisting The following is a valid transformation in a lazy functional language:

`function f(i:int): intfun =`
`  let`
`      function g(j:int) = h(i) * j`
`   in g`
`  end`
	`function f(i:int): intfun =`
	`  let var hi := h(i)`
	`      function g(j:int) = hi * j`
	`   in g`
	`  end`

but not in a strict language if the transformation appears in a context as `var a := f(42)` where `a` is never called at all and `h(42)` infinitely loops.

Dead-Code Removal

Another subtle problem with strict programming languages is the removal of dead code. Consider:

```
function f(i:int): int =
  let var d := g(x)
   in i + 2
  end
```

- In an imperative language (e.g. C), we cannot remove `g(x)` because it might contain side-effects that are needed by the program.
- In a strict pure functional language, removing `g(x)` might turn a non-terminating computation into a terminating one!
- In a lazy fuctional language, `g(x)` can be safely removed.

```
type intList = {head: int, tail: intList}
type intfun = int->int
type int2fun = (int,int) -> int

function sumSq(inc: intfun, mul: int2fun, add: int2fun) : int =
let
  function range(i: int, j: int) : intList =
    if i>j then nil else intList{head=i, tail=range(inc(i),j)}

  function squares(l: intList) : intList =
    if l=nil then nil
    else intList{head=mul(l.head,l.head), tail=squares(l.tail)}

  function sum(accum: int, l: intList) : int =
    if l=nil then accum else sum(add(accum,l.head), l.tail)

 in sum(0,squares(range(1,100)))
end
```

PROGRAM 15.15. Summing the squares.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

```
function look(t: tree, k: key) : ()->binding =
  if k < t.key() then look(t.left(),k)
  else if k > t.key() then look(t.right(),k)
  else t.binding
```

PROGRAM 15.16. Partial call-by-name using the results of strictness analysis; compare with Program 15.14.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Deforestation

In any language, it is common to break a program into a part that produces a data structure and another part that consumes it.

```
sumSquare n = sum (map square(upto 1 n))
```

- A *deforestation* transformation remove intermediate lists and trees and performs all operations in one pass.
- Deforestation is not valid in the presence of side-effects because it (usually) changes the order of operations.
- Deforestation is always legal in pure functional languages.

```
sumSquareDef acc m n =
  if m > n then acc
           else sumSquareDef (acc + square m) (m + 1) n
  end
```

Strictness Analysis

The overhead of thunk creation and evaluation is quite high.
It is better to use thunks only where they are needed:

> if a function $f(x)$ is certain to evaluate its argument x, there is no need to pass a thunk for x; we can just pass an evaluated x instead

We are trading trading an evaluation now for a certain eventual evaluation.

> A function $f(x_1, \ldots, x_n)$ is *strict in* x_i if, whenever a would fail to terminate, then $f(b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n)$ also fails to terminate, regardless of whether the b_j terminate.

Strictness Analysis (cont)

```
function f(x:int, y:int): int = x + x + y
```

```
function g(x:int, y:int): int = if x>0 then y else x
```

```
function h(x:string, y:int): tree =
         tree(key=x, binding=y, left=nil, right=nil)
```

```
function j(x:int): int = j(0)
```

In general, *exact strictness information is not computable* —like e.g. liveness and many other dataflow analyses— and thus compilers must *use a conservative approximation*:

> when the strictness of a function argument cannot be determined, the argument must be assumed non-strict.