Here's some background material on lambda calculus. I suggest that you read up to but not including the "denotational semantics" section.

The original URL for this material was http://cs.wwc.edu/~aabyan/221_2/PLBOOK/Functions.html . Thanks to Anthony A. Aaby at Walla Walla College, and thanks to the Wayback Machine for saving a copy when it went offline. I have corrected a few special characters.

Functional Programming

*A **functional program** consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. **Reduction** consists of replacing some part P of E by another expression P' according to the given rewrite rules. ... This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. The expression E* thus obtained is called the **normal form** of E and constitutes the output of the functional program* -H. P. Barendregt

Functional programming is characterized by the programming with values, functions and functional forms.

Keywords and phrases: Lambda calculus, free and bound variables, scope, environment, functional programming, combinatorial logic, recursive functions, functional, curried function.

Functional programming languages are the result of both abstracting and generalizing the data type of maps. Recall, the mapping *m* from each element *x* of *S* (called the domain) to the corresponding element *m(x)* of *T* (called the range) is written as:

$$m : S \rightarrow T$$

For example, the squaring function is a function of type:

$$\text{sqr} : \text{Num} \rightarrow \text{Num}$$

and may be defined as:

$$\text{sqr where } x \mapsto x*x$$

A linear function f of type

$$f : \text{Num} \rightarrow \text{Num}$$

may be defined as:

$$f \text{ where } x \mapsto 3*x + 4$$

The function:

$$g \text{ where } x \mapsto 3*x^2 + 4$$

may be written as the composition of the functions f and sqr as:

$$f \circ \text{sqr}$$

where

$$(f \circ \text{sqr})(x) = f(\text{sqr}(x)) = f(x*x) = 3 * x^2 + 4$$

The compositional operator is an example of a *functional form*. Functional programming is based on the mathematical concept of a function and functional programming languages include the following:

- A set of primitive functions.
- A set of functional forms.
- The *application* operation.
- A set of data objects and associated functions.
- A mechanism for binding a name to a function.

LISP, FP, Scheme, ML, Miranda and Haskell are just some of the languages to implement this elegant computational paradigm.

The basic concepts of functional programming originated with LISP. Functional programming languages are important for the following reasons.

- Functional programming dispenses with the assignment command freeing the programmer from the rigidly sequential mode of thought required with the assignment command.
- Functional programming encourages thinking at *higher levels of abstraction* by providing higher-order functions -- functions that modify and combine existing programs.
- Functional programming has natural implementation in concurrent programming.
- Functional programming has important application areas. Artificial intelligence programming is done in functional programming languages and the AI techniques migrate to real-world applications.
- Functional programming is useful for developing *executable specifications* and *prototype implementations*.
- Functional programming has a close relationship to computer science theory. Functional programming is based on the lambda-calculus which in turn provides a framework for studying decidability questions of programming. The essence of denotational semantics is the translation of conventional programs into equivalent functional programs.

  Terminology. Functional programming languages are called *applicative* since the functions are applied to their arguments, *declarative* and *non-procedural* since the definitions specify what is computed and not how it is computed.

1 The Lambda Calculus

Functional programming languages are based on the lambda-calculus. The lambda-calculus grew out of an attempt by Alonzo Church and Stephen Kleene in the early 1930s to formalize the notion of computability (also known as *constructibility* and *effective calculability*). It is a formalization of the notion of functions as rules (as opposed to functions as tuples). As with mathematical expressions, it is characterized by the principle that *the value of an expression depends only on the values of its subexpressions*. The lambda-calculus is a simple language with few constructs and a simple semantics. But, it is expressive; it is sufficiently powerful to express all computable functions.

As an informal example of the lambda-calculus, consider the function defined by the polynomial expression

$$x^2 + 3x - 5.$$

The variable x is a parameter. In the lambda-calculus, the notation λx.M is used to denote a function with parameter x and body M. That is, x is mapped to M. We rewrite our function in this format

$$\lambda x.(x^2+ 3x - 5)$$

and read it as ``the function of *x* whose value is defined by $x^2 + 3x - 5$''. The lambda-calculus uses prefix form and so we rewrite the body in prefix form,

λx. (- (+ (* x x) (* 3 x)) 5).

The lambda-calculus *curries* its functions of more than one variable i.e. (+ x y) is written as ((+ x) y), the function (+ x) is the function which adds something to x. Rewriting our example in this form we get:

λx.((- ((+ ((* x) x)) ((* 3) x))) 5)

To denote the application of a function f to an argument a we write

f a

To apply our example to the value 1 we write

λx.((- ((+ ((* x) x)) ((* 3) x))) 5) 1.

To evaluate the function application, we remove the λx. and replace each remaining occurence of x with 1 to get

((- ((+ ((* 1) 1)) ((* 3) 1))) 5)

then evaluate the two multiplication expressions

((- ((+ 1) 3)) 5)

then the addition

((- 4) 5)

and finally the subtraction

$^-1$.

We say that the variable x is *bound* in the lambda-expression λx.B. A variable occuring in the lambda-expression which is not bound is said to be *free*. The variable x is free in the lambda-expression λy.((+ x) y). The *scope* of the variable introduced (or bound) by lambda is the entire body of the lambda-abstraction.

The lambda-notation extends readily to functions of several arguments. Functions of more than one argument can be *curried* to produce functions of single arguments. For example, the polynomial expression xy can be written as

λx. λy. xy

When the lambda-abstraction λx. λy. xy is applied to a single argument as in (λx. λy. xy 5) the result is λy. 5y, a function which multiplies its argument by 5. A function of more than one argument is reguarded as a *functional* of one variable whose value is a function of the remaining variables, in this case, ``multiply by a constant function.''

The special character of the lambda-calculus is illustrated when it is recognized that functions may be applied to other functions and even permit self application. For example let C = λf. λx . (f(fx))

The pure lambda-calculus does not have any built-in functions or constants. Therefore, it is appropriate to speak of the lambda-calculi as a family of languages for computation with functions. Different languages are obtained for different choices of functions and constants.

We will extend the lambda-calculus with common mathematical operations and constants so that λx.((+ 3) x) defines a function that maps x to x+3. We will drop some of the parentheses to improve the readability of the lambda expressions.

A lambda-*expression* is *executed* by *evaluating* it. Evaluation proceeds by repeatedly selecting a *reducible expression* (or *redex*) and reducing it. For example, the expression (+ (* 5 6) (* 8 3)) reduces to 54 in the following sequence of reductions.

```
(+  (*  5 6) (* 8 3))        → (+  30  (* 8 3))
                          → (+  30   24)
                          → 54
```

When the expression is the application of a lambda-abstraction to a term, the term is substituted for the bound variable. This substitution is called β-*reduction*. In the following sequence of reductions, the first step an example of β-*reduction*. The second step is the reduction required by the addition operator.

```
(λx.((+ 3) x)) 4



((+  3) 4)



7
```

The pure lambda-calculus has just three constructs: primitive symbols, function application, and function creation. Figure N.1gives the syntax of the lambda-calculus.

Figure N.1: **The Lambda Calculus**

Syntax:

L in Lambda Expressions
x in Symbols

L ::= x | (L L) | (λx.L)

(L L) is function application, and
(λx.L) is a lambda-abstraction which defines a function with argument x and body L.

We say that the variable x is *bound* in the lambda-expression λx.B. A variable which occurs in but is not bound in a lambda-expression is said to be *free*. The *scope* of λx. is B. In the lambda-expression λy.x+y, x is free and y is bound.

We adopt the following notational conventions:

- We extend the lambda-calculus with the usual constants and functions so we allow

  (λx.((+ x) 3)) to represent the function x + 3

- We usually drop the outermost parentheses so we may write

  λx.((+ x) 3) instead of (λx.((+ x) 3)) and
  λx.((+ x) 3) 4 instead of (λx.((+ x) 3) 4)

- Function application associates to the left so we may write

  (+ x 3) instead of ((+ x) 3) that is, we may write
  λx.+ x 3 instead of λx.((+ x) 3)

- The body of a lambda-abstraction extends as far right as possible so we *must* write

  (λx.+ x 3) 4 instead of λx.+ x 3 4

- Replace the body of a lambda-abstraction with conventional infix notation so we may write

  (λx.x + 3) 4 instead of (λx.+ x 3) 4

- Multiple parameters are written together so we may write

  λxy.x + y instead of λx.λy.x + y

Operational Semantics

Calculation in the lambda-calculus is by rewriting (reducing) a lambda-expression to a normal form. For the pure lambda-calculus, lambda-expressions are reduced by substitution. That is, occurrences of the parameter in the body are replaced with (copies of) the argument. In our extended lambda-calculus we also apply the usual reduction rules. For example,

1.	$\lambda x.(x^2 - 5)\ 3$	f(3) where $f(x) = x^2 - 5$
2.	$3^2 - 5$	by substitution
3.	9 - 5	power
4.	4	subtraction

The normal form is formally defined in the following definition.

> **Definition:** A lambda-expression is said to be in **normal form** if no **β-redex**, a subexpression of the form (λx.P Q), occurs in it.

Non-terminating computations are examples of expressions that do not have normal forms. The lambda-expression

(λx.x x) (λx.x x)

does not have a normal form as we shall soon see.

We define substitution, B[x:M], to be the replacement of all free occurences of x in B with M. Figure N.2 contains a formal definition of substitution.

Figure N.2: **Substitution**

s[x:M]	= if (s=x) then M else s
(A B)[x:M]	= (A[x:M] B[x:M])
(λx.B)[x:M]	= (λx.B)
(λy.B)[x:M]	= if (z is a symbol not free in B or M) then λz.(B[y:z][x:M])

where s is a symbol, M, A and B are lambda-expressions.

Lambda expressions are simplified using β-reduction. β-reduction applies a lambda-abstraction to an argument producing an instance of the body of the lambda-abstraction in which (free) occurrences of the formal parameter in the body are replaced with (copies of) the argument. With the definition of substitution in Figure N.2 and the formal definition of β-reduction in Fugure N.3, we have the tools needed to reduce lambda-expressions to normal forms.

Figure N.3: **β-reduction**

$$(\lambda x.B)\ e \Rightarrow B[x:e]$$

It is easy to see that the lambda-expression

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

does not have a normal form because when the second expression is substituted into the first, the resulting expression is identical to the given lambda-expression.

Figure 2 defines the operational semantics of the lambda-calculus in terms of β-reduction.

Figure N.4: **Operational semantics for the lambda-calculus**

Interpreter: reduce expression E to normal form.

Reduce in **L** → **L**

$$\begin{aligned} &\mathit{Reduce}[s] &&= s \\ &\mathit{Reduce}[\lambda x.B\ M] &&= \mathit{Reduce}[\ B[x:M]\ ] \\ &\mathit{Reduce}[L_1\ L_2] &&= (\mathit{Reduce}[\ L_1\ ]\ \mathit{Reduce}[\ L_2\ ]) \end{aligned}$$

where

s is a symbol and B, L_1, L_2, and M are lambda-expressions

The operational semantics of Figure N.4 describe a syntactic transformation of the lambda-expressions.

Reduction Order

Given a lambda-expression, the substitution and β-reduction rules provide the tools required to reduce a lambda-expression to normal form but do not tell us what order to apply the reductions when more than one redex is avaliable. The following theorem, due to Curry, states that if an expression has a normal form, then that normal form can be found by leftmost reduction.

Theorem: If E has a normal form N then there is a leftmost reduction of E to N.

The leftmost outermost reduction (*normal order reduction*) strategy is called *lazy reduction* because it does not first evaluate the arguments but substitutes the arguments directly into the expression. *Eager reduction* is when the arguments are reduced before substitution.

A function is *strict* if it is sure to need its argument. If a function is non-strict, we say that it is *lazy*.

parameter passing: by value, by name, and lazy evaluation

Infinite Data Structures

call by need

streams and perpetual processes

A function f is *strict* if and only if $(f\ \bot) = \bot$

Scheme evaluates its parameters before passing (eliminates need for renaming) a space and time efficiency consideration.

Denotational Semantics

In the previous section we looked at the *operational* semantics of the lambda-calculus. It is called operational because it is `dynamic', it sees a function as a sequence of operations. A lambda-expression was evaluated by purely *syntactic* transformations without reference to what the expressions `mean'. The purpose of the *denotational semantics* of a language is to assign a value to every expression in the langauge.

We can express the semantics of the lambda-calculus as a mathematical function, **Eval**, from expressions to values. For example,

Eval[+ 3 4] = **7**

defines the value of the expression (+ 3 4) to be **7**. Actually something more is required, in the case of variables and function names, the function **Eval** requires a second parameter containing the environment *rho* which contains the associations between variables and their values. Some programs go into infinite loops, some abort with a runtime error. To handle these situations we introduce the symbol ⊥ pronounced `bottom'.

Figure N.5 gives a denotational semantics for the lambda-calculus.

Figure N.5: **Denotational semantics for the lambda-calculus**

Semantic Domains:

s in **D**

Semantic Function:

Eval in **L** → **D**

Semantic Equations:

$$\begin{aligned} \mathit{Eval}\,[\,s\,] &= \mathbf{s} \\ \mathit{Eval}\,[\,(\lambda x.B\ M)\,] &= \mathit{Eval}\,[\,B[x{:}M]\,] \\ \mathit{Eval}\,[\,(L_1\ L_2)\,] &= (\mathit{Eval}\,[\,L_1\,]\ \mathit{Eval}\,[\,L_2\,]) \\ \mathit{Eval}\,[\,E\,] &= \bot \end{aligned}$$

where s is a symbol, B, L_1, L_2, and M are expressions, B[x:M] is substitution as in Figure N.2, E is an expression which does not have a normal form, and **_|_** is pronounced bottom.

The denotational semantics of Figure N.5 describe a mapping of lambda expressions to values in some semantic domain.

Recursive Functions

We extend the syntax of the lambda-calculus to include named expressions as follows:

Lambda Expressions

L ::= ...| x : L | ...

where x is the name of the lambda-expression L.

With the introduction of named expressions we have the potential for recursive definitions since the extended syntax permits us to name lambda-abstractions and then refer to them within a lambda-expression. Consider the following recursive definition of the factorial function.

FAC : λn.(if (= n 0) 1 (* n (FAC (- n 1))))

which with syntactic sugaring is

FAC : λn.if (n = 0) then 1 else (n * FAC (n - 1))

We can treat the recursive call as a free variable and replace the previous definition with the following.

FAC : (λfac.(λn.(if (= n 0) (* n (fac (- n 1))))) FAC)

Let

H : λfac.(λn.(if (= n 0) 1 (* n (fac (- n 1)))))

Note that H is not recursively defined. Now we can redefine FAC as

FAC : (H FAC)

This definition is like a mathematical equation. It states that when the function H is applied to FAC, the result is FAC. We say that FAC is a *fixed point* or *fixpoint* of H. In general functions may have more than one fixed point. In this case the desired fixed point is the mathematical function factorial. In general, the `right' fixed point turns out to be the unique *least fixed point*.

It is desirable that there be a function which applied to a lambda-abstraction returns the least fixed point of that abstraction. Suppose there is such a function Y where,

FAC : Y H

Y is called a *fixed point combinator*. With the function Y, this definition of FAC does not use of recursion. From the previous two definitions, the function Y has the property that

Y H = H (Y H)

As an example, here is the computation of FAC 1 using the Y combinator.

```
FAC 1 = (Y H) 1
      = H  (Y H) 1
      = λfac.(λn.(if (= n 0) 1 (*  n (fac (- n 1)))))  (Y H) 1
      = λn.(if (= n 0) 1 (*  n((Y H)(- n 1)))) 1
      = if (= 1 0) 1 (*  1 ((Y H)(-11)))
      = (* 1 ((Y H)(-11)))
      = (* 1 ((Y H)0))
      = (* 1 (H  (Y H) 0))
      ...
      = (* 1 1)
      =  1
```

The function Y can be defined in the lambda-calculus.

Y : λh.(λx.(h (x x)) λx.(h (x x)))

It is especially interesting because it is defined as a lambda-abstraction without using recursion. To show that this lambda-expression properly defines the Y combinator, here it is applied to H.

```
(Y H)  =  (λh.(λx.(h  (x  x)) λx.(h  (x x))) H)
       =  (λx.(H  (x  x)) λx.(H  (x x)))
       =  H  ( λx.(H  (x x))λx.(H  (x x)))
       =  H  (Y  H)
```

Lexical Scope Rules

Blocks with local definitions may be defined in the lambda-calculus. We introduce two kinds of blocks, let and letrec expressions. Nonrecursive definitions are introduced with let expressions:

let n : E in B is an abbreviation for (λn.B) E

Here is an example using the let-extension.

let x : 3 in (* x x)

Lets may be used where ever a lambda-expression is permitted. For example,

λy. let x : 3 in (* y x)

is equivalent to

λy. (* y 3)

Simple recursive definitions are introduced with letrec expressions which are defined in terms of let expressions and the Y combinator:

letrec n : E in B is an abbreviation for let n : Y (λn.E) in B

Let and letrec expressions may be nested. The definitions of the let and letrec expressions are restated in Figure N.6.

Figure M.6: **Lexical Scope Rules**

$$\begin{aligned} &\text{let } n : E \text{ in } B &&= (\lambda n.B)\ E \\ &\text{letrec } n : E \text{ in } B &&= \text{let } n : Y\ (\lambda n.E) \text{ in } B \end{aligned}$$

Mutual recursion may also be defined but is beyond the scope of this text.

Translation Semantics and Combinators

The β-reduction rule is expensive to implement. It requires the textual substitution of the argument for each occurrence of the parameter and further requires that no free variable in the argument should become bound. This has lead to the study of ways in which variables can be eliminated.

Curry, Feys, and Craig define a number of *combinators* among them the following:

$$\begin{aligned} &\mathbf{S} = \lambda f .(\ \lambda g .(\ \lambda x.\ f\ x\ (\ g\ x\ )\ )\ ) \\ &\mathbf{K} = \lambda x\ .\lambda y.\ x \\ &\mathbf{I} = \lambda x.x \\ &\mathbf{Y} = \lambda f.\ \lambda x.(\ f(x\ x))\ \lambda x.(f\ (x\ x)) \end{aligned}$$

These definitions lead to transformation rules for sequences of combinators. The reduction rules for the SKI calculus are given in Figure N.7.

Figure N.7: **Reduction rules for SKI calculus**

$$\begin{aligned} &\mathbf{S}\ f\ g\ x &&\rightarrow f\ x\ (g\ x) \\ &\mathbf{K}\ c\ x &&\rightarrow c \\ &\mathbf{I}\ x &&\rightarrow x \\ &\mathbf{Y}\ e &&\rightarrow e\ (\mathbf{Y}\ e) \\ &(A\ B) &&\rightarrow A\ B \\ &(A\ B\ C) &&\rightarrow A\ B\ C \end{aligned}$$

The reduction rules require that reductions be performed left to right. If no **S**, **K**, **I**, or **Y** reduction applies, then brackets are removed and reductions continue.

The SKI calculus is computationally complete; that is, these three operations are sufficient to implement any operation. This is demonstrated by the rules in Figure N.8.

Figure N.8: **Translation Semantics for the Lambda calculus**

$$\begin{aligned} &\textit{Compile}\ [\ s\ ] &&\rightarrow s \\ &\textit{Compile}\ [\ (E_1\ E_2)] &&\rightarrow (\textit{Compile}\ [\ E_1]\ \textit{Compile}\ [\ E_2\ ]) \\ &\textit{Compile}\ [\ \lambda x.E] &&\rightarrow \textit{Abstract}\ [\ (x, \textit{Compile}\ [\ E]\ )\ ] \\ &\textit{Abstract}\ [\ (x, s)\ ] &&\rightarrow \text{if } (s{=}x) \text{ then } \mathbf{I} \text{ else } (\mathbf{K}\ s) \\ &\textit{Abstract}\ [\ (x, (E_1\ E_2))] &&\rightarrow ((\mathbf{S}\ \textit{Abstract}\ [\ (x, E_1)]\ )\ \textit{Abstract}\ [\ (x, E_2)\ ]\ ) \end{aligned}$$

where s is a symbol.

which translate lambda-expressions to formulas in the SKI calculus.

Any functional programming language can be implemented by a machine that implements the SKI combinators since, functional languages can be transformed into lambda-expressions and thus to SKI formulas.

Function application is relatively expensive on conventional computers. The principle reason is the complexity of maintaining the data structures that support access to the bound identifiers. The problems are especially severe when higher-order functions are permitted. Because a formula of the SKI calculus contains no bound identifiers, its reduction rules can be implemented as simple data structure manipulations. Further, the reduction rules can be applied in any order, or in parallel. Thus it is possible to design massively parallel computers (*graph reduction machines*) that execute functional languages efficiently.

Recursive functions may be defined with the Y operator.

Optimizations

Notice that the size of the SKI code grows quadratically in the number of bound variables. Figure N.9.

$$\mathbf{B} = \lambda x\ .(\ \lambda y\ .(\ \lambda z.\ ((x\ y)\ z)))$$
$$\mathbf{C} = \lambda x\ .(\lambda y.(\lambda z((x\ z)\ y)))$$

with the corresponding reduction rules.

$$\mathbf{B}\ a\ b\ c \rightarrow ((a\ b)\ c)$$
$$\mathbf{C}\ a\ b\ c \rightarrow ((a\ c)\ b)$$

Having these combinators we can simplify the expressions obtained by applying the rules in Figure N.9.

Figure N.9: **Optimizations for SKI code**

$$\begin{array}{ll} \mathbf{S}\ (\mathbf{K}\ e)\ (\mathbf{K}\ f) & \rightarrow \mathbf{K}\ (e\ f) \\ \mathbf{S}\ (\mathbf{K}\ e)\ \mathbf{I} & \rightarrow e \\ \mathbf{S}\ (\mathbf{K}\ e)\ f & \rightarrow (\mathbf{B}\ e)\ f \\ \mathbf{S}\ e\ (\mathbf{K}\ f) & \rightarrow (\mathbf{C}\ e)\ f \end{array}$$

The optimizations must be applied in the order given.

Just as machine language (assembler) can be used for programming, combinatorial logic can be used as a programming language. The programming language FP is a programming language based on the idea of combinatorial logic.

2 Scheme

Scheme, a descendent of LISP, is based on the lambda-calculus. Although it has imperative features, in this section we ignore those features and concentrate on the lambda-calculus like features of Scheme. Scheme has two kinds of objects, **atoms** and **lists**. Atoms are represented by strings of non-blank characters. A list is represented by a sequence of atoms or lists separated by blanks and enclosed in parentheses. **Functions** in Scheme are also represented by lists. This facilitates the creation of functions which create other functions. A function can be created by another function and then the function applied to a list of arguments. This is an important feature of languages for AI applications.

Syntax

The syntax of Scheme is similar to that of the lambda calculus.

Scheme Syntax

```
E in Expressions
A in Atoms ( variables and constants )
...
E ::= A | (E...) | (lambda (A...) E) | ...
```

Expressions are atoms which are variables or constants, lists of arbitrary length (which are also function applications), lambda-abstractions of one or more parameters, and other built-in functions.

Scheme provides a number of built in functions among which are +, -, *, /, <, <=, =, >=,>, and not. Scheme provides for conditional expressions of the form (if E_0 E_1 E_2) and (if E_0 E_1). Among the constants provided in Scheme are numbers, #f and the empty list () both of which count as false, and #t and any thing other than #f and () which count as true. nil is also used to represent the empty list.

Definitions

Scheme implements definitions with the following syntax

E ::= ...| (define I E) | ...

Lists with `nil, cons, car` and `cdr`

The list is the basic data structure with `nil` repesenting the empty list. Among the built in functions for list manipulation provided in Scheme are `cons` for attaching an element to the head of a list, `car` for extracting the first element of a list, and `cdr` which returns a list minus its first element.

Figure N.10: **Stack operations in Scheme**

```
( define empty_stack
   ( lambda ( stack ) ( if ( null? stack ) \#t \#f )))

( define push
   ( lambda ( element stack ) ( cons element stack ) ))

(define pop
   ( lambda ( element stack ) ( cdr stack )))

(define top
   ( lambda ( stack ) ( car stack )))
```

Figure N.10 contains an example of stack operations writtem in Scheme. The figure illustrates definitions, the conditional expression, the list predicate `null?` for testing whether a list is empty, and the list manipulation functions `cons, car,` and `cdr`.

Local Definitions

Scheme provides for local definitions with the following syntax

Scheme Syntax

...
B in Bindings
...

E ::= ...| (let B_0 E_0) | (let* B_1 E_1) | (letrec B_2 E_2) |...
B ::= ((I E)...)

The `let` definitions are done independently of each other (collateral bindings), the `let*` values and bindings are computed sequentially and the `letrec` bindings are in effect while values are being computed to permit mutually recursive definitions.

3 ML

4 Haskell

In contrast with LISP and Scheme, Haskell is a modern functional programming language.

Figure N.11: **A sample program in Haskell**

```
module AStack( Stack, push, pop, top, size ) where
data Stack a = Empty
             | MkStack a (Stack a)
push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Integer
size s = length (stkToLst s) where
```

```
                  stkToLst Empty         = []
                  stktoLst (MkStack x s) = x:xs where xs = stkToLst s

      pop :: Stack a -> (a, Stack a)
      pop (MkStack x s) = (x, case s of r -> i r where i x = x)

      top :: Stack a -> a
      top (MkStack x s) = x
```

```
module Qs where

qs :: [Int] -> [Int]
qs [] = []
qs (a:as) = qs [x | x <- as, x <=a] ++ [a] ++ qs [x | x <- as, x> a]
```

```
module Primes where

primes :: [Int]
primes = map head (iterate sieve [2 ..])

sieve :: [Int] -> [Int]
sieve (p:ps) = [x | x <- ps, (x `mod` p)=0]
```

```
module Fact where

fact :: Integer -> Integer
fact 0 = 1
fact (n+1) = (n+1)*fact n -- * "Foo"
fact _ = error "Negative argument to factorial"
```

```
module Pascal where

pascal :: [[Int]]
pascal = [1] : [[x+y | (x,y) <- zip ([0]++r) (r++[0])] | r <- pascal] tab :: Int -> ShowS
tab 0 = λx -> x
tab (n+1) = showChar ' ' . tab n

showRow :: [Int] -> ShowS
showRow [] = showChar '\n'
showRow (n:ns) = shows n . showChar ' ' . showRow ns

showTriangle 1 (t:_) = showRow t
showTriangle (n+1) (t:ts) = tab n . showRow t . showTriangle n ts
```

```
      module Merge where

      merge :: [Int] -> [Int] -> [Int]
      merge [] x = x
      merge x [] = x
      merge l1@(a:b) l2@(c:d) = if a < c then a:(merge b l2)
                                    else c:(merge l1 d)

      half [] = []
      half [x] = [x]
      half (x:y:z) = x:r where r = half z

      sort [] = []
      sort [x] = [x]
      sort l = merge (sort odds) (sort evens) where
                   odds = half l
                   evens = half (tail l)
```

5 Historical Perspectives and Further Reading

In the 1930s Alonso Church developed the lambda-calculus as an alternative to set theory for the foundations of mathematics and Haskell B. Curry developed conbinatory logic for the same reason. While their goal was not realized, the lambda-calculus and combinators capture the most general formal properties of the notion of a mathematical function.

The lambda-calculus and combinatory logic are abstract models of computation equivalent to the Turing machine, recursive functions, and Markov chains. Unlike the Turning machine which is sequential in nature, they retain the implicit parallelism that is present in mathematical expressions.

The lambda-calculus is a direct influence on the programming language LISP, the *call by name* parameter passing mechanism of Algol-60, and textual substitution performed by macro generators.

Explicit and systematic use of the lambda-calculus in computer science was initiated in the early 1960s by Peter Landin, Christopher Strachy and others who started a formal theory of semantics for programming languages called *denotational semantics*. Dana Scott (1969) discovered the first mathematical model for the type-free lambda-calculus.

New hardware designs are appearing to support the direct execution of the lambda-calculus or combinators which support parallel execution of functional programs, removing the burden (side-efficts, synchonization, communication) of controlling parallelism from the programmer.

LISP (LISt Processing) was designed by John McCarthy in 1958. LISP grew out of interest in symbolic computation. In particular, interest in areas such as mechanizing theorem proving, modeling human intelligence, and natural language processing. In each of these areas, list processing was seen as a fundamental requirement. LISP was developed as a system for list processing based on recursive functions. It provided for recursion, first-class functions, and garbage collection. All new concepts at the time. LISP was inadvertantly implemented with dynamic rather than static scope rules. Scheme is a modern incarnation of LISP. It is a relatively small language with static rather than dynamic scope rules. LISP was adopted as the language of choice for artificial intelligence applications and continues to be in wide use in the aritficial intelligence community.

ML

Miranda

Haskell is a modern language named after the logician Haskell B. Curry, and designed by a 15-member international committee. The design goals for Haskell are have a functional language which incorporates all recent ``good ideas'' in functional language research and which is suitable for for teaching, research and application. Haskell contains an overloading facility which is incorporated with the polymorphic type system, purely functional i/o, arrays, data abstraction, and information hiding.

Functional programming languages have been presented in terms of a sequence of virtual machines. Functional programming languages can be translated into the lambda-calculus, the lambda-calculus into combinatorial logic and combinatorial logic into the code for a graph reduction machine. All of these are virtual machines.

Models of the lambda-calculus.

History \cite{McCarthy60} For an easily accessable introduction to functional programming, the lambda-calculus, combinators and a graph machine implementation see Revesz (1988). For Backus' Turing Award paper on functional programming see \cite{Backus78}. The complete reference for the lambda-calculus is \cite{Bare84}. For all you ever wanted to know about combinatory logic see \cite{CF68,CHS72,HS86}. For an introduction to functional programming see Henderson (1980), BirdWad88, MLennan90. For an intoduction to LISP see \cite{McCarthy65} and for common LISP see \cite{Steele84}. For a through introduction to Scheme see \cite{AbSus85}. Haskell On the relationship of the lambda-calculus to programming languages see \cite{Landin66}. For the implementation of functional programming languages see Henderson (1980) and Peyton-Jones (1987).

Henderson, Peter (1980)
Functional Programming: Application and Implementation Prentice-Hall International.
Peyton-Jones, Simon L (1987)
The Implementation of Functional Programming Languages Prentice-Hall International.
Revesz, G. E. (1988)
Lambda-Calculus, Combinators, and Functional Programming Cambridge University Press.

6 Exercises

1. [Time/Difficulty](section)
2. Simplify the following expressions to a final (*normal*) form, if one exists. If one does not exist, explain why.
    1. ((λx. (xy))(λz.z))
    2. ((λx. ((λy.(xy))x))(λz.w))
    3. ((((λf.(λg.(λx.((fx)(gx)))))(λm.(λn.(nm))))(λn.z))p)
    4. ((λx.(xx))(λx.(xx)))
    5. ((λf.((λg.((ff)g))(λh.(kh))))(λy.y)))
    6. (λg.((λf.((λx.(f(xx)))(λx.(f(xx)))))g))
    7. (λx.(λy.((-y)x)))45
    8. ((λf.(f3))(λx.((+1)x)))
3. Find a lambda-expression that not only does not have a normal form but grows in length as well.
4. In addition to the β-rule, the lambda-calculus includes the following two rules:

```
α-rule: (λx.E) ⇒ (λy.E[x:y])
η-rule: (\x.E  x) ⇒ E where x does not occur free in  E
```

   Redo the previous exercise making use of the η-rule whenever possible. What value is there in the α-rule?
5. The lambda-calculus can be used to simulate computation on truth values and numbers.

    1. Let **true** be the name of the lambda-expression λx. λy. x and **false** be the name of the lambda-expression λx. λy. y. Show that ((\mbox{true} E_1)E_2) ⇒ E_1 and ((\mbox{false} E_1)E_2) ⇒ E_2. Define lambda-expressions **not, and,** and **or** that behave like their Boolean operation counterparts.
    2. Let **0** be the name of the lambda-expression λx. λy. y, **1** be the name of the lambda-expression λx. λy. (xy), **2** be the name of the lambda-expression λx. λy. (x(xy)), **3** be the name of the lambda-expression λx. λy. (x;(x(xy))), and so on. Prove that the lambda-expression **succ** defined as λz. λx. λy.(x ((zx)y)) rewrites a number to its successor.

6. Recursively defined functions can also be simulated in the lambda-calculus. Let **Y** be the name of the expression λf. λx.(f(xx)) λx. (f(xx))
    1. Show that for any expression E, there exists an expression W such that (**Y**E) ⇒ (WW), and that (WW) ⇒ (E(WW)). Hence, (**Y**E) ⇒ E(E(E(...E(WW)...)))
    2. Using the lambda-expressions that you defined in the previous parts of this exercise, define a recursive lambda-expression **add** that performs addition on the numbers defined earlier, that is, ((**add**m)n) ⇒ m+n.
7. Let T = AA where A = λxy.y(xxy). Show T F = F (T F). T is Turing's fixed point combinator.
8. Data constructors can be modeled in the lambda-calculus. Let **cons** = (λa. λb. λf. f a b), **head** = (λc. c (λa. λb. a)) and **tail** = (λc. c (λa. λb. b)). Show that
    1. **head** (**cons** a b) = a
    2. **tail** (**cons** a b) = b
9. Show that (((S(KK))I)S) is (KS).
10. What is (((SI)I)X) for any formula X?
11. Compile (λx.+xx) to SKI code.
12. Compile λx. (F (xx)) to SKI code.
13. Compile λx. λy. xy to SKI code. Check your answer by reducing both ((λx. λy. xy) a b) and the SKI code applied to a b.
14. Apply the optimizations to the SKI code for λx. λy. xy and compare the result with the unoptimized code.
15. Apply the optimizations to the SKI code for λx. (F (xy)) and λy. (F (xy)).
16. Association lists etc
17. HOF
18. Construct an interpreter for the lambda calculus.
19. Construct an interpreter for combinatorial logic.
20. Construct a compiler to compile lambda expressions to combinators.
