ABSTRACT

We introduce PyFL (pronounced pie-full), a new full featured functional language. PyFL (Python based Functional Language) is designed to be accessible to "ordinary" programmers experienced only with conventional programming languages and mathematics. To that end it uses conventional mathematical notation—no currying. It is dynamically typed—no type declarations. And input/output is done unapologetically with side effects—no monads.

The basic data types of PyFL are essentially those of LISP—numbers, strings, words (like atoms) and finite hierarchical lists. In addition it has finite sets, infinite streams, structs, dictionaries, and of course functions. With a full set of infix and prefix operators (mainly the usual ones).

In PyFL all data objects are first class and can be combined at will. So, for example, we can have a list of dictionaries each of which associate keys with sets of functions.

PyFL also has a number of innovative features, among them variable binding operators, a very general pattern matching mechanism, and a simple form of declarative while loop. The variable binding operators are similar to the $\sum$ and $\prod$ operators of conventional mathematics, coded in ASCII. The case statement is based on an algebra of patterns. And the while loop allows iterative algorithms to be expressed by equations rather than commands.

PyFL has been implemented in about 7,000 lines of disciplined Python. We give an overview of the syntax and semantics of PyFL and of its implementation, and document the details of its various features.

CHAPTER 1

INTRODUCTION

Functional programming has definitely established itself as a practical alternative to imperative programming, thanks largely to the achievements of the Haskell project [Haskell]. Nevertheless, it still remains a minority or niche specialty that most programmers don't really understand. One reason, of course, is that it requires thinking of the solution to a problem in terms of functions that transform input into output, rather than commands which alter a state. But another reason is that functional languages, like Haskell, are very different from conventional imperative languages. One goal of the PyFL project is to show that this doesn't have to be the case.

One of the first things a typical programmer notices is the syntax of function application. Whereas in all the languages she knows she writes `f(a,b+c)`, in Haskell it must be `f a (b+c)`. A minor point, but is the difference really necessary? No, and PyFL uses the conventional syntax for functional definition and application:

```
I(f,a,b) = let
             m1 = f(a);
             m2 = f((a+b)/2);
             m3 = f(b);
           in
             (b-a)*(m1 + 4*m2 + m3)/6
           end;
```

More serious is the often complex and elaborate type declarations required for even small programs. Static typing can have enormous benefits, mainly in catching certain kinds of bugs and simplifying the implementation. But it adds a burden to the programmer and makes certain programs difficult or impossible. For example, we have to be quite clever if we want to have a heterogenous list — a list whose components are of different types. And other expressions, like self-application, are simply impossible, because untypeable. PyFL spares our programmer these considerations by using dynamic (runtime) type checking and eliminating any static declarations. In PyFL, `[33 dog 'hello' [1 2 3]]` is a valid list constant, and

```
Y(f) = g(g)
       where
```

```
        g(h) = f(h(h));
    end
```

is a valid (non-recursive) definition of the fixpoint combinator Y.

Our programmer is relieved to discover that I/O in, say, Haskell, is familiar. Output is done with primitives like `putString`, which are transparently commands that alter state. Yet relief is tempered with puzzlement when she learns this all must be done through "monads" ([Haskell-Monads]). And what is a monad? The answer involves category theory: a monad is a monoid in the category of endofunctors. So understanding even primitive programs that do I/O involves knowing some higher mathematics.

I/O in PyFL uses side effects without apology. At the moment we don't know how else to do it. The "function" call `output(X,'Value is:')` has the effect of printing "Value is: " followed by the value of `X`. Similarly, evaluating the expression `input('Number please:')` has the side effect of printing the prompt "Number please: ", reading the input given, and returning this input as the value of the expression. However we don't invoke category theory or any higher mathematics to pretend that this is "pure".

Up to now we have discussed mainly what PyFL doesn't have (currying, type declarations, monads). However our programmer will be glad to discover several very useful features, present in almost *no* other functional language; for example:

- The first, as we indicated, is self application and, more generally, any combinator of the untyped lambda calculus.
- The second is, as we have also seen, heterogeneous lists.
- The third is what we call variable binding operators. These correspond to the $\sum$ and $\prod$ operators of mathematics. They are in the spirit of list comprehensions but are more general and use a friendlier syntax. For example, if our programmer wants the Euclidean length of the list `C` of numbers she can write

  ```
  sqrt(sum x*x for x in C end),
  ```

  and if `S` is a list of sets,

  ```
  union s for s in S end
  ```

  gives the result of taking the union of all the components of `S`.

Finally, our programmer, skilled in imperative iterative constructs, will be surprised and relieved to discover that PyFL has a while loop (paradoxical though this may seem). One of the great weaknesses of functional languages is the lack of iterative constructs. Iterative algorithms have to be coded in terms of tail recursion. PyFL automates this approach. For example, to define a function to compute the Fibonacci numbers, our programmer can write

```
fib n =
  while k <= n
    k = 1 fby k+1;
    fib = 1 fby fib + prefib;
    prefib = 0 fby fib;
    result = fib;
  end;
```

This looks like PyFL is using the `fby` operator of Lucid [Lucid] (this is intentional) but actually `fby` is just a syntax word separating the expressions for the first and succeeding values

of the variable.

This concludes our overview, and justification for, the design decisions for the language. In the rest of the paper we will give an overview of the syntax and semantics of PyFL. We will also give an overview of the implementation and of the advantages of using Python. Finally we will describe possible future work, including type checking and compiling.

Running PyFL

Now we describe how users actually run programs. Obviously we don't want them to have to know anything about Python, other than for installing the software. Once python3 is installed and the PyFL files have been downloaded, we launch the IDE with the (unix) command `python3 ide.py` from the appropriate directory.

Once the IDE is launched we see the version number, a list of commands, and the identity of the "current module" or a declaration that there is none.

The IDE is based on a very simple notion of module. A PyFL module consists of three files: an `mn` file of definitions, including a definition of `result`; a `df` file of function definitions; and an `im` file which is a list of import modules. The user can load a module with the `m` command, and it becomes the current module. If there already was already a current module, the user is asked if they want to save it.

Once a module is loaded, we can edit it with the `v` command. The command `vm` invokes the `vi` editor to edit the `mn` file, `vd` the `df` file, and `vi` the `im` file. Actually, these commands don't edit the files directly. Each file has a buffer initialized to the corresponding file and it is the buffer which is edited. When the current module is changed, the user is asked if they want to save the current module. If the answer is yes, the buffers are copied back to the corresponding source files. Otherwise the buffers revert to the source files.

Once the files are edited we can execute the program with the `e` command. The interpreter computes the transitive closure of the `im` files to locate every module whose definitions are imported directly or indirectly. Then it produces a stand alone PyFL program, in the form of a `valof` clause, whose body includes the contents of the buffer of the current module's `mn` file and of the buffers for all the `df` files in the transitive closure.

The IDE does nothing to enforce namespaces; it is the programmer's responsibility to ensure, for example, that if the current module's `df` file has a definition for a function `foo`, the same does not also hold for one of the imported `df` files.

The IDE has a number of commands, but `m`, `v`, and `e` are the main ones.

Typographic Conventions

→Say things about what fonts mean

PyFL supports single-line, as well as multiline comments. Both variants follow the syntax of C. Everything after `//` to the end of line is ignored. Multiline comments are pieces of text delimited by `/*` and `*/`; note that they *cannot* be nested: comments started by `/*` end after the *first* occurrence of `*/`.) In this manual, we represent comments by writing them in an italic monospace font, *`just like this`*.

CHAPTER 2

DATA TYPES

PyFL adopts (with minor modifications) the data types, operators, and syntax of POP-2 ([POP-2]), a long dead AI language developed at Edinburgh (POP-2 lives on in POP-11 [POP-11]). PyFL has essentially the data types of LISP. It has numbers, strings, *words* (like atoms), hierarchical finite lists. In addition it has *infinite streams*, structs and dictionaries, and functions as first class objects, which we wil discuss in Chapter 4, Chapter 6, Chapter 7, and Chapter 8.

2.1 Numbers

PyFL understands basically only two number types: integers and floats, for which it provides a number of built-in operations. PyFL provides functions `isint` and `isfloat` which check if their argument is an integer or a floating-point number. To check if something is a number more broadly, use `isnum`.

For the basic arithmetic operations, the usual infix operators `+`, `-`, `*`, `/`, and `**` (exponentiation) are available, with the usual operator precedences. All of those except division operate on both integers and floats, and the result is integer if (and only if) both operands are integers, and float otherwise. Division, however, produces a float, regardless of the type of the operands, even if the quotient is an integer in the mathematical sense. Needless to say, one should avoid dividing by zero.

PyFL ships with functions for other widely used numerical operations, such as `abs` for absolute value, `sin`, `cos`, and `tan` for the usual trigonometric functions, `exp` for exponentiation and `log` for the logarithm (both with base e), as well as variants `log2` and `log10` for base-2 and base-10 logarithms, and `sqrt` for the square root; all operate on floats. PyFL also knows about the constants `pi` and `phi`.

There are also specialized operations for integers, which are especially useful for doing number theoretic computations; the most important are `mod` and `div`, which compute the remainder and quotient of two the Eucleidean division. Following the usual mathematical convention, they are *infix*: `44 div 3` is 14, and `12 mod 5` is 2.

One function is designed to produce integers from floats: `floor` returns the largest integer less than or equal to its argument. Using it, we can define the dual function

```
ceiling(x) = - floor(-x)
```

`floor` can also be used to explain `div` and `mod`; both computations evaluate to `"true"`:

```
a div b == floor(a/b);      a mod b == a - b*floor(a/b).
```

Numbers can be compared for equality with `==`, and for inequality with `<` and `>`. Following the usual programming conventions, `<=` and `>=` stand for the usual mathematical relations $\leq$ and $\geq$. Those are *predicates*, i.e. they return Boolean values `"true"` and `"false"`. The binary functions used for comparisons are `max` and `min`, with the obvious meanings.

Finally, it is worth saying a few things about the representation of numbers: floating-point numbers have a finite precision, but integers have *arbitrary precision*; this means that in PyFL we can safely write programs handling *arbitrarily big numbers* with no fear of overflows (other than those caused by exhaustion of the computer's memory). For example,

```
a(6)
  where
    a(n) = A(n,n)
    A(m,n) = if m==0 then n+1 else A(m-1,A(m,n-1)) fi
  end
```

is a PyFL program which will happily compute the value of the Ackermann function at 6 (on a computer with enough memory).

2.2 Characters & Strings

Many programs need to process text; accordingly PyFL provides functions to do text-processing

Text is built of *characters*, i.e. letters, digits, punctuations, etc. Putting characters in a row gives a string. PyFL takes a rather simplifying approach to the matter, by treating characters and strings as objects of the same type—there is only one built in data-type for text: strings. Characters are just strings of length one.

Strings are a basic (ground) data-type. String literals are created as sequences of text enclosed in single quotes (`'`): `'This is a PyFL book'`. PyFL provides the function `isstring` to check if something is a string. Thus, `ischar`, which checks if something is a character, could be defined as

```
ischar(x) = isstring(x) and length(s)==1
```

The infix operator "`+`" denotes string concatenation, which is arguably the main function on strings. There is also another function `charlist`, which returns a list of the characters in a string.

What is the point of `charlist`? Well, to allow string-processing programs:

```
//  Example goes here;
//  convert string to uppercase with reduce/fold.
```

2.3 Words

In PyFL, *words* are basically symbols; they are a conventient way to define "tags" with symbolic names, without worrying about having to find an implementation for those tags. We have already met a word: the `"true"` value.

Informally, words are sequences of characters enclosed in double quotes ("), subject to certain constraints. Formally, a word can be any of the following:

(a) an *identifier*: a letter followed by a mix of letters and numbers.

(b) a *sign*: a string composed of characters in the string "`@!+-*/=<>#:&^_`".

(c) *punctuation*: a single character (string of length one) from the string "`"(),;?`".

Generally, user-defined words for most applications will be of the first category, and it is bad practice to use other kinds of words unless absolutely needed for specific tasks. Note that in all cases, words must be enclosed in double quotes, and violating the constraints (as in `"foo("`) is a syntax error.

Apart from hard-coding them as word literals, PyFL allows another way to define words. The function `word` takes an argument (which must be a string) and returns that argument as a word: `word('this')` returns the word `"this"`. Complementarily, PyFL provides the function `isword` which checks if its argument is a word.

Words are helpful because we don't have to understand (as language users) how they are implemented—for us, a word is just an atomic object, pretty much like the numerical literal `1` or the string `'this'`, which *evaluates to itself*. This is quite unlike variables, which have values *assigned* to them. Words play a big role in PyFL's design philosophy, and thus it is essential to understand them well.

PyFL uses a word (`"true"`) to indicate truth; another word is used to represent falsehood: `"false"`. What's important to understand is that there is nothing special about the implementation of `"true"` and `"false"`—we could just as well have come up with our own words for truth and falsehood and used those instead of the defaults.

Because we can create our own words at will, words can be used for the short, convenient representation of specific situations (or states) of the program that we otherwise would have to represent by incomprehensible tricks or elaborate hacks. As a simple example, suppose we are given a variable representing the mood of the programmer:

```
if mood == "bad"
  then CheerUp()
  else if mood == "moderate"
          then DoBetter()
          else KeepWell() // presumably,
                          // now mood == "good"
       fi
fi
```

In similar cases, other languages would force programmers to use strings (as in Python or JS) or manually declare symbolic names which resolve to values (e.g. as with enumerations in C); in PyFL there is no such need, and words are more efficient than strings (because internally they are atomic objects), and require no setup (in particular, you need not "declare" in advance the words you intend to use).

2.4 Lists

PyFL lists are very much like lists in any other language. Lists are finite, ordered sequences of arbitrary data objects. There are no constraints on the nature of the elements of a list: they can be of any type, including sublists, and the types can be mixed. The one, big deviation from

other languages is that PyFL has two syntactic forms for creating lists: *list literals*, and *list expressions*.

The most important list operators are head (`hd`), tail (`tl`), cons (`::`), and append (`<>`). If L, L' are lists and x is any object, the head of L is its first element, the tail of L is a list like L without its first element, the cons (short for "construction") of x on L (which we would write as x`::`L) is the list with head x and tail L, and L`<>`L' is the list obtained by juxtaposing L and L' in that order; `<>` can be implemented in terms of the other operations. Like all PyFL operations, these always return new objects.

There are also operations `el2`, `el3`, `el4` for the second, third, and fourth element of a list, as well as a generic `elt` for accessing any element specified by its ordinal position in the list.

A list literal consists of a sequence of individual tokens *separated by spaces* and delimited by "`[`" and "`]`"; for example,

```
[ 25.3 'cats and dogs' Amy {p q} [8 9 ['end']] ( + a b ) ].
```

Lists literals are called so because their elements are (constant) *literals*; this means that *nothing in a list literal is evaluated*. Every one of the data-types we've seen so far has literals; examples are `"word"` for words, `'a string'` for strings, and `13` for numbers. Hence, `[3+4]` is a list with three elements, which are in order `3`, `+`, and `4`, and accordingly `hd [3+4]` will return 3.

In particular, words appearing in a list literal *are not* enclosed in double quotes; if they are, the meaning of the list changes. For example, `["word"]` is a word of three tokens: `"`, the token `word` (which is a word), and finally `"` again; thus, if we try to extract the first and second elements of this list, the results will be `"""` and `"word"`. This is a consequence of the word constraints, in particular item (c).

When given a list literal, PyFL tries to read the tokens of the list so that they make sense, and in doing so, it follows the syntax constraints for the literals of the several data-types—the previous examples are special cases of that. The POP-2 convention used by PyFL to read constant lists has the advantage that when a program is enclosed in square parentheses, the result is a valid list containing the *lexical tokens* of the program. Spaces are not necessarily required. For example, the constant list `[foo(bar,'dog')]` is equivalent to the constant list `[foo ( bar , 'dog' )]`. (This list contains six elements, of which all but `'dog'` are words.) This is one reason why words are not simply strings.

There are also list expressions: those involve the evaluation of expressions. PyFL uses decorated square brackets ("`[%`" and "`%]`") enclosing a *comma-separated* list of expressions, as in `[%23+56.2, "halt", (a+b)/2,f,g, 'Greetings '+'Tom'%]`. Notice that the word "halt" is double-quoted now to prevent PyFL from trying to evaluate the (possibly undefined) variable `halt`.

List literals and expressions can be nested, but the rule is that anything inside a list literal will be treated verbatim. Thus, `[result is [%3+4%]]` will give the seven-element list

```
[result is [% 3 + 4 %]],
```

and not `[result is [7]]` as one might expect. (List literals are basically like the `quote` operator of LISP.)

2.5 Sets

Sets are similar to lists except that they use curly brackets `{` and `}`. They too come in a constant and an expression version, the latter using `{%` and `%}` for delimitation. Sets are unordered and

have no duplicates. Set union is "+", difference "−", and intersection " ^ "; `isset` checks if its argument is a set.

Thus `{p q r}` its a set containing the words `"p"`, `"q"`, and `"r"`, and `{%10+3, 10-3, 7%}` is a set whose elements are 13 and 7.

2.6 Other Data Structures

PyFL also supports *structs*, *dictionaries*, and *streams*.

Structs are like structures in C, or Records in Haskell; they pack together values of other types; in PyFL, each struct is its own type.

Dictionaries are basically hash tables, i.e. data structures that map keys to values. With their high efficiency, they are one of PyFL's most fundamental data structures.

Finally, streams are essentially infinite lists. For example, we can define

```
Nats = 0 join bump(Nats)
```

for the stream of natural numbers. Of course, `Nats` will not be computed all at once—only the elements of the stream that are directly used in the program will be actually computed.

~~Complexity of operations on Lists? On Sets?~~→**Complexity of operations on Lists? On Sets?**

2.7 NOTES, REFERENCES, AND PROBLEMS

2.7.1 Add notes, references, and problems here.

2.7.2 **Problem:** Why do you think the designer of the language added no constant for e?

2.7.3 **Problem:** What are the elements of the list `[a3ok+4]`? And what are those of the list `['strings'=type]`?

2.7.4 **Problem:** Write examples to find out what are the list tokenization rules.

2.7.5 **Problem:** Describe how to implement append (`<>`) using the other list operations.

CHAPTER 3

CONSTRUCTS

PyFL has, besides prefix and infix operators, several constructs. The most important of these encapsulate definitions. The simplest such construct (which we haven't yet seen) is the `valof` block, on top of which all others build.

3.1 The "valof" block

A `valof` block consists of a set of definitions between the keywords `valof` and `end`. No two definitions can define the same variable, and there must be exactly one definition for the variable `result`, which is the value returned from the block. For example,

```
valof
  s = a**2 + b**2;
  result = sqrt(s);
end
```

will compute the square root of two numbers.

The value of a `valof` block is the value of `result` in the context of the enclosed definitions. This means that in evaluating a `valof` block we begin by trying to evaluate `result`. This leads us to evaluating other variables and we do so by looking up their definitions and evaluating the right hand sides. If we need the value of a variable not defined in the `valof` block, we use the definitions in the enclosing blocks, *proceeding from inside to out.* Notice that the value of a `valof` block is independent of the order in which its definitions appear.

3.2 Local assignments: where and let

The `where` and `let` clauses are just friendlier interfaces for `valof`.

The `where` clause consists of an expression, called the *subject*, followed by the keyword `where`, followed by a set of definitions (subject to the rules that there must be no two definitions of the same variable, and that there is no definition of the keyword `result`), followed by the word `end`. A `where` clause is equivalent to a `valof` clause enclosing its definitions, extended by a definition equating `result` to its subject.

The other type of definitional construct, `let` consists of the word `let` followed by a set of definitions subject to the same constraints as `where`, followed by the word `in`, followed by

a *subject* expression, followed by the word `end`:

```
let a = 1
    b = 6
in  a**b
end;
```

The value of a `let` clause is the value of the subject expression given its definitions and those of enclosing blocks, as with a `where` clause.

Neither `where` nor `let` check if their definitions are unique; if a definition occurs twice, one of the two values will be used, but the behaviour is unspecified.

3.3 Conditionals

Arguably, the most important construct in any language is the conditional. An `if` clause consists of the keyword `if`, followed by a sequence of test/result pairs, followed by the keyword `else`, followed by an expression, then the keyword `fi`. A test/result pair is two expressions separated by the keyword `then`. Consecutive pairs can optionally be separated by the keyword `elseif`. For example:

```
if
  disc < 0  then 'no real root' elseif
  disc == 0 then 'double root'
else
  'two real roots'
fi
```

The first expression of every test/result pair must be boolean expressions, i.e. they must evaluate to `"true"` or `"false"`; if they don't, a run time error will be triggered.

3.4 Variable Binding Operators

In conventional mathematics, iterative computations, like summing the elements of an array, are very common, and mathematicians have developed a concise and elegant syntax to specify such calculations. We are talking about the $\sum$ notation. On paper, the notation is two dimensional but we can identify its components: a bound variable (say, i); a range (say, 0 to n); a predicate restricting the range (say, $a[i] > 0$), and an expression to be summed (say, $a[i]^3$). PyFL supports a linear ASCII syntax for such expressions, for example

```
sum a(i)**3 for i in range(0,n): a(i)>0 end
```

The predicate is optional, and can be omitted if all elements are to be taken:

```
sum j*j for j in range(1,11) end
```

PyFL also has a linear textual form of the $\prod$ construct, given by the `prod` keyword, as in

```
prod (1+1/i) for i in range (2,11): prime(i) end
```

In total, there are 9 more variable binding operators given by the following keywords:

`cons`	create a list from the values of the expression over the range;
`append`	append the values of the expression (which should be lists);
`concat`	concatenate the values of the expression (which should be strings);

`collect`	join the values of the expression into a set;
`union`	take the union of the values (which should be sets);
`intersect`	take the intersection of the values (which should be sets);
`avg`	take the average of the values (which should be numbers);
`max`	take the maximum of the values (which should be numbers);
`min`	take the minimum of the values (which should be numbers).

There are also four simpler constructs whose meaning should be obvious:

```
exist i in range(0,10): a(i) > 0 end;
forall j in range(0,10): a(j) < 5 end;
those k in L: a(k) == 0 end;
first m in L a(m) > 0 end;
```

The range expression of these constructs does not have to be a list. It can be a set, a string or even a single integer. If it's a set, the variable runs through the elements of the set in an unspecified order. If it is a string, it runs through the characters of the string, a character being a list of length one. Finally, if it is a single number n, it runs through `range`$(0, n)$.

Here is the definition of a function `perms` which generates a list of the permutations of a list (without repetitions).

```
perms(s) =
if s==[] then [[]] else
  append
    dist(hd(s),pr)
  for pr in  perms(tl(s)) end
fi;

dist(c,s) =
if s == [] then [%[%c%]%]
           else
        (c :: s) ::
        cons
          hd(s) :: t
        for t in dist(c,tl(s)) end
fi;
```

3.5 NOTES, REFERENCES, AND PROBLEMS

CHAPTER 4

FUNCTIONS

PyFL is a functional language and functions are the fundamental data type. Functions in PyFL transform input into output and nothing else — there are *no side effects*.

The only exception is the I/O primitives and entities defined in terms of them, which have the side effects of reading and writing and printing prompts. We do not pretend they are pure and do not invoke higher mathematics to claim otherwise. In particular we do not use monads.

Functions in PyFL are first class objects and can be passed to other functions and returned by other functions. We can have lists, sets and streams of functions. When functions appear in sets they are presumed to be distinct, even though they may be equal in the mathematical sense — after all, function equality is undecidable and PyFL can't be expected to possess superhuman powers.

4.1 Defining Functions

~~Functions can be defined two ways, first by definition but also using lambda expressions. A function definition uses conventional mathematical notation:~~ → **The most basic way to define functions is by lambda expressions; an alternative, but equivalent, notation is the usual mathematical notation:** the left hand side consists of a function variable followed immediately by a comma-separated list of formal parameters enclosed in round parentheses:

```
f(a,b) = a**2 + b**2;
```

The formal parameters must be distinct variables. PyFL has pattern matching, ~~but not in function definitions~~ → **but it doesn't work as in Haskell:** in PyFL, in any given scope, each function can have exactly one defining equation. Function definitions can be recursive.

The right hand side of a function definition can be arbitrary. When the interpreter evaluates a function call, it checks that the expression being called actually evaluates to a function, and that it is being given the correct number of arguments.

We have described functions as input-output transformations. For this to be accurate, we have to be very careful how we evaluate function applications. The rule is that, for example, if the definition of `f` is of the form `f(x,y)=`$\mathcal{D}$ and the call is `f(`$\mathcal{A}$`,`$\mathcal{B}$`)`, then we evaluate the body $\mathcal{D}$ of `f` in the defining environment, augmented by definitions of `x` and `y` as the results of evaluating $\mathcal{A}$ and $\mathcal{B}$ respectively in the calling environment. This means PyFL uses *static binding* ([Static-Binding]). For example, the value of

```
a where g = 10; f(x) = x+g; a = f(5) where g = 20; end end
```

is 15, not 25, which dynamic binding would give us.

A function can be defined directly in terms of an expression which returns a function value. For example,

```
f = lambda (a,b) a**2 + b**2 end;
```

More elaborate examples are possible, e.g.

```
f = g(0)
    where g(c) = lambda (a,b) a**2 + b**2 + c**2 end; end;
```

4.2 Higher-Order Programming

CHAPTER 5

FUNCTIONAL ITERATION

One of the perceived weaknesses of functional languages is their inability to express iterative algorithms. Variable binding operators help to some extent, because the computations they specify (like adding up the elements of a list) are clearly iterative. But they are not very general — for example, they can't be used to generate the Fibonacci numbers.

Iteration seems to require an imperative framework; we apparently need commands to repeatedly update loop variables. But the Lucid language embodies a different approach, where for each loop variable we declare the initial value and the updating expression. PyFL adopts a simple but general version of this declarative approach to iteration.

A PyFL `while` clause consists of the word `while`, a condition (expression that should evaluate to a boolean), ~~a set of ordinary or definitions of first/next pairs,~~ →**meaning unclear here**, a definition of the variable result, and finally the word `end`. The clause defines an iteration that proceeds as long as the condition is true, then halts and returns the current value of the variable with name the keyword `result`. We've already seen a while-loop, used to calculate the n-th Fibonacci number. The `while` clause uses `fby` not as an operator, but rather as a syntax word separating the first and next expressions.

An `until` clause is like a `while` clause except there are *two* conditions and an extra definition of a variable `result1`. The conditions are halting conditions, and the value of `result` is returned when the first condition is the first to be true. When the second condition is the first to succeed, the current value of `result1` is returned. Here is an `until` clause that computes the square root of n using Newton's method:

```
until err < 0.0001, k > 10
  k = 1 fby k+1;
  a = 1 fby (a+n/a)/2;
  err = abs(n-a*a);
  result = a;
  result1 = "nontermination";
end
```

CHAPTER 6

STREAMS

Lists in PyFL are finite and eager; by this we mean that the cons function is strict: it evaluates its arguments. (We made this choice for performance reasons, to avoid evaluating closures.) However, by making lists eager rather than lazy, we lose the possibility to have infinite lists, such as are available in other functional languages. To remedy this gap we introduced *streams*, essentially one-dimensional lazy lists.

A stream is an infinite sequence of data items. There are no stream constants — all streams one intends to use must be constructed, and there are three operators for that task: `first`, `rest`, and `join` (abbreviated `fs`, `rs`, and `jn`). They are essentially the stream counterparts to the list operators `hd`, `tl`, and `cons`, respectively. For example, the recursive definition

```
Ones = 1 join Ones
```

defines `Ones` to be the stream $\langle 1, 1, 1, \ldots \rangle$.

Next we define a function `bump` that takes a stream and returns the stream with 1 added to each component. The function is defined recursively, and is actually quite intuitive:

```
bump(S) = 1+first(S) join bump(Rest(S))
```

Then `Nats = 0 join bump(Nats)` defines the stream $\langle 0, 1, 2, \ldots \rangle$ of natural numbers.

We can continue in this way defining more elaborate streams but the details are tedious. To avoid this we introduced a special `stream` clause. Syntactically it is like a `while` clause without a termination condition. Its value is the sequence of values of `result` that are not equal to the keyword `none`. For example:

```
Nats =
  stream
    i = 0 fby i+1
    result = i
  end
```

and

```
Squares =
  stream
    i = 0 fby i+1
    result = i*i
  end
```

And, borrowing from an earlier example,

```
Fib =
  stream
```

```
    f = 1 fby f+pf
    pf = 1 fby f
    result = f
  end
```

produces the stream $\langle 1, 2, 3, 5, 8, 11, \ldots \rangle$ of Fibonacci numbers.

The exception for `none` allows us to skip values. For example,

```
L =
  stream
    i = 0 fby i+1
    d5 = i mod 5 == 0
    result = if d5 then "none" else i fi
  end
```

defines `L` to be the stream $\langle 1, 2, 3, 4, 6, 7, 8, 9, 11, \ldots \rangle$ of numbers *not* divisible by 5.

Finally, suppose that `isprime(`n`)` is true iff n is prime; then

```
Primes =
  stream
    n = 2 fby n+1
    result = if isprime(n) then n else "none" fi
  end
```

produces the stream $\langle 2, 3, 5, 7, 11, \ldots \rangle$ of all primes.

We can't print streams because they are infinite; but we can print initial segments with a function like

```
seg(S,n) = if n<= 0
             then []
             else first(S) :: seg(rest(S), n-1)
           fi
```

Then with `Primes` defined as above, `seg(Primes, 1000)` is the (finite) list of the first 1000 primes.

CHAPTER 7

STRUCTS

Almost every programming language has some form of struct. A struct is a record with named components. For example, in most languages you can declare something like

```
struct Person {
    name: string
    surname: string
    dob: date
    job: occupation
}
```

for the packaging together of four values (to be referred to as `name`, `surname`, `dob`, and `job`) in a single "package" called `Person`; the values packed together have the corresponding types (here `date` and `occupation` are also structs, declared elsewhere).

PyFL does not copy this approach because it involves mandatory type declarations, and in particular those of the worst kind, namely those which contribute to the semantics of the program. In other words, they can't be omitted, because the program doesn't make sense without them. Mandatory type declarations are a headache for the programmer who has to come up with them before writing any code, and they're a headache for the implementer who has to parse and store them and extend the evaluator to deal with them.

Yet, basically every other language with structs requires some form of declaration. Instead we take inspiration from PyFL's treatment of lists. PyFL has lists but no list declarations. It has various operations that produce lists, such as `cons` and `append`. Also it has list constants like `[dog 3 'dick']`. The lack of declarations has one obvious advantage, namely it allows heterogeneous lists, one (like the constant just given) that has elements of different types.

Adopting this approach to structs means specifying operations which produce structs—an algebra of records. We call these "dynamic" structs.

7.1 Struct Expressions

The main construct expects expressions for the components, and places their values in the structure components labelled by the given component names. The keywords `<<` and `>>` delimit the construct, so, for example, `<< xcoord:3 ycoord:3+1 angle:pi/4 >>` is the

way to define a struct.

The fieldnames must be constants, not expressions. Structs can contain structs; e.g.

```
<< name: 'Karen' surname: 'Wilson'
   dob: << year:2025-30 month:mon(6) day:4*7 >>
   job: "teacher"
>>
```

The operation to access components is almost universally denoted by the dot operator ("."), and PyFL follows suit. If `K` denotes the struct above, then `K.name` is '`Karen`' and `K.dob.year` is 1995—notice that the dot operator is left-associative.

Finally, there is a not-so-obvious function for combining structs. It is callled `xby`, for "extended by". Given struct expressions $\mathcal{S}$ and $\mathcal{T}$, $\mathcal{S}$ `xby` $\mathcal{T}$ is $\mathcal{T}$ with missing values inherited from $\mathcal{S}$. More precisely, the set of fieldnames of $\mathcal{S}$ `xby` $\mathcal{T}$ is the union of the sets of fieldnames of $\mathcal{S}$ and $\mathcal{T}$; and given any fieldname n, the value of $\mathcal{S}$ `xby` $\mathcal{T}$ at n is that of $\mathcal{T}$ at n if $\mathcal{T}$ has n as a fieldname, or that of $\mathcal{S}$ at n otherwise.

~~There are also struct constants, which we won't cover here.~~ →**Do cover them here!**

7.2 Function Components

We allow function definitions inside structs. It means having function components, and most of the languages with structs allow it. For example in (PyFL)

```
<< xcoord:3
   ycoord:4
   degrees: lambda (r) 360*r/(2*pi) end
>>,
```

the `degrees` component is a function (which converts radians to degrees). Then if we write

```
f(1.57)
 where
  f = C.degrees
  C =
    << xcoord:3
       ycoord:4
       degrees: lambda (r) 360*r/(2*pi) end
    >>
 end,
```

we get 90 as the result.

For notational simplicity, we allow the definition of function components as normal function; thus in the example above we could have written:

```
degrees(r): 360*5/(2*pi),
```

and the result would be the same.

7.3 Self-reference

However function components are not very interesting if they can't refer to other components of the same structure. The variable `self` refers to the current structure. We can write

```
<< xcoord:3
   ycoord:4
   dist(): sqrt(self.xcoord**2+self.ycoord**2)
>>
```

and if `D` is the above struct, `D.dist()` evaluates to 5.

These simple features work well together. For example, if we added a field `upd()` to the struct `D` with value

```
upd(): self xby << xcoord:6 >>,
```

then `D.upd()` is a new struct which is identical to `self` except the `xcoord` filed is 6; the `self` would refer to the current structure, which in this case is `D`. Like all PyFL functions, the combination of `self` and `xby` creates a new struct—not equal to `self`.

Using this idea, here is a more elaborate structure `E` for representing points:

```
<<
 xcoord:0
 ycoord:0
 left(dx): self xby << xcoord: self.xcoord+dx >>
 up(dy): self xby << ycoord: self.ycoord+dy >>
 dist(): sqrt(self.xcoord**2+self.ycoord**2)
>>.
```

`E` provides five components, of which three are functional components! (They move a point vertically or horizontally, and compute its distance from the origin.)

~~Then `E.up(y)` is `E` with `ycoord` set to `y` (overriding the current value).~~ →**Better: [Then `E.up(y)` is `E` with `ycoord` icremented by `y`].** Similarly, ~~`F.left(x)` is `F` with the `xcoord` set to `x`~~ →**Better: [`F.left(x)` is `F` with the `xcoord` incremented by x]**. As a result, `E.up(4).left(3).dist()` evaluates to 5.

The combination of struct features already gives a reasonable analog of OO as found in imperative languages—no need to extend the language. Structs are analogous to objects. Struct components with data values, like `xcoord` and `ycoord`, correspond to class variables. Struct components with function values correspond to methods.

Accessing components with function values looks exactly like invoking methods, e.g. `E.up(4)`. The main difference is that, since structs (and all other PyFL data objects) are immutable, `E` is unchanged: `E.up(4)` is *a new* structure with the same component names as `E` and the same component values, except for `xcoord` (which, in this case, is incremented by 4)—so we can chain the "method calls" as in `E.up(4).left(3).dist()`.

Finally, inheritance. The `xby` operator provides the analog of inheritance. If `D` is a struct with default values for persons (e.g. `country: Canada`) then `D xby P` is the struct (object) `P` with `P` inheriting the defaults of `D`. But, as in real OO, `P` can override the defaults.

7.4 NOTES, REFERENCES, AND PROBLEMS

7.4.1 Explain what a call like `self xby field()` actually does. Why is it not surprising that it does not modify the current structure?

CHAPTER 8

DICTIONARIES

A PyFL dictionary is basically the same as the dictionaries found in many other languages, including, of course, Python. It is essentially a set of key-value pairs, with a given key appearing in exactly one pair. Only a word, a string, or a number can be a key, but a value can be an arbitrary data value.

The main difference from other languages is that PyFL dictionaries—like all PyFL data—are immutable. The dictionary operators we discuss below do not modify their arguments; instead, they produce new dictionaries that are (possibly) different from the original argument.

There is one dictionary constant: `dict0`; it corresponds to the empty dictionary, i.e. the empty set of key-value pairs. Every dictionary in PyFL is build by successively adding keys to `dict0`.

Key-value pairs are added by the `update` operation: if `d`, `k`, and `v` denote a dictionary d, a key k, and a value v respectively, `update(d,k,v)` produces a *new* dictionary identical to d except that key k yields value v. This is true whether or not k was already a key of d.

Accessing a dictionary is done with the `atkey` infix operator: `d atkey k` denotes the value associated with k. If k is not a key of d, the result is a runtime error. We can avoid these runtime errors by checking if our dictionary has a given key. We do this with the `has` infix operator: `d has k` returns `"true"` if k is a key of d, otherwise `"false"`.

Adding key-value pairs one at a time can quickly become tiring. Thus PyFL provides another function, `entries`, that automates the process. `entries` is infix: the first argument (to the left) is a dictionary, and the second (to the right) is a list of *entries* to be added; an entry is a struct with two keywords: the `key` keyword, and the `value` keyword. The result of `entries` is a new dictionary, which is like the first argument but extended with the entries of the second argument. Thus, to add make a dictionary associating digits with their spelled-out names, we would do:

```
digits = dict0 entries
  [% <<key:0, value: 'zero' >>,  <<key:1, value: 'one' >>,
     <<key:2, value: 'two' >>,   <<key:3, value: 'three' >>,
     <<key:4, value: 'four' >>,  <<key:5, value: 'five' >>,
     <<key:6, value: 'six' >>,   <<key:7, value: 'seven' >>,
     <<key:8, value: 'eight' >>, <<key:9, value: 'nine' >>
```

```
%]
```

We will also add an operator `keys` and a predicate `isempty`.

In addition we will define two infix operators `map` and `filter`. If `d` denotes a dictionary d and `f` a function f then `d map f` denotes the result of applying f to each of the values of d. And if p denotes a predicate p, `d filter p` denotes the dictionary whose set of *entries* is those entries of d for which p is true.

Finally, we plan to extend VBOs to allow the use of dictionaries as ranges. Thus for example `sum e.value**2 for e in dict d end` produces the sum of the squares of the values of d.

8.1 NOTES, REFERENCES, AND PROBLEMS

8.1.1 Problem: Dictionaries, also known as *hash tables* are one of the most widely used data-structures in every language. Their main advantage is their efficiency: in Python, the implementation of dictionaries allows for →**Add here complexity of dict operations.**

8.1.2 Problem: Dictionaries are used internally in the implementation of PyFL: →**say few things**

8.1.3 Problem: The implementation of dictionaries in PyFL is not as straightforward as one might —on a cursory inspection— be led to think. The problem is that Python's dictionaries are mutable, whereas PyFL, as a functional language, must have immutable data structures. It takes some engineering to get efficient, immutable dictionaries. →**say a few things: how are they implemented?**

8.1.4 Problem: If a key already exists in a dictionary and we `update` it, the resulting dictionarty has that's keys associated value updated. Write code for a new kind of *write-safe* dictionary, which is like a normal dictionary, except it does not update existing keys. Where might we use such a dictionary?

8.1.5 Problem: Write code for a function like `entries`.

CHAPTER 9

PATTERNS

9.1 Case Statements

PyFL does not have pattern matching for function definitions. Every function has a unique single definition in which the formal parameters are distinct individual variables. This greatly simplifies the implementation, among other things.

PyFL does however have a pattern matching case statement which makes pattern matching function definitions possible. A case statement consists of the words `case`, followed by a subject expression, followed by the word `of`, followed by a sequence of pattern/result pairs, followed by the word `end`. A pattern/result pair is a pattern followed by a semicolon, followed by an expression, and terminated by a semicolon. To see how this all works, suppose we want to define a function `ls` which sums the elements of a list `K`. There are two patterns here, one when `K` is empty and one when `K` is non empty, and has a head and a tail. In PyFL we can write

```
ls(K) =
 case K of
        : 0;
   h | t : $ h + ls($ t);
 end;
```

~~In the result expressions variables which are matched~~ → and bound are preceded by `$` signs. →**Bad syntax, meaning unclear. What are result expressions? Should there be a comma after "expressions"?** The evaluation of a case statement proceeds by trying to match the patterns in turns, till a match is successful, in which case the value of the corresponding expression is returned. *If no match is found, the result is a runtime error.*

Now suppose that there are various kinds of elements in the list and we want to sum only the integers. There is a built-in pattern `int` which matches exactly integers, but it doesn't bind any variables. To do this we need the operator `=`. The pattern expression `=v` (wher `v` is a variable) always succeeds, but has the effect of binding `v` to the last item matched. Our definition becomes

```
ls(K) =
```

```
 case K of
                : 0;
  int =i | t : $i + ls($t);
  h | t      : ls($t);
 end;
```

Finally, suppose there are also floats in the list and we want to include them as well in the sum. We can do this using the either/or construct written with decorated angle brackets:

```
ls(K) =
 case K of
                          : 0;
  <%int float%> =n | t : $n + ls($t);
  h | t                 : ls($t);
 end;
```

There are several other pattern operators but the examples make the principle clear.

CHAPTER 10

INPUT & OUTPUT

PyFL handles input/output with pseudo-functions whose evaluation causes as side effects the desired I/O activity. We currently don't see any other way to do this and we make no attempt to claim that our approach is "pure" in any sense.

The simplest pseudo-function is `output`. Evaluating the "call" `output(i)` returns `i` as its value but has the side effect of printing `i` on the output stream. The value of `i` is printed without surrounding blanks, following the conventions of list constants. Thus, in particular, strings appear in quotes, but words appear without double quotes. The pseudo-function `outputs` is like `output` except strings appear without quotes.

The pseudo-function `input` reads one input item, using the list constant convention. The call `input(s)`, where `s` a string, prints `s` as a prompt (without quotes). There are other variations of the pseudo-functions, for example `outputln`, which works like `output` but prints a newline after.

Of special importance is `outputf`, which provides formatted output: `outputf(s,i1,i2)` will print `i1` and `i2` using string `s` as a (Python) format.

As an example, here is a first attempt to write a PyFL program to calculate the approximate integral of a function `f` from `a` to `b`:

```
output(intab)
  where
    m = (a+b)/2;
    intab = (b-a)*(f(a) + 4*f(m) + f(b))/6;
    a = input('a: ');
    b = input('b: ');
  end
```

However, there are problems with this program. First, there is no output prompt; input is requested, and then the output is abruptly given. It seems we need an extra output expression for the prompt, but where does it go?

The second problem is that the values of `a` and `b` are requested in reverse order. The reason is that the demand for `output(intab)` produces a demand for `intab` and for the sum that begins with `f(a)`. The demand for `(b-a)` generates a demand for `b`, which produces a

demand for `input('b:  ')`. Of course we could write `(b-a)` as `(-a+b)` but this is ad hoc. We need a more systematic general solution; we need to generate a demand for `a` before we really need it.

We can do this with the "ignore" operator `&&`. A demand for $\mathscr{A}$ `&&` $\mathscr{B}$ produces a demand for $\mathscr{A}$, but the resulting value is ignored (discarded), and instead $\mathscr{B}$ is evaluated and its value is passed on as the result; $\mathscr{A}$ is executed solely for its side effects.

Returning to our example, evaluating `a && b && output(intab)` will cause `a` and `b` to be read in and cached in that order. Additionally, evaluating

```
outputs("Approximate integral: ") && output(intab)
```

will put a prompt before the output. Thus, the correct program is the same as before, except the subject of the `where` clause is

```
a && b && outputs('Approximate integral is: ')
  && output(intab)
```

Notice that `&&` is associative; the `&&` operator is basically *the semicolon operator of imperative languages* →**(with the caveat that in PyFL there are no re-assignments of variables in the imperative fashion).**

CHAPTER 11

IMPLEMENTATION

11.1 Strategy

The implementation of PyFL proceeded incrementally in stages, each stage concluding with testing.

The first stage was a simple expression evaluator, which involves parsing data constants and implementing the data operations. Fortunately this stage had already been done as part of the PyLucid project. (PyLucid is a sister language of PyFL, also based on POP-2 — the borrowing from the PyLucid project saved writing about 1000 lines of code.)

The implementation of the POP-2 data types in terms of the Python data types is straightforward. For example, POP-2 strings are represented as Python strings beginning with "`$`", and POP-2 words as Python strings beginning with "`.`". The evaluator is just that, there is no attempt to compile expressions. Evaluation proceeds top-down the expression tree in the obvious fashion.

The expression parser is ~~top-down recursive descent~~→**rewrite this; unclear.** based on tables of operator precedence. It is simple and easily extensible.

~~The borrowing from the PyLucid project saved writing about 1000 lines of code.~~→**Incorporated this in earlier ¶**

11.2 Implementing Functions

Once the expression evaluator was working, the next step was to implement `valof` and `lambda` — and allow the programmer to define and apply functions. The crucial concept is that of a *closure*, both an expression closure and a function closure. An expression closure is an ordered pair consisting of an expression and an environment. A closure is basically a suspended computation. To lift the suspension, the expression is evaluated in the environment.

An environment encapsulates the context in which an expression is to be evaluated. In the simplest case it is a block, a set of mutually recursive definitions, like the body of a `valof`. However `valof`'s can be nested, so an environment is a sequence $\langle b_0, b_1, b_2, \ldots \rangle$ of blocks with b_0 the innermost block, b_1 the enclosing block, b_2 the block enclosing b_1, and so on.

To evaluate a variable v in an environment, we look up the relevant definition of v and evaluate the defining expression in the relevant environment. To find the definition of v we look first in b_0. If we find a definition, we evaluate its right hand side in the environment $\langle b_0, b_1, b_2, \ldots\rangle$ Then we look in b_1. If there is such a definition, we evaluate the defining expression in the environment $\langle b_1, b_2, b_3, \ldots\rangle$ If there is no definition we look in b_2, then b_3, then b_4, and so on. In general, if b_i is the first block for which we find a definition for $v\,v$, we evaluate the defining expression in environment $\langle b_i, b_{i+1}, b_{i+2}, \ldots\rangle$ If we reach the end of the b's without finding a definition, we have an undefined variable runtime error.

Evaluating a `valof` is now straightforward. Let $\mathcal{D}$ be its body (a set of mutually recursive definitions, i.e. a block). The value of the `valof` in environment $\langle b_0, b_1, b_2, \ldots\rangle$ is the value of `result` in environment $\langle \mathcal{D}, b_0, b_1, b_2, \ldots\rangle$

Now for functions. A `lambda` expression in a given environment $e = \langle e_0, e_1, \ldots\rangle$ is evaluated to a function closure. A closure consists of a list of formal parameters (denoted p_0, p_1, ...), a body b (an expression), and an environment. In the value of a lambda expression the parameters and body are supplied by the expression, and the environment —called the defining environment— is e.

The crucial procedure is evaluating an expression consisting of a function closure $((p_0, p_1, p_2, \ldots), b, e)$ applied to actual parameters a_0, a_1, a_2, ... in the *calling environment* $e' = \langle e'_0, e'_1, \ldots\rangle$.

To do this we form a block c in which each p_i is defined as the closure (a_i, e'). Then the value of the function call is the value of body b in the environment $\langle c, e_0, e_1, e_2, \ldots\rangle$. This corresponds to static binding and call-by-name.

If we want dynamic binding we replace e by e', and if we want call-by-value we equate each p_i to the value of a_i in e'. PyFL uses static binding and call-by-name because only this combination is faithful to the syntactic rules of the lambda calculus. Only this combination is consistent with treating PyFL functions as mathematical functions that map arguments to results.

11.3 The `where` and `let` constructs

Once we have implemented `valof` it is straightforward to implement `where` and `let` — by translation. For example, here is how a `where clause` is transformed to a `valof` one:

```
S                                    valof
  where                                result = S;
    a = A;                    ⟹         a = A;
    b = B;                               b = B;
    c = C;                               c = C;
  end                                  end
```

Of course this assumed that in the `where` clause, `result` is neither defined nor used. A similar translation handles `let` clauses.

11.4 Implementation of Variable Binding Operators

Variable Binding Operators are handled by translating into a `map`/`reduce` expression.

```
reduce(l,s,b) = if l == []  then b
                            else s(hd(l),reduce(tl(l),s,b))
```

```
                    fi;
    map(l,f) = if l == [] then []
                          else f(hd(l)) :: map(tl(l),f)
               fi;
    sumof(x,y) = x+y;
    b = 0;
```

The function `reduce` runs down the list `l` accumulating the values of `s`, starting out with the base value `b`. Given the VBO

```
sum a[i]**2 for i in range(0,10) end
```

we translate it into

```
reduce(map(range(0,10), lambda (i) a[i]**2 end), sumof, 0)
```

11.5 `while` and `until`

The `while` and `until` clauses are also handled by translation—to tail recursion. The `while` clause

```
    while p(A,B,C)
     A = a0 fby an;
     B = b0 fby bn;
     C = c0 fby cn;
     result = f(A,B,C);
    end
```

for example, becomes

```
    h(a0,b0,c0)
     where
      h(a,b,c) =
      if p(a,b,c) then f(a,b,c)
                  else h(an,bn,cn)
      fi
     end,
```

and a slightly more elaborate translation handles `until`.

11.6 Case statements

Unfortunately there was no quick way to implement case statements. We had to produce a parser for the mini language of patterns. Then, an interpreter for parsed patterns. The interpreter has the form of a procedure $\mathcal{I}$ which takes a parsed pattern $\mathcal{P}$ and a list $\mathcal{K}$. $\mathcal{I}(\mathcal{P}, \mathcal{K})$ returns false (no match), or returns true (match found), plus an assignment a of bindings to "dollar variables". Finally, we altered the expression evaluator to take the values of dollar variables into account.

11.7 Implementation of Input/Output

Input/Output is straightforward to implement because it is based on side effects and side effects are easily produced in an imperative implementation language.

There is one complication, and that is that we must not ask twice for input with the same prompt. Instead, once an item has been read in, in response to the prompt p, it is stored in an associative cache with key p. Then if in the future the expression `input(`p`)` is evaluated, it returns the cached value.

PyFL currently does not clear the input cache because there is no a priori way to know when an input will no longer be used. However some simple program analysis could handle most cases.

11.8 Error Handling

One of the disadvantages of dynamic typing is the need to deal with runtime errors—for example, being asked to multiply two strings. There is no standard approach.

One possibility is to interrupt the computation and print an error message. The problem with this is that it doesn't give any information about how the error occurred, and doesn't allow the program to recover.

PyFL instead uses *error objects*—special data items that encapsulate error messages but are otherwise first class objects that can be arguments of operations, actual parameters, or results of function calls. So, for example, if we are asked to multiply two strings we can return as the result `error("Product of strings")`.

The only complication is that operations have to be prepared to accept error objects as arguments, and not just return them as results. The usual solution is to simply pass on the error objects, but it is also possible to examine them and pass on more informative errors. In any event, the error objects tend to bubble up to the top so that the "result" of a computation is an indication of what went wrong. The computation is not interrupted.

The downside of this policy is that the implementation code is full of tests to tell if data items are errors. Essentially every time we produce a new value we have to check that it's not an error object, and if it is pass it on or pass on a derived error. This enables good error reporting, though it undoubtedly slows down the interpreter somewhat.

11.9 The Advantages if Python

Up to now we have had very little to say about Python, in spite of PyFL's name. That is because there is no distinguishing feature of Python that was decisive for the implementation. We could, with a comparable effort, implement PyFL in Rust or Go (RuFL? GoFL?).

Yet there are several (not necessarily exclusive) features of Python that made the work much easier, compared to doing it in a rudimentary language like C. The first, of course, was the existence of the Python implementation of PyLucid. This saved writing hundreds of lines of code plus making important design decisions, like having an IDE or an abstract algebra of expressions.

Another feature of Python of decisive importance was automatic storage management. An earlier similar project using C foundered because of the difficulty of ensuring no memory leaks in thousands of lines of code.

Another surprisingly important feature was the ability of a Python function to return multiple values. This came into play when designing the abstract algebra of expressions. For

each category of expressions (say, VBOs) we have a constructor, which produces a VBO given its parts, and a destructor which, given a VBO, yields its parts. The destructor obviously needs to return multiple values.

One big advantage is the existence of a huge Python community. Python is arguably the number one most popular programming language. This means there are millions of programmers experienced in the language and capable —in theory— of contributing to the PyFL project.

A related advantage is the huge collection of external modules available. These were not required for the interpreter itself but they are invaluable for enabling different applications of PyFL. For example, we produced a version of the interpreter which imported matplotlib, the popular and powerful graphics package. Then we added a dozen or so lines of code to implement simple interface pseudo functions for rectangular and polar graphs. This means a programmer can generate data sets in PyFL and see them plotted. There are modules for every imaginable application—numerical analysis, natural language processing, speech recognition, and so on ad infinitum. The modules for game writing are especially promising.

11.10 Performace

Readers by now are undoubtably aware of a big elephant in the room—performance. The PyFL implementation is already an interpreter and Python itself is itself interpreted. Two levels of interpretation mean it is much slower than, say, Haskell. Furthermore we made very little effort to optimize computations.

Nevertheless modern computers are so powerful that the PyFL interpreter is usable for small and medium programs. This is a recent development because even five years ago the project would have been impractical.

~~We make no apology for not producing a fast implementation. We are following~~→**At the early stage of the language's development, speed was none of our aims; we followed** Knuth's famous dictum, that "premature optimization is the root of all evil". We believe it is crucial to get the language right, and only then to figure out how to implement it efficiently.

Also it is not as if no effort was made to speed up the computations. In particular two relatively simple optimizations produced significant results.

The most important change involved expression closures. Recall that an expression closure is a pair consisting of an expression and an environment, and that it stands in for the value of the expression in the environment. A closure represents a temporarily suspended version of this computation, and if the value of the closure is demanded, the computation is launched.

A simple optimization such as this ensures that this computation is not duplicated if the value is needed again. When the value of the computation is returned, this value is substituted for the closure in place. This converts every reference to the closure to a reference to its value. This optimization is crucial, not the least because every function call generates expression closures for all the actual parameters.

Another improvement that worked surprisingly well involved the implementation of environments. Recall that an environment is a sequence of blocks, each block a set of mutually recursive definitions. ~~Originally an environment was implemented as a linked list of blocks, each block itself being a linked list of key-value pairs.~~→**Internally, an environment is a linked list of blocks; originally each block itself was a linked list of key-value pairs. (The variable names were the keys, with values the corresponding defining expressions.)** The optimization was to represent blocks as Python dictionaries, with variable names as keys and

expressions as values, as above. This alone yielded a speedup of about 30%.

There were other attempts at optimization that did not pan out. We tried replacing call-by-name by call-by-value (evaluating actual parameters before evaluating the function body). There are obviously examples where this fails, but otherwise one would expect that avoiding evaluating closures would speed things up. This was not the case.

Another attempted optimization involved the list primitives Head, Tail, and Cons. These are called literally millions of times in medium size programs and one would expect that streamlining them would be of benefit. We produced streamlined versions (for example omitting the test in Head that the argument is not empty) but it made very little difference.

Presumably some sort of compilation scheme should speed up evaluation annd we tried a few, without success.

However we eventually devised a significant optimization. Recall that the interpreter repeatedly takes apart expressions and evaluates the parts. Originally expressions were implemented as linked lists and all this classifying and dissemballing generated the millions of Head/Tail calls mentioned above.

We changed the representation to use amall dictionaries, one for each node in the parse tree. Then instead of traversing a linked list to locate a component, a single dictionary lookup suffices. Python dictionaries are very efficient and the revised interpreter was 2 to 3 times faster than the old list-based one.

11.11 NOTES, REFERENCES, AND PROBLEMS

11.11.1 Why not add some remarks?

11.11.2 Or some references about implementing programming languages?

11.11.3 **Problem:** Or even problems?

CHAPTER 12 CONCLUSIONS

The implicit research question of the PyFL project is, “can we design a functional language which is accessible to programmers familiar with conventional imperative languages and mathematical notation?”.

We believe we have answered that affirmatively, in that PyFL is just such a language. Our evidence is the description just given and the many programs already written, (some of which appear above).

A secondary question is, “can it be practically implemented”. We can’t claim to have answered that definitely because although we have an implementation, it is very slow. However that’s not to say it is of no value. Even though the efforts to optimize it have been minimal, on modern hardware it is fast enough to be usable. In the end, however, these questions will only be answered with certainty if a reasonable implementation is produced and the language becomes popular. This is more a social issue rather than a technical one, but hopefully this paper will contribute to a favourable outcome.

12.1 Future Work

There are a number of direction in which future work might proceed.

One possibility is to identify more features —more built-ins like while— but it is not clear what they would be. There are no obvious gaps.

The IDE notion of module is rudimentary and could do more to ensure safer code by having some enforced notion of namespace.

It is tempting to add some sort of static type checking, perhaps optional type hints. This would help both programmers and implementors.

But finally, the elephant in the room needs our attention. The priority for future work has to be improving performance. In our opinion the best hope for improving performance is some kind of compilation. Right now, if the program contains an expression $\mathcal{A}$+$\mathcal{B}$, it will most likely be evaluated many times. Each evaluation involves predicates that determine that it is a strict operation applied to arguments. One operation extracts the operator + and another the list [$\mathcal{A}$,$\mathcal{B}$] of operands. The evaluator proceeds down the list of arguments evaluating each in turn, producing a list of operands. A dictionary lookup produces a function that takes the list of operands and returns their sum.

To avoid all this, we could precompile $\mathcal{A}$+$\mathcal{B}$ into a list `[eval` $\mathcal{A}$ `eval` $\mathcal{B}$ `add]` of commands that when executed return the sum.

It is plausible that compilation would speed things up but the only way to confirm this is to try it.

APPENDIX

A BUILT-INS LIST

A.1 Numbers

A.1.1: Constants

`pi`	3.1415926
`phi`	1.6
`eee`	2.71828
`none`	$none$

A.1.2: Prefix Operators

`isnum (x)`	is x a number
`isint (x)`	is x an integer
`-u`	$-u$
`sin (u)`	$\sin(n)$
`cos (u)`	$\cos(n)$
`tan (u)`	$\tan(n)$
`exp (u)`	$\exp(u)$
`log (u)`	$\log(u)$
`log10 (u)`	$\log_1 0(u)$
`abs (u)`	absolute value of u
`sqrt (u)`	root of u
`floor (u)`	floor of u
`id (x)`	x

A.1.3: Infix operators

`u+v`	$u + v$
`u-v`	$u - v$
`u*v`	$u * v$
`u/v`	u/v
`u**v`	$u \sup v$

`u div v`	$u//v$
`u mod v`	$u \bmod v$
`u < v`	$u < v$
`u > v`	$u > v$
`u >= v`	$u \geq v$
`u <= v`	$u \leq v$
`u == v`	$u = v$
`u != v`	$u \neq v$
`u max v`	$\max(u, v)$
`u min v`	$\min(u, v)$

A.2 Lists

A.2.1: Constants

`[]`	the empty list

A.2.2: Prefix Operators

`islist l`	is l a list
`hd l`	the head of list l
`tl l`	the tail of list l
`el2 l`	the second element of l
`el3 l`	the third element of l
`el4 l`	the fourth element of l
`length l`	the length of l
`elements l`	set of elements of list l

A.2.3: Infix Operators

`x :: l`	the list with head x and tail l
`l <> m`	m appended to l
`l elt n`	the n-th element of l
`l ? p`	l whenever p
`n .. m`	list of numbers from n upto (not including) m
`x isin l`	does x occur in list l

A.2.4: Functions

`[%x,y,z,...%]`	list beginning x, y, z

A.3 Sets

A.3.1: Constants

`{}`	the empty set

A.3.2: Prefix Operators

`isset x`	is x a set
`hd s`	an arbitrary element of set s

`tl s`	the remaining members
`size s`	the number of elements of set *s*
`enumeration s`	a list enumerating the elements of *s*

A.3.3: Infix Operators

`x isin s`	is *x* an element of set *s*
`s+t`	union of sets *s* and *t*
`s++t`	union of sets *s* and *t* known to be disjoint
`s-t`	difference of sets *s* and *t*
`s ^ t`	the intersection of sets *s* and *t*
`s * t`	the cross product of sets *s* and *t*
`s <= t`	is set *s* a subset of set *t*
`s * f`	the result of applying function *f* to every element of set *s*
`s / p`	those elements of set *s* for which predicate *p* is true
`x addto s`	the result of adding *x* to set *s*
`enumeration s`	a list enumerating the elements of set *s*

A.3.4: Functions

`{%x,y,z,...%}`	the set containing *x*, *y*, *z*, and others

A.4 Strings

A.4.1: Constants

`''`	the empty string

A.4.2: Prefix Operators

`string x`	is *x* a string
`charlist a`	the list of characters in string *a*
`hd a`	the first character of string *a*
`tl a`	the remaining characters of string *a*
`length a`	length of string *a*

A.4.3: Infix Operators

`a+b`	concatenation of strings *a* and *b*

A.5 Booleans

A.5.1: Constants

`"true`	true
`"false"`	false

A.5.2: Prefix Operators

`isbool x`	is x boolean
`not p`	negation of boolean p

A.5.3: Infix Operators

`p and q`	conjunction of booleans p and q
`p or q`	disjunction of booleans p and q

A.6 Structs

A.6.1: Infix Operators

`u . c`	component c of struct u
`u.c(x,y,x,...)`	function component c applied to actuals x, y, z, ...
`u xby v`	struct u extended by struct v

A.6.2: Functions

`<< c:C d:D e:E ... >>`	struct with components C, D, E, ..., labelled c, d, e, ...

A.7 Streams

A.7.1: Prefix Operators

`first I`	first component of stream I
`rest I`	stream I with first component removed

A.7.2: Infix operators

`x join I`	result of prepending x o stream I
`I seg n`	list of first n components of stream I

Note: `first`, `rest` and `join` can be abbreviated as `fs`, `rs`, and `jn`

A.8 IO

A.8.1: Input

`input(a)`	print prompt string a and input an object
`inputall(a)`	print prompt string a and return the list of all objects presented in response
`inputfile(f)`	read and return one object from file with name f

A.8.2: Output

`output(x)`	output object x
`output(s)`	outputnstring s without quotes
`outputln(x)`	output x followed by a newline
`outputp(p,x)`	output prompt p then x
`outputpln(p,x)`	output p then x then a newline
`outputf(f,x)`	output format f with parameter x
`outputf(f,x0,x1)`	output format f with parameters x_0 and x_1,
`outputfile(fname,x)`	output x to file with name $fname$
`outputi(p,x)`	output prompt p the x then wait for input; if input is q then produce remaining output without pausing for `outputi`; otherwise continue pausing for next call to `outputi`