Notes on Programming Language Concepts

1 Abstract

We learn programming through one or more languages, and the programs we write then become natural subjects of study to understand languages at large. This note provides an introduction to programming languages: a study, from one level up, of the media by which we structure data and programs. There are many ways to organize the study of programming and programming languages. The central theme here is the concept of program reasoning. Programs are typically static entities. But when we run a program, it produces a complex, dynamic behavior that yields services, and (sometimes) frustration. Everyone who writes programs ultimately care whether they realize it or not in having evidences of program correctness. Sometimes we even write programs to help us with this task. Program reasoning is a metric for programming language design.

We take an operational approach to programming language concepts; studying those concepts in interpreters, compilers and run-time structures for simple languages, and pointing out their relations with real-worls programming languages. We will use the OCAML programming language as presentation language through out to illustrate programming language concepts by the implementation of interpreters for the languages we consider. OCAML is ideal form implementing interpreters because it provides algebraic datatypes, pattern-matching and it is strongly typed. This leads to brevity and clarity of examples that cannot be matched by languages without these features.

2 Programming Languages and Programming Paradigms

Software is written in programming languages, and programming languages adopts programming paradigms. Some paradigms are concerned primarily with implications for the execution model of the language, such as allowing *side effects*, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned primarily with the way that code is organized, such as grouping code into units along with the state that is modified by the code. Common programming paradigms include imperative which allows side effects, functional which does not allow side effects, declarative which does not state the order in which operations execute, object-oriented which groups code together with the state the code modifies, procedural which groups code into functions. For example, languages that fall into the imperative paradigm have two main features: they state the order in which operations take place, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in object-oriented programming, code is organized into objects that contain state that is only modified by the code that is part of the object. Most object oriented languages are also imperative languages. In contrast, languages that fit the declarative paradigm do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute. The implementation of the language's execution model tracks which operations are free to execute and chooses the order on its own.

Here we will consider two programming paradigms:

- imperative programming: This is the style most people learn first: variables, assignments, loops. A problem is solved by dividing its data into small bits (variables) that are updated by assignments. Since the data is broken into small bits, updating goes in stages, repeating assignments over and over, using loops. This programming paradigm comes from 1950's computer hardware — it is all about reading and resetting hardware registers. It is no accident that the most popular imperative language, C, is a language for systems software. Today, object languages like Java and C# rely on variables and assignments, but the languages try to "divide up" computer memory so that variables are "owned" by objects, and each object is a kind of "memory region" with its own "assignments", called methods. This is a half-step in the direction of making us forget about 1950's computers.

- functional programming: The imperative paradigm requires memory that is updated by assignments; it is single-machine programming. This approach is impossible for a distributed system, a network,

the internet, the web. A node might own private memory, but sharing and assignment to it cause race conditions. Indeed, there is not even a global clock in such systems! Functional (declarative) programming dispenses with memory and assignments. All data is message-like or parameter-like, copied and passed from one component to the next. Since there are no assignments, command sequencing ("do this line first, do this line second, etc.") becomes unimportant and can even be discarded. The result is a kind of *programming algebra*, where you wrote a set of simultaneous equations to define an answer to a problem — the equations were definitions and their ordering did not matter. Since there is no memory, data values (parameters) must be primitives (ints, strings) and also data structures (sequences, tables, trees). Components pass these complex parameter data structures to each other. There are no race conditions because there are no global variables — all information is a parameter or a returned answer. This paradigm applies both to a program ("function") that lives on one machine and also to a distributed system of programs — parameters replace memory.

Although C looks radically different from ML or Smalltalk, all programming languages are languages, and languages have standard foundations. There is a traditional list of these foundations, which states that every language has a:

- syntax: how the words, phrases, and sentences (commands) are spelled.
- semantics: what the syntax means in terms of nouns, verbs, adjectives, and adverbs.
- pragmatics: what application areas are handled by the language and what is the (virtual) machine that executes the language.

To understand syntax, we need a suitable notation for stating syntactic structure: grammar notation. To understand semantics, we need to learn about semantic domains of expressible, denotable, and storable values. To understand pragmatics, we need to learn the standard virtual machines for computing programs in a language. The virtual machines might use variable cells, or objects, or algebra equations, or even logic laws. In any case, the machines compute execution traces of programs and show how the language is useful.

We will understand the nature of languages by writing programs about them. These programs will implement many interesting features of languages from different perspectives, embodied in different actions:

- An interpreter will consume programs in a language and produce the answers they are expected to produce.
- A type checker will consume programs in a language and produce either true or false, depending on whether the program has consistent type annotations.
- A pretty-printer will consume programs in a language and print them, prettified in some way.
- A verifier will consume programs in a language and check whether they satisfy some stated property.
- A transformer will consume programs in a language and produce related but different programs in the same language.
- A compiler, will consume programs in a language and produce related programs in a different language (which in turn can be interpreted, type-checked, pretty-printed, verified, transformed, even compiled...).

Observe that in each of these cases, we have to begin by consuming (the representation of) a program. We will briefly discuss how we do this quickly and easily, so that in the rest of our study we can focus on the remainder of each of these actions.

2.1 Interpreters and Compilers

An interpreter executes a program on some input, producing an output or result. An interpreter is usually itself a program, but one might also say that a processor is an interpreter implemented in silicon. For an interpreter program we must distinguish the interpreted language **L** (the language of the program being executed) from the implementation language **I** (the language in which the interpreter is written). A compiler takes as input a source program and generates as output another program, called target program, which can be executed. We must distinguish three languages: the source language **S** of the input program, the target language **T** of the output program and the implementation language **I** of the compiler itself. The compiler does not execute the program: after the target program has been generated it must be executed by an interpreter which can execute programs written in the language **T**.

In general, interpretation leads to greater flexibility and better diagnostics (error messages) than does compilation. Because the source code is being executed directly, the interpreter can include an excellent source-level debugger. It can also cope with languages in which fundamental characteristics of the program, such as the sizes and types of variables, or even which names refer to which variables, can depend on the input data. Some language features are almost impossible to implement without interpretation: in Lisp and Prolog, for example, a program can write new pieces of itself and execute them on the fly. (Several scripting languages, including Perl, Tcl, Python, and Ruby, also provide this capability.) Compilation, by contrast, generally leads to better performance. In general, a decision made at compile time is a decision that does not need to be made at runtime. For example, if the compiler can guarantee that variable x will always lie at location 49378, it can generate machine language instructions that access this location whenever the source program refers to x. By contrast, an interpreter may need to look x up in a table every time it is accessed, in order to find its location. Since the (final version of a) program is compiled only once, but generally executed many times, the savings can be substantial, particularly if the interpreter is doing unnecessary work in every iteration of a loop.

2.2 Everything (We Will Say) About Parsing

Both interpreters and compilers analyze the source code to check the code conforms to the rules of the grammar defining the language. This activity is called parsing. A parser is a software component that takes the source code and builds a data structure often some kind of parse tree, abstract syntax tree or other hierarchical structure giving a structural representation of the source program, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. Parsing is a very general activity whose difficulty depends both on how complex or ambiguous the input might be, and how much structure we expect of the parsers output. We would like the parser to be maximally helpful by providing later stages as much structure as possible.

A key problem of parsing is the management of ambiguity: when a given expression can be parsed in multiple different ways. For instance, the input

$$23 + 5 * 6$$

could parse in two different ways: either the multiplication should be done first followed by addition, or vice versa. Though simple disambiguation rules (that you probably remember from elementary school) disambiguate arithmetic, the problem is much harder for full-fledged programming languages.

Ultimately, we would like to get rid of ambiguity once-and-for-all at the very beginning of processing the program, rather than deal with it repeatedly in each of the ways we might want to process it. Thus, if we follow the standard rules of arithmetic, we would want the above program to turn into a tree that has a (representation of) addition at its root, a (representation of) 23 as its left child, multiplication as its right child, and so on. This is called an *abstract syntax tree* (AST): it is abstract because it represents the intent of the program rather than its literal syntactic structure (spaces, indentation, etc.); it is syntax because it represents the program that was given; and it is usually a ree but not always.

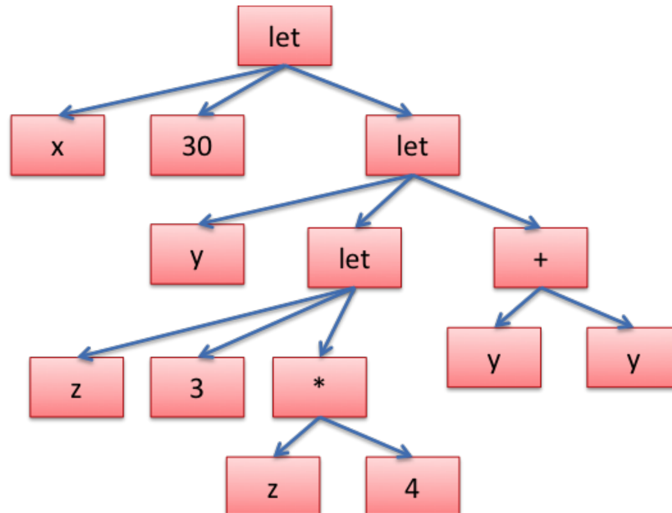

Figure 1: An Abstract Syntax Tree

More in detail, an abstract syntax tree, or just syntax tree, is a tree representation of the abstract syntactic structure of source code. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

The abstract syntax tree of the simple OCAML program below is displayed in Figure 1.

```
let x = 30 in
  let y = (let z = 3 in z*4)
    in y+y;;
```

Back to the expression $23 + 5 * 6$, as we have said, we could push the problem of disambiguation onto a parser. This is what most real programming languages do. Because parsing is not our concern, we are instead going to ask the programs author to use an unambiguous syntax. For instance, the above expression might be written avoiding the ambiguity induced by not properly parenthesizing the program by the XML structure of Figure 2 or by the JSON representation of Figure 3.

These are both worthy notations. Instead, we will use a related, and arguably even simpler, OCAML structure. Here, we will use OCAML algebraic data types to represent both concrete and abstract syntax. Since we will be writing programs to process other programs, it is especially helpful to keep apart the program being processed and that doing the processing. For us, the program being processed will be written as a suitable OCAML algebraic type, while the program doing the processing will be an OCAML program.

In the case of simple arithmetic expressions we must first introduce an algebraic datatypes for managing expressions:

```
# type exp = CstInt of int
                  | Sum of exp * exp
                  | Times of exp * exp;;
  type exp = CstInt of int | Sum of exp * exp | Times of exp * exp
```

In our case the expression $23 + 5 * 6$ will be represented as

```
# let e = Sum(CstInt 23, Times(CstInt 5, CstInt 6));;
val e : exp = Sum (CstInt 23, Times (CstInt 5, CstInt 6))
```

```
<plus>
  <args>
    <arg position="1">
      <number value="23"/>
    </arg>
    <arg position="2">
      <mult>
        <args>
          <arg position="1">
            <number value="5"/>
          </arg>
          <arg position="2">
            <number value="6"/>
          </arg>
        </args>
      </mult>
    </arg>
  <args>
```

Figure 2: XML code for expression 23 + 5 * 6

```
{plus:
  [{number: "23"},
   {mult:
     [{number: "5"},
      {number: "6"}]}
   ]}
```

Figure 3: JSON code for expression 23 + 5 * 6

To complete the job we need to write the interpreter. The interpreter processes a computer program, it first builds the program abstract syntx tree. Then, it computes the semantics of the tree with a recursively defined tree-traversal function, like the ones you learned in algorithmic courses.

Here we well write interpreter in OCaml. In the case of arithmetic expressions, the interpreter is a function `eval: exp -> int` that uses pattern matching to traverse the abstract syntax tree of the expression to evaluate. In other words, OCaml pattern matching is exploited to distinguish the various syntactic forms of expressions (the nodes of the abstract syntax trees). Note that to evaluate `e1+e2`, the interpreter first recognises the operation, then it evaluates `e1`, and `e2` to obtain two integer values and finally adds those integers.

```
# let rec eval (e : exp) : int =
         match e with
         | CstInt i -> i
         | Sum(e1, e2) -> let v1 = eval e1 in
                          let v2 = eval e2 in v1+v2
         | Times(e1,e2) -> let v1 = eval e1 in
                          let v2 = eval e2 in v1*v2;;
val eval : exp -> int = <fun>
```

Finally, we apply the interpreter to our (running) expression

```
# eval e;;
- : int = 53
```

yielding as result the integer value 53.

We have just completed our first representation of a program execution. From now on we can focus entirely on programs represented as OCAML algebraic data types (aka recursive trees), ignoring the varieties of surface syntax.

2.3 A First Taste of Semantics

What is the meaning of addition and multiplication in our language of expressions? We will address this problem as follows. Which of these is the same?

- `1 + 2`
- `"1" + "2"`

The point is that there are many kinds of addition in computer science:

1. First of all, there are many different kinds of numbers: fixed-width (e.g., 32-bit) integers, signed fixed-width (e.g., 31-bits plus a sign-bit) integers, arbitrary precision integers; in some languages, rationals; various formats of fixed- and floating-point numbers; in some languages, complex numbers; and so on. After the numbers have been chosen, addition may support only some combinations of them.
2. In addition, some languages permit the addition of datatypes such as matrices.
3. Furthermore, many languages support addition of strings. In some languages this always means concatenation; in some others, it can result in numeric results (or numbers stored in strings).

These are all different meanings for addition . Semantics is the mapping of syntax (e.g., +) to meaning (e.g., some or all of the above). Returning to our interpreter, what semantics do we have? We have adopted the semantics OCAML provides, because we map the Sum operator to OCAML "+".

2.4 Representing Expressions by Objects

Here, we use the OCAML algebraic datatype `exp` to represent the abstract syntax tree of expressions. We use the function eval to define the interpreter of the language of expressions, exploiting pattern matching to distinguish among the different forms of expressions. Now we briefly consider an alternative, object-oriented modeling (say in Java) of expression syntax and expression evaluation. This model requires an abstract base class `Exp` (instead of the `exp` datatype) and a concrete subclass for each form of expressions (instead of the datatype constructors).

```
abstract class Exp { }
class CstInt extends Exp {
   protected final int I;
   public CstInt(int i){this.I = i;}
}
class Sum extend Exp {
   protected final Exp e1;
   protected final Exp e2;
   public Sum(Exp e1, Exp e2) {
     this.e1 = e1; this.e2 = e2;}
}
class Times extend Exp {
   protected final Exp e1;
   protected final Exp e2;
   public Times(Exp e1, Exp e2) {
     this.e1 = e1; this.e2 = e2;}
}
/**
  * The Exp object e constructed as
  */

Exp e = new Sum(new CstInt(4), new CstInt(5))

/** will represent the expression 4+5 **/
```

How can we define an interpreter for expression similar to the OCAML eval defined above? The OCAML function uses pattern matching which is not available in Java. The standard object-oriented solution is to declare an abstract method eval in the class `Exp`, and to override the eval method in each subclass. This solution relies on the Java dynamic dispatching mechanism to invoke the correct method. This solution can be found below.

```
abstract class Exp {
   abstract public int eval(); }
class CstInt extends Exp {
   protected final int I;
   public CstInt(int i){this.I = i;}
   public int eval(){return I;}
}
class Sum extend Exp {
   protected final Exp e1;
   protected final Exp e2;
   public Sum(Exp e1, Exp e2) {
     this.e1 = e1; this.e2 = e2;}
   public int eval(){
        return e1.eval() + e2.eval();}
}
```

2.5 Growing the Language Without Enlarging It

We have picked a very restricted programming language, so there are many ways we can grow it. Some, such as representing data structures and functions, will clearly force us to add new features to the interpreter itself. Others, such as adding more of arithmetic itself, can possibly be done without disturbing the core language and hence its interpreter.

First, we will add subtraction. Because our language already has integer, addition, and multiplication, it is easy to define subtraction as `a-b=a+(-1)*b`. But now we should turn this into concrete code. To do so, we face a decision: where does this new subtraction operator reside? It seems natural to just add one more case to our existing datatype for expressions. What are the negative consequences of modifying the `exp` algebraic datatype? This creates a few problems:

1. The first, obvious, one is that we now have to modify all programs that process `exp`. So far is only our interpreter, which is pretty simple, but in a more complex implementation, there could be many programs built around the datatype a type-checker, compiler, etc. which must all be changed, creating a heavy burden.

2. Second, we were trying to add new constructs that we can define in terms of existing ones; it feels slightly self-defeating to do this in a way that is not modular.

3. Third, and most subtly, there is something conceptually unnecessary about modifying `exp`. That is because `exp` represents a perfectly good core language. Atop this, we might want to include any number of additional operations that make the user's life more convenient, but there is no need to put these in the core. Rather, it is clever to record conceptually different ideas in distinct datatypes, rather than put them into one. The separation makes the program much easier for future developers to read and maintain.

Therefore, we will define a new algebraic datatype `exp_ext` (estended expressions) to reflect our intended design methodology. This looks almost exactly like `exp`, other than the added case, which follows the familiar recursive pattern.

```
# type exp_ext =
        | CstIntExt of int
        | SumExt of exp_ext * exp_ext
        | TimesExt of exp_ext * exp_ext
        | Minus of exp_ext * exp_ext;;
  type exp_ext =
  | CstIntExt of int
  | SumExt of exp_ext * exp_ext
  | TimesExt of exp_ext * exp_ext
  | Minus of exp_ext * exp_ext
```

Given this algebraic datatype, we should do two things. First, we should modify our interpreter to deal with extended expressions, and always construct `exp_ext` expressions (rather than any `exp ones`). Second, we should implement a function that translates `exp_ext` terms into `exp` ones. We follow the second alternative.

```
# let rec translate (e : exp_ext) : exp =
         match e with
         | CstIntExt i -> (CstInt i)
         | SumExt(e1, e2) ->
                Sum(translate (e1), translate (e2))
         | TimesExt(e1,e2) ->
                Times(translate (e1), translate (e2))
         | Minus(e1, e2) ->
                Sum(translate (e1),  Times(CstInt(-1), translate (e2)));;
val translate: exp_ext -> exp = <fun>
```

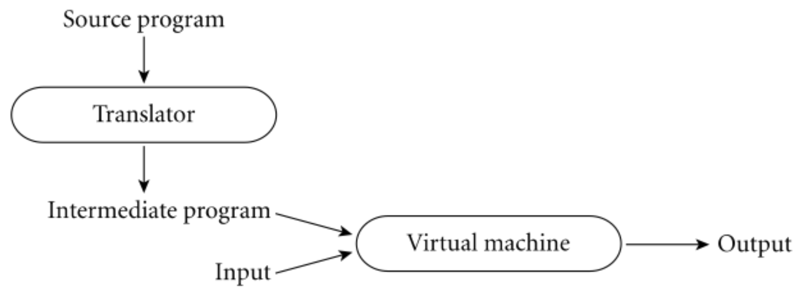

Figure 4: Programming Language Implementation Stages

We now apply our work in a sort of REPL (Read-Eval-Print-Loop) as follows:

```
# let e_ext = Minus(CstIntExt(5),CstIntExt(4));;
val e_ext : exp_ext = Minus (CstIntExt 5, CstIntExt 4)
# translate(e_ext);;
- : exp = Sum (CstInt 5, Times (CstInt (-1), CstInt 4))
```

It is easy to see how the new interpreter will work. We simply need to translate the surface syntax (the `exp_ext` expressions) into `exp` expressions. Then, it suffices to apply the interpreter for `exp`. For example we have:

```
# let e = translate (e_ext);;
val e : exp = Sum (CstInt 5, Times (CstInt (-1), CstInt 4))
# eval e;;
- : int = 1
```

The example considered above describes the real situation for the implementation of a high-level programming language. We have a language that has to be implemented (the language of extended expressions `exp_ext`) and we have a language that has been already implemented (the language `exp`). To implement `exp_ext` we have designed a compiler (the function translate) that translates extended expressions in expressions and then the resulting expression is executed by the `exp` interpreter (the function `eval`) often referred to as the abstract or virtual machine. Most programming language implementations include a mixture of compilation and interpretation as we presented above. They typically look like in Figure 4.

Exercise Let us consider another extension, which is a little more interesting: unary subtraction. Modify the definitions above to deal with unary substraction. Hint. There are many ways we can define unary subtraction. We can define it naturally as $-b = 0 - b$, or with the expansion $-b = 0 + (-1) * b$.

2.6 Conditionals

Now that we have the first kernel of a programming language, let us grow it out a little. The heart of a programming language consists of having a certain control on the order in which instructions are executed instructions and on the representations of information (data) consumed and produced by programs. We will therefore add both control and data to our language, through a simple and important concept: the conditional expressions. Though this seems (and is) a very simple concept, it will force us to tackle several design issues in both the language and the interpreter, and is thus surprisingly illuminating.

Even the simplest conditional exposes us to many variations in language design. Consider one of the form `(if test-exp then-part else-part)`. The intent is that `test-exp` is evaluated first; if it results in a true value then (only) `then-part` is evaluated, else (only) `else-part` is evaluated. (We usually refer to these two

Value	JS	Perl	PHP	Python	Ruby
0	falsy	falsy	falsy	falsy	truthy
””	falsy	falsy	falsy	falsy	truthy
nil, null, None, undef	falsy	falsy	falsy	falsy	falsy
”0”	truthy	falsy	falsy	truthy	truthy
-1	truthy	truthy	truthy	truthy	truthy
[ ]	truthy	truthy	falsy	falsy	truthy
empty object	truthy	falsy	falsy	falsy	truthy

Figure 5: The design space of conditionals

parts as branches, since the programs control must take one or the other.) However, even this simple construct results in at least three different, mostly independent design decisions.

What kind of values can the `test-exp` be? In some languages they must be Boolean values (two values, one representing truth and the other falsehood). In other languages this expression can evaluate to just about any value, with some set colloquially called `truthy` representing truth (i.e., they result in execution of the `then-part`) while the remaining ones are `falsy`, meaning they cause `else-part` to run.

Initially, it may seem attractive to design a language with several `truthy` and `falsy` values: after all, this appears to give the programmer more convenience, permitting non-Boolean-valued functions and expressions to be used in conditionals. However, this can lead to cause inconsistencies across languages. See the table below of Figure 5.

What kind of terms are the branches? Some programming languages make a distinction between statements and expressions; in such languages, designers need to decide which of these are permitted. In some languages, there are even two syntactic forms of conditional to reflect these two choices: e.g., in C, if uses statements (and does not return any value) while the ternary operator ((...?...:...)) permits expressions and returns a value.

If the branches are expressions and hence allowed to evaluate to values, how do the values relate? Many (but not all) languages with static type systems expect the two branches to have the same type (e.g. OCAML). Languages without static type systems usually place no restrictions.

For now, we will assume that the conditional expression can only be a Boolean value; the branches are expressions (because that is all we have in our language anyway); and the two branches can return values of the same type.

To add conditionals to the language, we have to cover a surprising amount of ground. First, we need to define syntax. We will use:

1. the constant value `true`,
2. the constant value `false`,
3. the test expression `(if test-exp then-exp else-exp)`.

to represent the two Boolean constants and the conditional expression.

Our new expression language is given below.

```
type exp =  CstInt of int
        | CstTrue
        | CstFalse
        | Times of exp * exp
        | Sum of exp * exp
        | Sub of exp * exp
        | Eq of exp * exp
        | Iszero of exp
        | Or of exp * exp
        | And of exp * exp
        | Not of exp
        | Ifthenelse of exp * exp * exp
```

Since our terms are no longer purely arithmetic in nature, then we can no longer expect our interpreter to produce only numeric answers! For instance, what should be the result of evaluating the the simple program consisting of `CstTrue`, that is, the program corresponding to the source text true?

Let Booleans be their own type. In any respectable implementation this will impose little additional cost to program execution, while greatly reducing programmer confusion. The consequence of this decision is that we will need a way to represent all the possible outcomes from the interpreter. The **Expressible Values** (EV) are those values which can be the result of the evaluation of a complex expression.

We first need to define the type of values, `evT`, namely the values that can be obtained as result of the evaluation of an expression. Here is the corresponding datatype.

```
type evT = Int of int | Bool of bool
```

We introduced two constructors to fully give all the information about the expressible values. Intuitively, the constructors are the abstract representation of the runtime descriptors specifying the type information of data entities (in our simple case, integers and Booleans). The interpreter is responsible for checking at run-time that every operation is applied only to operands of the correct type. The next step consists of determining the method of evaluation of operations. We have already pointed out that the algebraic datatype `exp` corresponds to the abstract syntax tree in which:

- Every non-leaf node is labelled with an operator.
- Every subtree that has as root a child of a node N constitutes an operand for the operator associated with N.
- Every leaf node is labelled with a constant, variable or other elementary operand.

This representation allows us to avoid considering all the issues concerning precedence and associativity of operators. The parser has already done this job for us. However, the abstract syntax tree representation nothing tells us about the evaluation order of sub-expressions. When the application of operator to operands is considered, two evaluation strategies can be followed. The first, called eager evaluation, consists of first evaluating all the operands and then applying the operator to the values thus obtained. The strategy probably seems the most reasonable when reasoning in terms of standard arithmetic operators. The expressions that we use in programming languages, however, pose problems over and above those posed by arithmetic expressions, because some can be defined even when some of the operands are missing. Let us consider the example of a conditional expression of the form `if a == 0 then b else b/a`.

This expression results from the application of a single operator (the `ifthenelse`) to three operands (the boolean expression, `a==0`, and the two arithmetic expressions `b` and `b/a`). Clearly here we cannot exploit eager evaluation for such conditional expression because the expression `b/a` would have to be evaluated even when `a` is equal to zero and this would produce an error. In such a case, it is therefore better to use a lazy evaluation strategy which mainly consists of not evaluating operands before the application of the operator, but in passing the un-evaluated operands to the operator, which, when it is evaluated, will decide which operands are required, and will only evaluate the ones it requires. The lazy evaluation strategy, used in some declarative languages, is much more expensive to implement than eager evaluation and for this reason, most languages use eager evaluation (with the significant exception of conditional expressions as we will see below).

The problem detailed above presents itself with particular clarity when evaluating Boolean expressions. For example, consider the following expression (in C syntax) `a == 0 || b/a > 2`. If the value of `a` is zero and both operands of logical disjunction `||` are evaluated at the same time, it is clear that an error will result. To avoid this problem, and to improve the efficiency of the code, C, like other languages uses a form of lazy evaluation, also called short-circuiting evaluation, of boolean expressions. If the first operand of a disjunction has the value true then the second is not evaluated, given that the overall result will certainly have the value true. In such a case, the second expression is short-circuited in the sense that we arrive at the final value before knowing the

value of all of the operands. Analogously, if the first operand of a conjunction has the value false, the second is not evaluated, given that the overall result can have nothing other than the value false.

Note that not all languages use the short-circuited evaluation strategy for boolean expressions. Counting on the presence of a short-circuited evaluation, without being certain that the language uses it, is dangerous. For example, we can write in Pascal

```
p := list;
while (p <> nil ) and (p^.value <> 3) do
p := p^.next;
```

The intention of this code is to traverse a list until we have arrived at the end or until we have encountered the value 3. This is badly written code that can produce a runtime error. Pascal, in fact, does not use short-circuit evaluation. In the case in which we have `p = nil`, the second operand of the conjunction yields an error when it dereferences a null pointer. Similar code, on the other hand, mutatis mutandis, can be written in C without causing problems. In order to avoid ambiguity, some languages (for example C and Ada), explicitly provide different boolean operators for short-circuit evaluation.

We can now illustrate the interpreter of our language of expressions. The interpreter exploits the following type checking function.

```
let typecheck (x, y) = match x with
         | "int" ->
             (match y with
                         | Int(u) -> true
                         | _ -> false)

         | "bool" ->
              (match y with
                          | Bool(u) -> true
                          | _ -> false)

         | _ -> failwith ("not a valid type");;

val typecheck : string * evalT -> bool = <fun>
```

The typecheck function uses pattern matching to distinguish the two syntactic forms of types the evaluated values may assume in our language. Next, for each syntactical operation we have in the language we introduce a semantic function that type checks the evaluated values of the arguments and computes the result.

```
let is_zero x = match (typecheck("int",x), x) with
              | (true, Int(y)) -> Bool(y=0)
              | (_, _) ->    failwith("run-time error");;

let int_eq(x,y) =
match (typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v = w)
  | (_,_,_,_) -> failwith("run-time error ");;

 let int_plus(x, y) =
 match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;

let int_times(x, y) =
 match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v * w)
  | (_,_,_,_) -> failwith("run-time error ");;

let int_sub(x, y) =
 match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v - w)
  | (_,_,_,_) -> failwith("run-time error ");;

let bool_and(x, y) =
match(typecheck("bool",x),typecheck("bool",y), x, y) with
  | (true, true, Bool(v), Bool(w)) -> Bool(v && w)
  | (_,_,_,_) -> failwith("run-time error ");;

let bool_or(x, y) =
match(typecheck("bool",x),typecheck("bool",y), x, y) with
  | (true, true, Bool(v), Bool(w)) -> Bool(v || w)
  | (_,_,_,_) -> failwith("run-time error ");;


let bool_not(x) =
match(typecheck("bool",x), x) with
  | (true, Bool(v)) -> Bool(not(v))
  | (_,_) -> failwith("run-time error ");;
```

The semantic function `typecheck` provides type errors at the level of the language being implemented. It does not depend on OCAML type system. This means that the implementation can give an error in terms of the interpreted language. Finally, note that OCAML logical `and` adopts a short-circuit evaluation rules: in `e1 && e2`, expression `e1` is evaluated first, and if it returns false, expression `e2` is not evaluated at all. A similar consideration applies to OCAML logical `or`.

Finally, we are ready to handle the actual point of this section: defining the interpreter of conditionals. The two Boolean constants are easy:

```
# let rec eval (e : exp) : evalT =
         match e with
         | CstTrue -> Bool(true)
         | CstFalse -> Bool(false)
         :
```

The interpreter is quite simple since we have chosen to represent explicitly boolean values. Now we have to present the rule for conditional expressions.

```
# let rec eval (e : exp) : evalT =
         match e with
        :
         | ifthenelse(e1,e2,e3) ->
                let g = eval(e1) in
                  match (typecheck("bool", g), g) with
                    | (true, Bool(true)) -> eval(e2)
                    | (true, Bool(false)) -> eval(e3)
                    | (_, _) -> failwith ("nonboolean guard")

         :
```

We conclude by presenting the overall structure of the interpreter.

```
# let rec eval  e =
      match e with
      | CstInt(n) -> Int(n)
      | CstTrue -> Bool(true)
      | CstFalse -> Bool(false)
      | Iszero(e1) -> is_zero(eval(e1))
      | Eq(e1, e2) -> int_eq(eval(e1), eval(e2))
      | Times(e1,e2) -> int_times(eval(e1), eval(e2))
      | Sum(e1, e2) -> int_plus(eval(e1), eval(e2))
      | Sub(e1, e2) -> int_sub(eval(e1), eval(e2))
      | And(e1, e2) -> bool_and(eval(e1), eval(e2))
      | Or(e1, e2) -> bool_or(eval(e1), eval(e2))
      | Not(e1) -> bool_not(eval(e1))
      | Ifthenelse(e1,e2,e3) ->
                let g = eval(e1) in
           (match (typecheck("bool", g), g) with
            |(true, Bool(true)) -> eval(e2)
            |(true, Bool(false)) -> eval(e3)
            |(_, _) -> failwith ("nonboolean guard"))
      | _ -> failwith ("run-time error");;
```

We can exercise our interpreter in the OCAML top level loop as follow:

```
# let e1= Eq(CstInt(5),CstInt(5));;
val e1 : exp = Eq (CstInt 5, CstInt 5)
# let e2 = Sum(CstInt(5),CstInt(1));;
val e2 : exp = Sum (CstInt 5, CstInt 1)
# let e3 = CstInt(2);;
val e3 : exp = CstInt 2
# let e = Ifthenelse(e1,e2,e3);;
val e : exp =
  Ifthenelse (Eq (CstInt 5, CstInt 5), Sum (CstInt 5, CstInt 1), CstInt 2)
# eval e;;
- : evT = Int 6
#
```

Let us consider another example of our interpreter at work.

```
# let e1 = CstInt(0);;
val e1 : exp = CstInt 0
# let e2 = CstInt(5);;
val e2 : exp = CstInt 5
# let e3 = Or(e1,e2);;
val e3 : exp = Or (CstInt 0, CstInt 5)
# eval e3;;
Exception: Failure "run-time error ".
#
```

The run-time exception is raised because the interpreter uses an eager evaluation strategy: the parameters `e1` and `e2` are both evaluated before calling the OCAML short circuited or.

Exercise Modify the interpreter and implement the evaluation of conjunction and disjunction in terms of short-circuited evaluation strategies.

2.7 Functions

Now that we have basic expressions and conditional expressions, let us grow to have a complete programming language by adding functions. A function is a piece of code identified by a name which is able to exchange information with the rest of the program using parameters. This concept translates into two different linguistic mechanisms. The first, definition (or declaration) of a function, and its use (or call).

Imagine we are modeling a simple programming environment. The developer defines functions in a definition window, and uses them in an interaction window. For historic reasons, the interaction window is also called a REPL or read-eval-print loop. For simplicity, let us assume all function definitions go in the definition window only (we will relax this constrain later) and all stand-alone expressions in the interaction window only. Thus, running a program simply loads definitions. Our interpreter will correspond to the interaction window prompt and assumes it has been supplied with a set of definitions.

A set of definitions suggests no ordering, which means, presumably, any definition can refer to any other. To keep things simple, let us just consider functions of one argument without recursion. Here are examples.

```
fun double(x) {return x + x }
fun quad(x){return double(double(x))}
fun const5(x){return 5}
```

What are the parts of a function definition? It has a name (above, double and quad), which we will represent as a string ("double", etc.); its formal parameter or argument has a name (e.g., x), which we can model as a string ("x"); and it has a body. We will determine the body representation in stages, but let us start to lay out our algebraic datatype for function definitions.

```
type funName = Fid of string;;
type funArg = Ide of string;;
type funBody = Body of exp;;
type funDef = Fun of funName*funArg*funBody
```

What is the body? Our design choice now is that it has the form of an expression: for instance, the body of `const5` can be represented as `CstInt(5)`. But representing the body of `double` requires something more: not just addition (which we have), but also `x`. You are probably used to calling this a variable, but we will not use that term for now. Instead, we will call it an identifier. Finally, let us look at the body of `quad`. It has yet another new construct: a function application. Be very careful to distinguish between a function definition, which describes what the function is, and an application, which uses it. The argument (or actual parameter) in the inner application of `double` is `x`; the argument in the outer application is `double(x)`. Thus, the argument can be any complex expression.

We have to design a better datatype to represent functions. Clearly we are extending what we had before (because we still want all of arithmetic and Boolean operators).

```
type exp =
        :
     | Val of string
     | Apply of funName * exp
```

Since we assume a global set of function declarations, then we simply need a constructor to represent function calls. When we apply a function by giving it a value for its parameter, we are in effect asking it to replace all instances of that formal parameter in the body i.e., the identifiers with the same name as the formal parameter with that value. Application consists of two parts: the functions name, and its argument. We have already agreed that the argument can be any full-fledged expression (including identifiers and other applications).

Using these definitions, its instructive to write out the representations of the examples we defined above:

```
Fun(Fid("double"),Ide("x"), Body(Sum(Val("x"),Val("x"))))
Fun(Fid("quad"),Ide("x"), Body(Apply(Fid("double"),
    Apply(Fid("double"), Val("x")))))
Fun(Fid("const5"), Ide("x"), CstInt(5))
```

Finally we also need to choose a representation for a set of function definitions. It is convenient to represent these by a list.

```
type funDecl = list of funDef;;
```

Now we are ready to tackle the interpreter proper. First, let us remind ourselves of what it needs to consume. Previously, it consumed only an expression to evaluate. Now it also needs to take a list of function definitions.

```
eval : exp * funDecl -> evT
```

Let us revisit our interpreter of expressions In the case of integer and boolean constants , clearly we still return the corresponding values. In the case of operations, we still need to recur (because the sub-expressions might be complex), but which set of function definitions do we use? Because the act of evaluating an expression neither adds nor removes function definitions, the set of definitions remains the same, and should just be passed along unchanged in the recursive calls of the interpreter. Similarly in the case of conditional expressions.

Now let us tackle application. Given the name of the function we have to look up in the list of function definitions the corresponding definition, for which we will assume we have a helper function of this type available. We also need other helper functions whose role is to pick up the argument and the body of the function.

```
getFunDef: funName*funDecl -> funDef
getFunArg: funDef -> funArg
getFunBody: funDef -> funBody
```

Here is the code of these helper functions.

```
let rec getFunDef (n, l) = match (n, l) with
  | (Fid(s1), Fun(Fid(s2), Ide(af), Body(b))::ls) ->
      if s1 = s2 then Fun(Fid(s2), Ide(af), Body(b))
                 else prova(n,ls)
  | (Fid _, []) -> failwith("run-time error");;

let getFunArg f = match f with
  | Fun(Fid(nf), Ide(na), Body(b)) -> Ide(na);;

let getFunBody f = match f with
  | Fun(Fid(nf), Ide(na), Body(b)) -> Body(b);;
```

Assuming we find a function of the given name, we need to evaluate its body. However, remember what we said about identifiers and parameters? We must search-and-replace , a process that is usually called substitution. Let f be a function with a single formal parameter, x, and let e be an expression of a type that is compatible with that of x. A call to f with an actual parameter a is semantically equivalent to the execution of the body of

f in which all occurrences of the formal parameter, x, have been replaced by the value of the expression a. As can easily be seen, it is a very simple rule. It reduces a function call to the syntactic operation of expanding the body of the function after a textual substitution of the value of the actual parameter for the formal parameter.

This notion of substitution, however, is not a simple concept because it must take into account the fact it might have to deal with several different identifiers with the same name. Hence it is sufficiently important that we should talk first about substitution before returning to the interpreter.

Substitution is the act of replacing a name (in this case, that of the formal parameter) in an expression (in this case, the body of the function) with another expression (in this case, the value of the actual parameter). Its header is of the form:

```
Subst: exp*funArg *exp -> exp
```

The first argument is what we want to replace the name with; the second is at what name we want to perform substitution; and the third is in which expression we want to do it.

Suppose we want to substitute `3` for the identifier `x` in the bodies of the three example functions above. What should it produce? In `double`, this should produce as result `3 + 3`; in `quad`, it should produce `double(double(3))`; and in `const5`, it should produce `5` (i.e., no substitution happens because there are no instances of the parameter occurs in the body). A common mistake is to assume that the result of substituting, e.g., `3` for `x` in `double` is `fun double(x)return 3 + 3`. This is incorrect. We only substitute at the point when we apply the function, at which point the functions invocation is replaced by its body. The header enables us to find the function and ascertain the name of its parameter; but only its body participates in evaluation.

These examples already tell us what to do in almost all the cases. Given a number, theres nothing to substitute. If it is an identifier, we have to to replace the identifier if it is the one we are trying to substitute, otherwise leave it alone. In the other cases, descend into the sub-expressions, performing substitution.

Before we turn this intuition into code, there is an important case to consider. Suppose the name we are substituting happens to be the name of a function. Then what should happen?

There are many ways to approach this question. One is from a design perspective: function names live in their own world , distinct from ordinary program identifiers. Some languages (such as C) take this perspective, and partition identifiers into different namespaces depending on how they are used. In other languages, there is no such distinction; indeed, we will examine such languages later. For now, we will take a pragmatic viewpoint. If we evaluate a function name, it would result in an integer or a boolean. However, these cannot name functions. Therefore, it makes no sense to substitute in that position, and we should leave the function names unchanged. (Thus, a function could have a parameter named x as well as refer to another function called x, and these would be kept distinct.)

Now we have discussed all our decisions, we can provide the OCAML code of the substitution function.

```
let rec subst  (e1, arg, e2) =
  match e2 with
   | CstInt(n) -> CstInt(n)
   | CstTrue -> CstTrue
   | CstFalse -> CstFalse
   | Iszero(e) -> Iszero(subst(e1, arg, e))
   | Eq(ea, eb) -> Eq(subst(e1,arg,ea), subst(e1,arg,eb))
   | Times(ea,eb) ->
      Times(subst(e1,arg,ea),subst(e1,arg,eb))
   | Sum(ea, eb) ->
      Sum(subst(e1,arg,ea), subst(e1,arg,eb))
   | Sub(ea, eb) ->
      Sub(subst(e1,arg,ea), subst(e1,arg,eb))
   | And(ea, eb) ->
      And(subst(e1,arg,ea), subst(e1,arg,eb))
   | Or(ea, eb) -> Or(subst(e1,arg,ea), subst(e1,arg,eb))
   | Not(e) -> Not(subst(e1, arg, e))
   | Ifthenelse(eg,et,ef) ->
      Ifthenelse(subst(e1, arg, eg),
                 subst(e1, arg, et),
                 subst(e1, arg, ef))
   | Val(s) -> (match arg with
                   | Ide(s) -> e1
                 | _ -> Val(s))
    | Apply(f,e) -> Apply(f,subst(e1, arg, e))
    | _ -> failwith ("run-time error");;
```

An example of substitution at work is:

```
# subst(CstInt(3),Ide("x"),Sum(Val("x"),Val("x")));;
- : exp = Sum (CstInt 3, CstInt 3)
```

Now that we have completed the definition of substitution, let us complete the interpreter. Substitution was a heavyweight step, but it also does much of the work involved in applying a function. One is tempting to write:

```
eval(Apply(f, a), dcl) ->
  let t = getFunDef(f, dcl) in
        subst(a, getFunArg(t),getFunBodyExp(t))
```

Nice tempting, but wrong. Reason from the types. What does the interpreter return? Expressible values (`evT`). What does substitution return? Expressions! For instance, when we substituted in the body of double, we got back the representation of `3 + 3`. This is not a valid answer for the interpreter. Instead, it must be reduced to an answer. That, of course, is precisely what the interpreter does:

```
eval(Apply(f, a), dcl) ->
 let t = getFunDef(f, dcl) in
  eval(subst(a, getFunArg(t),getFunBodyExp(t)),dcl)
```

This leaves out only one case: identifiers. What could possibly be complicated about them? They should be just about as simple as numbers! And yet we have put them off to the very end, suggesting that their management is complex or critical.

Let us suppose we had defined double as `fun double(x) return x + y;`. When we substitute `5` for `x`, this produces the expression `5 + y`. So far so good, but it is left the substitution for `y`. As a matter of fact, it should be clear this definition of double is erroneous: there is no binding for `y`. The identifier y is said to be free, an adjective that in the setting of programming language implementation has negative connotations. In

other words, the interpreter should never meet an identifier. All identifiers are indeed parameters that have already been substituted (known as bound identifiers, a positive connotation) before the interpreter ever sees them. As a result, there is only one possible response the interpreter gives when an identifier is met:

```
eval(Val("x"), dcl) ->
 failwith("run-time error: unbound name")
```

We can now present the overall structure of the interpreter. Note that the interpreter needs to take as input the current list of function definitions (dcl).

```
let rec eval  (e, dcl) =
 match e with
 | CstInt(n) -> Int(n)
 | CstTrue -> Bool(true)
 | CstFalse -> Bool(false)
 | Iszero(e1) -> is_zero(eval(e1,dcl))
 | Eq(e1, e2) -> int_eq(eval(e1,dcl), eval(e2,dcl))
 | Times(e1,e2) ->
       int_times(eval(e1,dcl), eval(e2,dcl))
 | Sum(e1, e2) -> int_plus(eval(e1,dcl), eval(e2,dcl))
 | Sub(e1, e2) -> int_sub(eval(e1,dcl), eval(e2,dcl))
 | And(e1, e2) -> bool_and(eval(e1,dcl), eval(e2,dcl))
 | Or(e1, e2) -> bool_or(eval(e1,dcl), eval(e2,dcl))
 | Not(e1) -> bool_not(eval(e1,dcl))
 | Apply(f, a) ->
    let t = getFunDef(f, dcl) in
     let es = subst(a,getFunArg(t),getFunBodyExp(t)) in
            eval(es,dcl)
 | Val("x") -> failwith("run-time error: unbound name")
 | Ifthenelse(e1,e2,e3) ->
        let g = eval(e1,dcl) in
          (match (typecheck("bool", g), g) with
                | (true, Bool(true)) -> eval(e2,dcl)
           | (true, Bool(false)) -> eval(e3,dcl)
           | (_, _) -> failwith ("nonboolean guard"))
 | _ -> failwith ("run-time error");;

 val eval : exp * funDef list -> evT = <fun>
```

We now exercise in the usual REPL style our interpreter.

```
# let myf = Fun(Fid("double"),Ide("x"), Body(Sum(Val("x"),Val("x"))));;
val myf : funDef = Fun (Fid "double", Ide "x", Body (Sum (Val "x", Val "x")))
# let dclList = [myf];;
val dclList : funDef list =
  [Fun (Fid "double", Ide "x", Body (Sum (Val "x", Val "x")))]
# let myp : exp = Apply (Fid "double", CstInt 3);;
val myp : exp = Apply (Fid "double", CstInt 3)
# eval(myp, dclList);;
- : evT = Int 6
# # let cond = Eq(CstInt(5),CstInt(5));;
val cond : exp = Eq (CstInt 5, CstInt 5)
# let e = CstInt(2);;
val e : exp = CstInt 2
# let myp1 = Ifthenelse(cond,myp,e);;
val myp1 : exp =
  Ifthenelse (Eq (CstInt 5, CstInt 5), Apply (Fid "double", CstInt 3),
   CstInt 2)
# eval(myp1,dclList);;
- : evT = Int 6
#
```

There is more. Above we declared the substitution function as `subst: exp*funArg *exp -> exp`. Now suppose we apply `double` to `1 + 2`. This would substitute `1 + 2` for each occurrence of `x`, resulting in the expression `(1 + 2) + (1 + 2)` for interpretation. Is this necessarily what we want? When you learned elementary algebra in school, you may have been taught to do this differently: first reduce the argument to an answer (in this case `3`), then substitute the answer for the parameter. This notion of substitution might have the following type instead `subst: evT*funArg *exp -> exp`. The version of substitution we have does not scale up this language due to a subtle problem known as name capture . Fixing substitution is complex, subtle, and an exciting intellectual endeavor. If you are interested, however, read about the note on OCAML Operational Semantics, which outlines the basic idea for defining substitution correctly.

Exercise Modify our interpreter to substitute names with values, not expressions.

We have indeed addressed a profound distinction in programming languages. The act of evaluating arguments before substituting them in functions is called eager application, while that of deferring evaluation is called lazy and has some variations. For now, we will actually prefer the eager semantics, because this is what most mainstream languages adopt. Later, we will return to talking about the lazy application semantics and its implications.

Though we have a working definition of functions, you may feel a slight discontent about it. When the interpreter sees an identifier, you might have had a sense that it needs to look it up . Not only did it not look up anything, we defined its behavior to be an error. While absolutely correct, this is also a little surprising. More importantly, we write interpreters to understand and explain programming languages, and this implementation might strike you as not doing that, because it doesnt match our intuition. There is another difficulty with using substitution, which is the number of times we traverse the source program. It would be nice to have to traverse only those parts of the program that are actually evaluated, and then, only when necessary. But substitution traverses everything unvisited branches of conditionals, for instance and forces the program to be traversed once for substitution and once again for interpretation. There is yet another problem with substitution, which is that it is defined in terms of representations of the program source. Obviously, our interpreter has and needs access to the source, to interpret it. However, other implementations such as compilers have no need to store it for that purpose. Compilers might store versions of or information about the source for other reasons, such as reporting runtime errors, and Just in Time (JIT) compilers may need it to re-compile on demand. Our substitution indeed works as a JIT compiler. It would be nice to employ a mechanism that is more portable across implementation strategies.

3 OCaml Operational Semantics: An Overview

An OCaml expression can do several things:

- It can return a value
- It can loop forever
- It can print out a string to standard output or read one in from standard input
- It can read and write to files
- It can raise an exception
- It can ...

To begin to understand how an OCaml expression executes, we are going to focus exclusively on understanding what value the expression returns. The other things – reading, printing, raising exceptions, etc. – are called the effect of the expression. We are going to ignore that stuff for now. One of the great things about OCaml is that for this core fragment, the rules for evaluating expressions are incredibly simple. They are much,

much, much simpler than for any equivalent fragment of C or Java. We call the rules for evaluating a language its operational semantics.

Consider the following two expressions:

```
let x = 30 in
let y = 12 in
x+y;;

let a = 30 in
let b  = 12 in
a+b;;
```

Is the resulting expression any different? No! Both expressions evaluate to 42.

Now consider the following two expressions:

```
let x = 30 in
let y = 12 in
x+y;;

let x = 30 in
let x  = 12 in
x+x;;
```

Are they any different? Yes! The expression on the left evaluates to 42 as before. The expression on the right evaluates to 24. One more example:

```
let x = 30 in
let x = 12 in
x+x;;

let y = 30 in
let x  = 12 in
x+x;;
```

Different? No. Both evaluate to 24.

OCaml is a lexically scoped, also known as statically scoped language. This means that the value associated with a variable (say x) is given by the closest enclosing let (or function variable) declaration of that same variable.

An example:

```
let x = 30 in  <---- variable x is bound to 30
let y = 15 in
let x = 12 in  <---- variable x is bound to 12
let y = -4 in
let z = 16 in
x + y + z       <----  the x on this line refers to 12
```

The scope of a variable (say `x`) is all those places in the program text where you can refer to it. Hence, as mentioned above, the scope of any variable introduced (or bound) by a let expression is the body of that let expression, up to the point where another variable with the same name is introduced. In the example above the scope of the first variable named `x` (bound to 30) extends until the second variable `x` (bound to 12) is introduced. Here is another example.

```
let x = 30 in
let f (y:int) : int = y + x in
let x = 12 in
f x                (* here *)
```

What is the result of evaluating this expression? Because OCaml is statically scoped, the value of `x` within the body of `f` never changes. It remains 30 (the value assigned to it in the nearest enclosing let declaration). Hence, the function `f` is really just a function that always adds 30 to its argument. On the line marked (*here*), we see that `f` is applied to an argument (also named `x`). The value of that argument is 12 (because the nearest enclosing definition of `x`, on the line above, binds `fx` to 12). Hence, the result of the expression is 12 plus 30, which is 42.

How should you think about the two definitions of `x` in the code above? Think of the two definitions as defining completely different variables that just happen to have the same name.

Of course, it is not good programming style to reuse the same name for two different variables, especially non-descript names like `x`. A key principle of programming in ML is that these local names do matter: You can pick any name you want and it does not change the meaning of your program. However, when you change a definition of a name, you must change all the uses of the name consistently. For example, we can convert this OCaml expression:

```
let x = 30 in
let f (y:int) : int = y + x in
let x = 12 in
f x;;
```

to this espression

```
let z = 30 in
let f (y:int) : int = y + z in
let x = 12 in
f x ;;
```

Notice that we change the x to z at the point of definition and each of the uses also change from x to z. We can also change the above to

```
let z = 30 in
let x (y:int) : int = y + x in
let f = 12 in
x f ;;
```

One more example. We can change this:

```
let x = 2 in
let x = (let x = x + x in x + 7) in
x * 3 ;;
```

to this:

```
let a = 2 in
let c = (let b = a + a in b + 7) in
c * 3 ;;
```

The second piece of code uses a different variable name for each let declaration. Hence, you can clearly see ML's scoping rules.

In general, the act of changing one expression in to another expression that differs only because of a consistent renaming of variables is called alpha-conversion. The two expressions are also called alpha-equivalent expressions.

Consider the following expression:

```
let z = 30 in       (* 1 *)
let y = z + x in   (* 2 *)
y
```

In line 2, the variable `x` is called a free variable within the expression. It is free because there is no enclosing let expression (or function declaration) that defines it. In contrast, on line 2, `z` is called a bound variable. It is bound because it is within the scope of a definition (namely the definition on line 1). In the following expression, `x` is a bound variable. We might call line 0 the binding site of `x`.

```
let x = 17 in       (* 0 *)
let z = 30 in       (* 1 *)
let y = z + x in   (* 2 *)
y
```

Now consider again this simple program

```
let z = 30 in
let f (y:int) : int = y + z in
let x = 12 in
f x ;;
```

When describing the function `f`, we said that `f` was just a function that always added 30 to its argument. Why did we say that? We said that because we understood that we could evaluate a let expression simply by substituting the computed value on the right-hand side of the let for the variable, everywhere within the scope of that variable. In other words, the expression above is equivalent to the following as we are allowed to substitute the value 30 for all the uses of `z`.

```
let f (y:int) : int = y + 30 in
let x = 12 in
f x ;;
```

More generally, given any value v and any expression e, the notation: $e[v/x]$ is used to refer to the expression that results from substituting value v for all of the free occurrences of x within the expression e. For example, if e is the expression `let y = x + x in x * y` then the result of the substitution $e[17/x]$ is `let y = 17 + 17 in 17 * y`. As another example, suppose e is `let y = x + x in let x = 9 in x * y` then the result of the substitution of $e[2/x]$ is: `let y = 2 + 2 in let x = 9 in x * y`. We must be careful to only substitute for the free occurrences of a value, not the bound ones.

Having explained what free and bound variables are and having defined substitution, we are now in a position to define an operational semantics for the core expressions that make up ML. The operational semantics explains how to evaluate an ML expression step by step. If you want to reason about what your ML programs do, you will use this operational semantics.

When we want to say that one expression evaluates to another expression in a single computation step, we write:

$$e_1 \longrightarrow e_2$$

The arrow above is usually read aloud as "steps to" ore evaluates to . Please note that the arrow $\longrightarrow$ is not a part of any program – it is notation that we use to talk about programs. To talk about several execution steps in a row, we might write:

$$\begin{aligned} e_1 &\longrightarrow e_2 \\ &\longrightarrow e_3 \\ &\longrightarrow e_4 \end{aligned}$$

And if we don't care about the intermediate steps but just want to talk about 0 or more steps all at once, we write $e_1 \Longrightarrow e_2$.

The evaluation of let expressions is slightly more interesting than the other expressions we considered so far. The evaluation of let expression is where substitution comes in. Given a declaration `let var = exp1 in exp2`, we simply evaluate `exp1` until we get a value. Then we substitute that value for the variable in `exp2` and continue evaluating. Here are two examples.

```
let x = 2 + 5 in let y = x + x in y * 3       -->  let x = 7 in let y = x + x in y * 3
                                              -->  let y = 7 + 7 in y * 3
                                              -->  let y = 14 in y * 3
                                              -->  14 * 3
                                              -->  42

let x = (let x = 2 - 1 in x + x) in x * 21    -->  let x = (let x = 1 in x + x) in x * 21
                                              -->  let x = (1 + 1) in x * 21
                                              -->  let x = 2 in x * 21
                                              -->  2 * 21
                                              -->  42
```

Functions are evaluated much the same way as let declaration in that they use substitution. In general, given a function application expression with the form: `exp1 exp2`. We evaluate it by first evaluating `exp1` until we get a function value `f`. Then we evaluate `exp2` until we get an argument value `v`. Then we substitute `v` for the parameter into the body of `f`. For example, here is a declaration of the function `let add1 (x:int) : int = x + 1`. We now evaluate an expression involving `add1`:

```
add1 3 + add1 4  -->  (3 + 1) + add1 4
                 -->  4 + add1 4
                 -->  4 + (4 + 1)
                 -->  4 + 5
                 -->  9
```

Notice that we evaluate the arguments of + from left to right. When it comes to evaluating the application of `add1 3`, we do so by substituting `3` for the parameter `x` in the body of the add function. In other words, the body is `x + 1` and when we substitute `3` for `x` in that body, we obtain `3 + 1`.

Multi-argument functions work the same way as multi-argument functions are just abbreviations for single argument functions. For example, consider the addition function `let add (x:int) (y:int) = x + y` It can be thought of as an abbreviation for the function `let add (x:int) = (fun y -> x + y)`. Which is also equal to the function: `let add = (fun x -> (fun y -> x + y))`.

Now consider what happens when we evaluate the expression `add 2 3`:

```
add 2 3  ⟶  (fun x -> (fun y -> x + y)) 2 3

         ⟶  (fun y -> 2 + y) 3

         ⟶  2 + 3

         ⟶  5
```

On line 1, we substituted the definition of add in place of the variable add. Then on lines 2 and 3, we substituted arguments for parameters one at a time. Typically, we will not be so verbose when we work out how to evaluate applications of multi-argument functions and we will simply substitute all of the arguments at once (unless the function is only partially applied).

Intuitively, the last example showed what happens when we try to evaluate functions that return other functions as results. We can also evaluate functions that take other functions as arguments. For example, below we define the function pipe that pipes a value in to another function. We have:

```
let pipe (x:'a) (f:'a -> 'b) :  'b = f x;;
```

Watch how we evaluate an expression that uses such a function

```
pipe 3 (fun x -> 2 * x)  ⟶  (fun x -> 2 * x) 3

                         ⟶  2 * 3

                         ⟶  6
```

On line 1, we have substituted the function value `(fun x -> 2 * x)` for the function parameter `f` in the definition of `pipe`. Finally, here is an example involving a recursive function `let rec fact (x:int) :  int = if x <= 0 then 0 else (fact (x-1)) + x`. Watch what happens when we evaluate the expression.

```
fact 2  ⟶  if 2 <= 0 then 0 else (fact (2-1)) + 2
        ⟶  if false then 0 else (fact (2-1)) + 2
        ⟶  (fact (2-1)) + 2)
        ⟶  (fact 1) + 2
        ⟶  (if 1 <= 0 then 0 else (fact (1-1)) + 1) + 2
        ⟶  (if false then 0 else (fact (1-1)) + 1) + 2
        ⟶  ((fact (1-1)) + 1) + 2 --> ((fact 0) + 1) + 2
        ⟶  ((if 0 <= 0 then 0 else (fact (0-1) + 0)) + 1) + 2
        ⟶  ((if true then 0 else (fact (0-1) + 0)) + 1) + 2
        ⟶  ((0) + 1) + 2
        ⟶  1 + 2
        ⟶  3
```

Right. It gets pretty verbose very quickly. However, it is important to understand exactly how the language works. If you understand this operational semantics, then you can use it to analyze your ML program step by step if it does something you didn't expect. Of course, we can typically run through many steps in a row quickly in our heads. For example, using the multi-step relation, it is equally correct to skip a some of the intermediate steps and write down the following sequence

```
fact 2  ⟹  (fact 1) + 2
        ⟹  ((fact 0) + 1) + 2
        ⟹  ((0) + 1) + 2
        ⟹  3
```

There is one more basic concept that we must learn. The operational semantics for ML programs state that you may call a function, substituting the arguments for the parameters provided those arguments have been evaluated to a value.

Consider the function `let inc (x:int) :  int = x + 1`. We know that:

$$\texttt{inc 3} \longrightarrow \texttt{3 + 1} \longrightarrow \texttt{4}$$

And consequently, we may conclude that $inc3 == 4$. But what about the expression `inc (y + 3)` where y is a free integer variable? We might like to reason as follows:

$$inc(y + 3) == (y + 3) + 1 == y + 4$$

Indeed, this sort of proof is valid. It is valid because we know that no matter what value we substitute for y, $y + 3$ will evaluate to some other value (call it v) and then we can apply `inc` to v and continue.

That kind of reasoning leads one to believe that perhaps it is the case that if you have a function and an argument expression to that function then you can always substitute the argument expression for the parameter in the body of the function. In other words, one might conjecture that the following equational rule is true:

For any expression exp and for any function f defined as `let f x = body` The following expressions are equivalent

$$fexp == body[exp/x]$$

We calle this equality the *Bad Eval Equality*. Because of its name you have a pretty big evidence that the above equation is NOT a valid equivalence. Here is a counter-example. Consider the function `forever`, which loops forever and the function `one`, which, when called, always returns the value `1`. Both functions are defined below:

```
let rec forever () : int = forever ();;
let one (x:int) : int = 1;;
```

Now, using the Bad Eval Equality we can immediately prove that:

$$one(forever()) == 1$$

But that is obviously incorrect because the left-hand side of the equation loops forever and never returns a value whereas the right-hand side returns the value 1 immediately. Hence, the two expressions are not equal.

The problem with the Bad Eval Equality is that it allows any expression to be substituted for a function argument. If it was a bit more restrictive and only allowed for substitution of those function argument expressions that are guaranteed to always terminate and produce a value then it would be a good notion for equality

An expression that always terminates and produces a value is called a valuable expression.

The following expressions are all examples of valuable expressions:

- constants: any constant (e.g.: `1, 2, "hello", 3.5, 'a', []`, etc.) is valuable
- values: any value (e.g.: `(2,3), [14], None`, `fun x -> x + x`) is valuable
- variables: any variable x is valuable because such variables will be replaced by values when the code is executed.
- pairs: if `exp1` and `exp2` are both valuable expressions then pair expression `(exp1, exp2)` is also valuable
- lists: if `exp1` and `exp2` are both valuable expressions then list expression `exp1::exp2` is also valuable
- any other data type constructor: if `exp` is a valuable expression and `C` is a data type constructor then `C exp` is also valuable
- if statements: an if statement `if exp1 then exp2 else exp3` is valuable if `exp1`, `exp2` and `exp3` are all valuable expressions. (pattern matching statements are similar if they are not missing any pattern matching branches).

- total functions: if `f` is a total function and `exp` is a valuable expression then `f exp` is a valuable expression as well.

In the last case, we referred to the idea of a total function. A total function is a function that when supplied with any argument (of the correct type) always terminates and returns a value. Instead, a partial function is a function that when supplied with some argument (of the correct type) will fail to terminate or will raise an exception.

For example, the function `inc` defined above is a total function — it always terminates and adds one. Many of the built-in operations such as +, -, etc are total functions. However, some are not. For instance, the function `List.hd`, which returns the first element of a list, is a partial function as it raises an exception when its argument is the empty list.

Recursive functions can be total functions. For instance, a list-processing function like `List.length` is total – it will terminate and produce an integer when given any list. Here is the definition of `List.length`.

```
let rec length (xs:'a list) : int =   match xs with
      | [] -> 0
      | hd::tail -> 1 + length tail ;;
```

We can tell that `length` is a total function because:

- It does not itself call any partial functions.
- It does not have any incomplete pattern matching.
- Whenever length is called recursively, it does so on a strictly smaller argument (and there are no infinite sequences of lists that get smaller and smaller forever). In particular, in the second branch of the match statement, the recursive call `length tail` involves an argument with length that is strictly smaller than the input, which is `hd::tail`.

Consequently, `length` terminates on all inputs, never raises an exception and always returns a value.

The following are examples of partial functions.

```
let rec forever (xs:'a list) : int =   match xs with
      | [] -> 0
      | hd::tail -> 1 + forever (1::tail) ;;

let rec oops (xs:'a list) (acc:int) (num:int) : int =   match xs with
      | [] -> acc / num
      | hd::tail -> oops tail (acc+hd) (num+1) ;;
```

Above, the function `forever` loops forever because we pass the recursive call an argument that is not shorter than the input list. Hence, forever is a partial function. (Note, it does terminate in one case – when given the empty list – but it must terminate in all cases to be total.) The function `oops` has a subtler problem. It might raise a divide-by-zero error if the denominator num turns out to be zero in the base case of the recursion. Therefore, `oops` is not total.

As an aside in Java the most interesting expressions, like method calls and field dereferences, are partial (not total) and might raise an exception because Java's type system does not distinguish between null and non-null values (like ML's type system distinguishes between values with option type and non-option type). In contrast, most (or at least many, if you follow the style advocated in this class) functions in ML are total. This is one of the reasons that Java is substantially harder to reason about than ML in theory, and in practice you see the same thing: lots of null pointer dereference bugs show up in real applications.

Finally, we can now state our good, valuable equality rule:

Eval Value Equality : For any valuable expression e and for any function f of the form `let f x = body`. The following expressions are equivalent

$$f exp == body[exp/x]$$

All identifiers are indeed parameters that have already been substituted (known as bound identifiers, a positive connotation) before the interpreter ever sees them. As a result, there is only one possible response the interpreter gives when an identifier is met:

4 Names, Bindings and Environment

The evolution of programming languages can be seen in large measure as a process which has led, by means of appropriate abstraction mechanisms, to the definition of formalisms that are increasingly distant from the physical machine. In this context, names play a fundamental role. A name, indeed, is nothing more than a (possibly meaningful) sequence of characters used to represent some other thing. They allow the abstraction either of aspects of data, for example using a name to denote a location in memory, or aspects of control, for example representing a set of commands with a name. The correct handling of names requires precise semantic rules as well as adequate implementation mechanisms.

Now we will analyse these issues. We will, in particular, look at the concept of environment and the constructs used to organise it. We will also look at visibility (or scope) rules. Let us immediately observe how, in languages with procedures, in order to define precisely the concept of environment one needs other concepts, related to parameter passing.

When we declare a new variable in a program `int x = 0;` or we define a function: `int foo = x==1`, we introduce new names, such as `x` and `foo` to represent an program entity (an integer variable and a function in our example). A name is therefore nothing more than a sequence of characters used to represent, or denote, another program entity.

In most languages, names are formed of identifiers, that is by alphanumeric tokens, moreover other symbols can also be names. For example, + and - are names which denote, in general, primitive operations.

Even though it might seems obvious, it is important to emphasise that a name and the entity it denotes are not the same thing. A name, indeed, is just a character string, while its denotation can be a complex objects such as a variable, a function, a type, and so on. And in fact, a single entity can have more than one name (in this case, one speaks of aliasing), while a single name can denote different entities at different times. When, therefore, we use, as we may, the phrase the variable x or the phrase the function foo , it should be remembered that the phrases are abbreviations for the variable with the name x and the function with the name foo . More generally, in programming practice, when a name is used, it is almost always meant to refer to the object that it denotes.

The use of names implements a first, elementary, data abstraction mechanism. For example, when, in an imperative language, we define a name using a variable, we are introducing a symbolic identifier for a memory location; therefore we are abstracting from the low-level details of memory addresses. If, then, we use the assignment command `x = 2`; the value 2 will be stored in the location reserved for the variable named `x` .

At the programming level, the use of the name avoids the need to bother with whatever this location is. The correspondence between name and memory location must be guaranteed by the implementation. We will use the term environment to refer to that part of the implementation responsible for the associations between names and the entities that they denote. Names are fundamental even for implementing a form of control abstraction. A function is nothing more than a name associated with a set of expressions, together with certain visibility rules which make available to the programmer its sole interface (which is composed of the procedures name and possibly some parameters).

4.1 Expressions with Let-binding and Static Scope

We start by considering a simple expression language with let binding of the form `let x = e1 in e2`. As usual, we introduce the corresponding algebraic data type

```
# type ide = string;;
type ide = string

# type exp = CstInt of int
                | CstTrue
                | CstFalse
                | Den of ide
                | Sum of exp * exp
                | Times of exp * exp
                | Ifthenelse of exp * exp * exp
                | Eq of exp * exp
                | Let of ide * exp * exp ;;
type exp =
    CstInt of int
  | CstTrue
  | CstFalse
  | Den of ide
  | Sum of exp * exp
  | Times of exp * exp
  | Ifthenelse of exp * exp * exp
  | Eq of exp * exp
  | Let of ide * exp * exp
```

Our syntax clearly distinguishes between variable declaration (the `Let` constructor) and variable occurrence and usage (the `Den` constructor). We adopted the `Den` constructor to indicate the entity the variable refers to, namely its denotation. For instance, the OCaml program:

```
let x = 30 in
  let y = 12 in
     x + y;;
```

will be represented as follows

```
# Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
- : exp = Let ("x", CstInt 30, Let ("y", CstInt 12,
Sum (Den "x", Den "y")))
#
```

4.2 Introducing the environment

The intuition is the interpreter looks up an identifier in some sort of directory. The directory records the intent to substitute, without actually rewriting the program source. The resulting run-time data structure, which is called an environment, avoids the need for source-to-source rewriting and maps nicely to low-level machine representations. Each name association in the environment is called a binding. This does not mean our study of substitution was useless; to the contrary, many tools that work over programs such as compilers and analyzers use substitution. Just not for the purpose of evaluating it at run-time. Observe carefully that what we are changing is the implementation strategy (and the corresponding run-time structures) for the programming language, not the language itself. Therefore, none of our datatypes for representing programs should change, neither and this is the critical part should the results that the interpreter provides. Ideally, we should prove that the two interpreters behave the same, which is a good topic for more advanced study.

Let us first define the data structure implementing the environment. An environment is a collection of names associated with...what? When we considered the substitution model we discussed that we want substitution to map names to results, corresponding to an eager function application strategy. Therefore, the environment should map names to *expressibile values*

We extend expressible values (`evT`) with the distinguished constructor `Unbound` to manage the case where a certain variable does not refer to any value, i.e. its umbound. Now we introduce the environment and its operations.

```
type evT = Int of int | Bool of bool | Unbound

type env = ( string * evT ) list;;

let emptyEnv  = [ ("", Unbound) ] ;;

let bind (s:env) (i:string) (x:evT) = ( i, x ) :: s;;

let rec lookup (s:env) (i:string) = match s with
    | [] ->  Unbound
    | (j,v)::sl when j = i -> v
    | _::sl -> lookup sl i;;
```

We assume three basic operations over environments:

1. Creation of the empty environment (`emptyEnv`). For convenience our implementation uses one sentinel node to track the end of the list. This choice does not change the basic behavior of the environment data structure.
2. Creation of a binding in an environment (`bind`).
3. Accessing an element in the environment by its name (ćodelookup).

Now we can tackle the interpreter. Constants and basic operations are easily handled. Recall that in the substitution-based implementation, the interpreter recurred without performing any new substitutions. As a result, there are no new deferred substitutions to perform either, which means the environment does not change.

```
let rec eval  (e:exp) (s:env) =
 match e with
 | CstInt(n) -> Int(n)
 | CstTrue -> Bool(true)
 | CstFalse -> Bool(false)
 | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
 | Times(e1,e2) -> int_times((eval e1 s), (eval e2 s))
 | Sum(e1, e2) -> int_plus((eval e1 s), (eval e2 s))
 | Ifthenelse(e1,e2,e3) ->
                let g = eval e1 s in
                match (typecheck("bool", g), g) with
                        | (true, Bool(true)) -> eval e2 s
                        | (true, Bool(false)) -> eval e3 s
                        | (_, _) -> failwith ("nonboolean guard")
```

Now let us handle identifiers. Clearly, encountering an identifier is no longer an error: this was the very motivation for this change. Instead, we must look up its value in the environment:

```
| Den(i) -> lookup s i
```

Finally, it remains to handle the declaration with the let expression.

```
| Let(i, e, ebody) -> eval ebody (bind s i (eval e s));;
```

The body of the let is evaluated in the environment obtained by extending the current environment with the binding between the variable i introduced by the let expression and the value of the expression associated to that variable. The new binding of the variable i will hide any existing binding for i thanks to the definition of the lookup function. Also. The old environment s is not destructively modified further evaluation will continue in the old environment.

The interpreter eval takes and expression, an environment and yields as result an expressible type:

`val eval :  exp -> env -> evT = <fun>`

We now exercise our interpreter in the standard REPL style.

```
# let myp = Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
val myp : exp =
  Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y")))
# eval myp emptyEnv;;
- : evT = Int 42
# let myp' = CstInt(3);;
val myp' : exp = CstInt 3
# let e = Eq(CstInt(5),CstInt(5));;
val e : exp = Eq (CstInt 5, CstInt 5)
# let myite = Ifthenelse(e,myp,myp');;
val myite : exp =
  Ifthenelse (Eq (CstInt 5, CstInt 5),
   Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y"))),
   CstInt 3)
# eval myite emptyEnv;;
- : evT = Int 42
```

5 A static analysis primer

An expression is closed if no variable occurs free in the expression. In most programming languages a valid program must be closed: it cannot have unbound (i.e. undeclared) identifiers. We now introduce an example of a static analysis, that is a check on the source code of a program that is performed without actually executing the program itself. Our analysis checks whether an expression is closed. For simplicity we consider the language of arithmetic expressions with let declaration, the environment is modified accordingly. Here is the corresponding algebraic data type.

```
type aexp = CstInt of int
         | Den of ide
         | Sum of aexp * aexp
         | Times of aexp * aexp
         | Let of ide * aexp * aexp ;;
```

We first introduce an auxiliary function.

```
let rec closedin (e:aexp) (lst: string list) =
 match e with
 | CstInt(n) -> true
 | Times(e1,e2) -> closedin e1 lst && closedin e2 lst
 | Sum(e1, e2) -> closedin e1 lst && closedin e2 lst
 | Den(i) -> List.exists (fun y -> i = y) lst
 | Let(i, e, ebody) -> let lst1 = i :: lst in
      closedin e lst1 && closedin ebody lst1;;
```

The function closedin checks whether the expression e is closed in a lsit of boud variables. It type is

`val closedin :  aexp -> ide list -> bool = <fun>`

We now comment on its definition. A constant is always closed. A variable occurrence `Den(i)` is closed in `lst` if it appears in `lst`. A let declaration introduces a bound variable as expected.

An expression is closed if it is closed in the empty list of bound variables.

```
# let closed e = closedin e [];;
val closed : aexp -> bool = <fun>
```

We now exercise our simple static machinery for arithmetic expressions:

```
# let e1 = Sum(Den("x"), CstInt(5));;
val e1 : aexp = Sum (Den "x", CstInt 5)
# closed e1;;
- : bool = false
let e2 = Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
val e2 : aexp =
  Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y")))
# closed e2;;
- : bool = true
```

As expected expression `e1` is not closed: the occurrence of the identifier `x` is undeclared. Instead, the expression `e2` is closed.

We can easily implement the interpreter for arithmetic expressions (do it now!)

`val evalAexp :  aexp -> env -> evT = <fun>`.

Hence, the evaluation tool-chain for arithmetic expressions is given by

```
let execAexp (e:aexp) =
 match closed e with
 | true -> evalAexp e emptyEnv
 | false -> failwith("undefined variables");;

     val execAexp : aexp -> evT = <fun>
```

We now introduce a second static analysis technique which replaces symbolic names (identifiers) with addresses (integers) in real machine code. Most run-time structures adopt this strategy to efficiently manage accesses to data at run-time.

To illustrate this static analysis technique we first introduce an abstract syntax for run-time arithmetic expressions that exploits integer indexes instead of identifiers.

```
type rtaexp =
        | RTCstInt of int
        | RTSum of rtaexp * rtaexp
        | RTTimes of rtaexp * rtaexp
        | RTDen of int
        | RTLet of rtaexp * rtaexp;;
```

The next step is define the function compile

`val compile:  aexp -> string list -> rtaexp`.

This function translates arithmetic expressions into run-time arithmentic expression by replacing identifiers. It also drops the bound name identifier from let-expressions. Note that the compiler takes as input the static environment that consists of the list of bound identifiers. Usually, the static environment is known under the name of symbol table. Also, the position of a symbolic name in the static environment represents the binding depth, i.e. the number of let-bindings between the occurrence of the identifier and the corresponding binding. We now present the code of the compiler. The compiler exppoits the function`getindex`. This function uses the static environment to map an identifier into its integer index: the first occurrence of the identifier in the list.

```
let rec compile (e:aexp) (st: ide list): rtaexp =
        match e with
      | CstInt(n) -> RTCstInt(n)
      | Den(i) -> RTDen(getindex st i)
      | Sum(e1, e2) ->
               RTSum((compile e1 st), (compile e2 st))
      | Times(e1, e2) ->
              RTTimes((compile e1 st), (compile e2 st))
      | Let(i, e, body) ->
              let st1 = i :: st in
               RTLet((compile e st), (compile body st1));;

let rec getindex (st: ide list) x =
    match st with
    | [] -> failwith("Variable not found")
    | y::yr -> if x=y then 0 else 1 + getindex yr x
```

We still have to discuss the code of the interpreter of run-time arithmentic expressions. The interpreter, shown below, takes as input the run-time environment, i.e. a list of integers storing the values of identifiers in the same order they appear in the static environment. Therefore we can simply exploit the binding depth of an identifier to access its value at run-time: identifiers are not active entities at run-time. The integer giving the position in the run-time environment is usually called offset by compiler writers and deBruijn index by lambda-calculus theoreticians.

```
let rec rteval (e : rtaexp) (rtenv : int list)  =
    match e with
      | RTCstInt(n) -> n
      | RTDen(n)  -> List.nth rtenv n
      | RTLet(e, body) ->
            let xval = rteval e rtenv in
               let rtenv1 = xval :: rtenv  in
                  rteval body rtenv1
      | RTSum(e1, e2) ->
            rteval e1 rtenv + rteval e2 rtenv
      | RTTimes(e1, e2) ->
            rteval e1 rtenv * rteval e2 rtenv
```

We now exercise the machineries we developed.

```
# let e = Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
val e : aexp =
  Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y")))
# let o = compile e [];;
val o : rtaexp =
  RTLet (RTCstInt 30, RTLet (RTCstInt 12, RTSum (RTDen 1, RTDen 0)))
# rteval o [];;
- : int = 42
```

We have indeed introduced a two-stage execution chain of the form
`exec(e:aexp) = rteval(compile(e), [])`

Note that in the one-stage interpreter of arithmetic expressions (`evalAexp`) the environment had type `(string * int) list` and contained identifiers and their values. In the two-stage execution, the static environment had type `ide list` and contained identifiers only, whereas the run-time environment had type `int list` and contained values only.

The idea behind the two-stage execution is to split the environment into a stati (compile-time) environment and a dynamic (run-time) environment. This is not accident: the goal of compilation is to performs some computational steps (such as the lookup of identifiers) early at compile-time, and performs other computations (such as multiplications) only later at run-time.

Instruction	Stack Before		Stack Later
$STCstInt(n)$	s	$\longrightarrow$	s, n
$STAdd$	s, n_1, n_2	$\longrightarrow$	$s, (n_1 + n_2)$
$STSub$	s, n_1, n_2	$\longrightarrow$	$s, (n_1 - n_2)$
$STMul$	sn_1, n_2	$\longrightarrow$	$s, (n_1 * n_2)$
$STDup$	s, n	$\longrightarrow$	s, n, n
$STSwap$	s, n_1, n_2	$\longrightarrow$	s, n_2, n_1

Figure 6: Stack Machine Instructions: Operational Semantics

6 Stack machines

Expressions and more generally functional programs, are ofter executed by a stack machine. We shall study a simple stack machine, an interpreter that implements the evaluation strategy, for a language of expressions in postfix polish form.

Stack machine instructions for our example language (without identifiers and let bindings) may be described as follows.

```
type stkinstr =
  | STCstInt of int
  | STAdd
  | STSub
  | STMul
  | STDup
  | STSwap
```

The state of the stack machine is a pair of the form (c, s) consisting of the control and the stack. The control c is the sequence of instructions yet to be evaluated. The stack s is a list of values (in our simple case integer values). These values are the intermediate results of the evaluation.

The stack machine can be defined by the transition rules described in Figure 6. Each transition rule says how the execution of one instructions causes the machine to go from one state to another. The top of the stack is to the right.

The rules of the stack machine quite easily correspond to the the function `stkEval` defined in Figure 7 (top). Our stack machine is exercized in Figure 7 (botton)

7 Functions everywhere

When we introduce functions we were following the model of an idealized programming language, with definitions and their uses kept separate. Let us examine how necessary that is. Why cannnot functions definitions be expressions? In our current kernel programming language we face the question What value does a function definition represent? , to which we do not really have a good answer. But a real programming language obviously computes more than numbers and Booleans, so we no longer need to confront the question in this form; indeed, the answer to the above can just as well be, *a function value*. Let us see how that might work out.

What can we do with functions as values? Clearly, functions are a distinct kind of value than a number, so we cannot, for instance, add them. But there is one evident thing we can do: apply them to arguments! Thus, we can allow function values to appear in the function position of an application. The behavior would, naturally, be to apply the function. We are therefore proposing a language where the following would be a valid program (we adopte a OCaml-like syntax):

```
let rec stkEval (control: stkinstr list) (stack: int list): int =
 match (control, stack) with
   | ([], v::_) -> v
   |( [], []) -> failwith("no result on the stack")
   | (STCstInt(n)::cs, stack)-> stkEval cs (n::stack)
   | (STAdd::cs, n1::n2::ss) -> stkEval cs ((n1+n2)::ss)
   | (STSub::cs, n1::n2::ss) -> stkEval cs ((n1-n2)::ss)
   | (STMul::cs, n1::n2::ss) -> stkEval cs ((n1*n2)::ss)
   | (STDup::cs, n::ss) -> stkEval cs (n::n::ss)
   | (STSwap::cs, n1::n2::ss) -> stkEval cs (n2::n1::ss)
   | (_::_, []) -> failwith("too few operands")

(* The REPL Usage *)

# let c = STCstInt(5)::STCstInt(6)::STAdd::STCstInt(8)::STSub::[];;
val c : stkinstr list = [STCstInt 5; STCstInt 6; STAdd; STCstInt 8; STSub]
# stkEval c [];;
- : int = -3
```

Figure 7: Stack Machine Evaluation

```
type exp = CstInt of int
                | CstTrue
                | CstFalse
                | Den of ide
                | Sum of exp * exp
                | Sub of exp * exp
                | Times of exp * exp
                | Ifthenelse of exp * exp * exp
                | Eq of exp * exp
                | Let of ide * exp * exp
                | Fun of ide * ide * exp
                | Apply of exp * exp
```

Figure 8: Abstract syntax of a kernel functional language

```
# 2 + let f x = x+5 in f(4);;
- : int = 11
```

7.1 Functions as expressions and values

Let us first define the core language to include function definitions. For simplicity we consider functions that can take only one argument. The abstract syntax for our language is shown in Figure 8.

We introduce *anonymous functions*: an anonymous function is a function that is declared without being named. In `Fun("x", body)`

1. `"x"` is the formal paramenter,
2. `fbody` is the function body,

For instance the OCaml program `let f x = x+7 in f 2` would look like this in our abstract syntax

```
Let("f", Fun("x", Sum(Den("x"), CstInt(7))), Apply(Den("f"),CsInt(2)))
```

The abstract syntax of Figure 8 allows arbitrary expressions to be used in the components of function applications (calls), i.e. `Apply(e, arg)`. In this section we restrict our language to be first-order by requiring

`f` in `f(arg)` to be a function name. In our abstract syntax this means that in a function call `Apply(e,srg)` the expression `e` must be a function name. Therefore, for now all function applications must have the form `Apply(Den("f"), arg)` where `f` is a function name as in the example above.

We have simply introduced functions into our expression language. The interpreter now no longer returns just integer or boolean values: it also returns function values. Hence, our definition of expressible values `evT` has to be updated, but we will soon find that we need to think this a little more.

When the interpreter evaluates a function (e.g. `Fun("x", Sum(Den("x"), CstInt(7)))`) what is the value it has to provides as result of the evaluation? Let us consider the following OCaml program

```
let x = 5 in
```

which evaluates to the integer value 6.

The main point is that a function cannot just evaluate to its body because the body may contain free variables. The identifier `x` is free in the definition of the function `f` above. A function value needs to remember the *substitution* that have already been applied to it. Because we are representing substitutions using an environment, a function value therefore needs to be bundled with an environment. This resulting data structure is called a *closure*. The closure created for the function `f` above would be

`<Fun("z", Sum(Den("z"), Den("x"))), [("x", CstInt(5))]>`

So we get the following definition of the `evT` type

```
type 'v env = (string * 'v) list;;
type evT = Int of int | Bool of bool | Closure of ide * exp * evT env) | Unbond;;
```

The interpreter now uses environment and closures. Most cases are unchanged from before:

```
let rec eval  (e:exp) (s:evT env) = match e with
 | CstInt(n) -> Int(n)
 | CstTrue -> Bool(true)
 | CstFalse -> Bool(false)
 | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
 | Times(e1,e2) -> int_times((eval e1 s), (eval e2 s))
 | Sum(e1, e2) -> int_plus((eval e1 s), (eval e2 s))
 | Sub(e1, e3) -> let int_sub((eval e1 s), (eval e2 s))
 | Ifthenelse(e1,e2,e3) -> let g = eval e1 s in
                (match (typecheck("bool", g), g) with
                        | (true, Bool(true)) -> eval e2 s
                        | (true, Bool(false)) -> eval e3 s
                        | (_, _) -> failwith ("nonboolean guard"))
| Den(i) -> lookup s i
| Let(i, e, ebody) -> eval ebody (bind s i (eval e s))
```

Now we have to consider the new cases namely functional expressions and function application

```
| Fun(arg, ebody) -> Closure(arg,ebody,s)
| Apply(Den(f), eArg) ->
  let fclosure = lookup s f in
     (match fclosure with
       | Closure(arg, fbody, fDecEnv) ->
         let aVal = eval eArg s in
         let aenv = bind fDecEnv arg aVal in
         eval fbody aenv
       | _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function") ;;
```

A functional value `Fun(arg, ebody)` is evaluated by creating a function closure `Closure(arg,ebody,s)` bearing the binding of the current environment where the functional value is evaluated. A function application `Apply(Den(f), eArg)` is evaluated by first checking that the function name `f` is bound in the environment to a function closure. The actual parameter is evaluated in the calling environment to obtain a value `aVal` and the function body is evaluated in the declaration environment extended with the binding of the formal parameter.

As usual we exercise our interpreter in the REPL style.

```
# let e = Let ("x", CstInt 5,
   Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)));;
val e1 : exp =
  Let ("x", CstInt 5,
   Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)))
# eval e1 emptyEnv ;;
- : evT = Int 6
```

7.2 Recursive functions

A function is recursive if it refers to itself in its definition. Recursion is important in any programming language, but is particularly important in pure functional languages because recursion is the only way to iterate.

We now experiment our current interpreter over a simple program exploiting a recursive definition.

```
# let r = Let("fact", Fun("n",
 Ifthenelse(Eq(Den("n"),CstInt(0)),
                 CstInt(1),
                 Times(Den("n"),Apply(Den("fact"),Sub(Den("n"),CstInt(1)))))),
                 Apply(Den("fact"),CstInt(3))) ;;

val r : exp =
  Let ("fact",
   Fun ("n",
    Ifthenelse (Eq (Den "n", CstInt 0), CstInt 1,
     Times (Den "n", Apply (Den "fact", Sub (Den "n", CstInt 1))))),
   Apply (Den "fact", CstInt 3))
```

The program `r` above has type `exp` as expected. However, by evaluating the program above the interpreter throws a runt-time exception:

```
# eval r emptyEnv;;
Exception: Failure "non functional value".
```

To explain this result, it is instructive to trace the execution of the interpreter. For simplicity, hereafter we adopt the following shorthands:

```
funF= Fun ("n",
    Ifthenelse (Eq (Den "n", CstInt 0), CstInt 1,
     Times (Den "n", Apply (Den "fact", Sub (Den "n", CstInt 1)))))

bodyF = Ifthenelse (Eq (Den "n", CstInt 0), CstInt 1,
             Times (Den "n", Apply (Den "fact", Sub (Den "n", CstInt 1))))

env1 =  bind emptyEnv "fact" (eval funFact emptyEnv)} =
bind emptyEnv "fact" Closure("n", bodyFact, emptyEnv) =
("fact", Closure("n", bodyFact, emptyEnv))::emptyEnv
```

We have

eval r emptyEnv

$\Downarrow$

eval Apply (Den "fact", CstInt 3)) env1

$\Downarrow$

eval Apply (Den "fact", CstInt 3)) bind emptyEnv "fact" Closure("n", bodyFact, emptyEnv)

$\Downarrow$

bodyFact bind emptyEnv "n" CstInt(3)

$=$

bodyFact ("n" CstInt(3)::emptyEnv

$\Downarrow$

eval Times (Den "n", Apply (Den "fact", Sub (Den "n", CstInt 1))) ("n" CstInt(3))::emptyEnv

Now, the rule of the interpreter for the evaluation of the `Times` operation will result in the invocation of the interpreter on the recursive call of the factorial function:

`eval Apply (Den "fact", Sub (Den "n", CstInt 1))) ("n" CstInt(3))::emptyEnv`.

It is easy to see that the environment `("n" CstInt(3))::emptyEnv` does not contains any binding for the identifier `"fact"`. Hence, the lookup operation `lookup ("n" CstInt(3))::emptyEnv "fact"` will yield as result the value `Unbound` and thus determining the execution of the pattern `| _ -> failwith("non functional value")`.

This means that recursive functions requires a special treatment. Here, in order to manage recursive functions, we adopt the OCaml design choice making a distinction between non recursive functions and recursive functions. In particular, recursive functions are defined via the **let rec** binding. This design decision has some good effects. As we discussed above, recursive functions are harder to implement than nonrecursive ones. It is also useful that, in the absence of an explicit rec, one can safely assume that a let binding is nonrecursive, and so can only build upon previous bindings: having a nonrecursive let binding makes it easier to create a new definition that extends and supersedes an existing one by simply shadowing it.

We now extend the abstract syntax for our language of Figure 8 by adding the constructor for recursive definitions:

```
type exp =
  :
  | Letrec of ide * ide * exp * exp
```

The intuition of `Letrec("f", "x", fbody, letbody)` is that

- `"f"` is the function name,
- `"x"` is the formal parameter,
- `fbody` is the function body,
- `letbody` is the body of the let.

For instance the application of the factorial function to the integer value 3 will look like in the abstract syntax as follows:

```
Letrec("fact", "n",
Ifthenelse(Eq(Den("n"),CstInt(0),
                CstInt(1)),
                Times(Den("n"),Apply(Den("fact"),Sun(Den("n"),CstInt(1))))),
Apply(Den("fact"),CstInt(3)))
```

We have to extend the syntax of expressible type (`evT`) in order to allow a function to apply itself recursively.

```
type evT = .. | RecClosure of ide * ide * exp * evT env;;
```

The name of the function is included in the closure to allow recursion. Finally, we need to extend the interpreter to handle the application of recursive functions. Consider for a moment a recursive approach to the summation of the numbers between 1 and 5. Using an OCaml syntax we write

```
let rec f x = If x = 1 then 1 else f (x - 1) + x in f 5
```

If we focus on the body of the let rec, we can see that we want the body of the recursive function `f` to be applied to the value 5. In addition, we need to ensure that any occurrences of `f` in the body of the function itself are bound in the environment. To what do we bind them? Fortunately, we have `RecClosure` as values, thus, we can bind the name `f` with the corresponding closure. This leads to the following definition of the interpreter:

```
| Apply(Den(f), eArg) ->
  let fclosure = lookup s f in
     (match fclosure with
       | Closure(arg, fbody, fDecEnv) ->
         let aVal = eval eArg s in
         let aenv = bind fDecEnv arg aVal in
         eval fbody aenv
       | RecClosure(f, arg, fbody, fDecEnv) ->
         let aVal = eval eArg s in
         let rEnv = bind fDecEnv f fclosure in
         let aenv = bind rEnv arg aVal in
         eval fbody aenv
       | _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function") ;;
```

In the case of recursive function, the interpreter behaves as follows:

1. The `RecClosure` is retrieved from the environment,
2. The actual paramenter is evaluated in the calling environment, getting the value `aVal`.
3. The function declaration environment `fDecEnv`, stored inside the `RecClosure`, is extended with the binding between the name of the function `f` and its recursive closure, obtaining the `rEnv` environment,
4. The actual execution environment `aenv` is computed by extending the `rEnv` environment with the binding between the formal parameter and the value `aVal` of the actual parameter.

As usual we exercize our interpreter

```
# let myRP =
Letrec("fact", "n",
Ifthenelse(Eq(Den("n"),CstInt(0)),
                 CstInt(1),
                 Times(Den("n"),Apply(Den("fact"),Sub(Den("n"),CstInt(1))))),
Apply(Den("fact"),CstInt(3)));;
val myRP : exp =
  Letrec ("fact", "n",
   Ifthenelse (Eq (Den "n", CstInt 0), CstInt 1,
    Times (Den "n", Apply (Den "fact", Sub (Den "n", CstInt 1)))),
   Apply (Den "fact", CstInt 3))
# eval myRP emptyEnv;;
- : evT = Int 6
```

8 Static Scope and Dynamic Scope

The simple programming language implemented in the previous section adopts an *static scope* discipline. Static scope means that each occurrence of an identifier refers to the (statically) innermost enclosing binding of that identifier. Thus one can decide, given just the program text, which binding a given identifier occurrence refers to.

In a language with static (or lexical) scope, the active bindings, at any point of the program and at any point during execution, depend uniquely on the syntactic structure of the program itself. Such an active environment can therefore be determined completely by the compiler, hence the term static . For example, let us consider the following OCaml program:

```
let y = 11 in
  let f x = x + y in
     let y = 22 in
        f 3
```

With static scope, the occurrence of `y` inside the body of function `f` refers to the binding `x = 11`. Hence, this program evaluates to `3 + 11 = 14`.

Static scope allows the determination of all the environments present in a program simply by reading its text. This has two important consequences of a positive nature. First, programmers have a better understanding of the program, as far as they can connect every occurrence of an identifier to its correct declaration by observing the textual structure of the program and without having to simulate its execution. Moreover, this connection can also be made by the compiler which can therefore determine each and every use of an identifier. This makes it possible, at compile time, to perform a great number of correctness tests using the information contained in types; it can also perform considerable number of code optimisations. For example, if the compiler knows (using declarations) that the identifier `x` which occurs in an expression is an integer variable, it will signal an error in the case in which a character is assigned to this variable. Similarly, if the compiler knows that the constant `foo` is associated with the value 10, it can substitute the value 10 for every reference to `foo`, so avoiding having to arrange for this operation to be performed at runtime, therefore, it updates the code. If, instead, the correct declaration for `x` and for `foo` can be determined only at execution time, it is clear that these checks and this optimisation are not possible as compilation time.

An alternative scope discipline is *dynamic* scope. With a dynamic scope, an occurrence of an identifier refers to the (dynamically) most recent binding of that identifier. In the above example. when function `f` is called the occurrence of `y` inside the body of `f` would refer to the second let-binding of `y`, which enclosed the call to `f`. Under the dynamic scope discipline this program would evaluated to `3+22 = 25`.

It is easy to modify the interpreter of previous section to implement dynamic scope. In a function call the body of the function should be simply evaluated in the environment in force when the function is called. Note that the *function declaration environment* is not used any longer and could be left out from the closure.The basic structure of the interpreter with dynamic scope disciple is outlined below.

```
| Fun(arg, ebody) -> Closure(arg,ebody)
| Apply(Den(f), eArg) ->
  let fval = lookup s f in
     (match favl with
       | Closure(arg, fbody) ->
         let aVal = eval eArg s in
         let aenv = bind s arg aVal in
         eval fbody aenv
       | _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function") ;;
```

This simple example indeed shows that dynamic scope is easier to implement that static scope. In fact static scope imposes a fairly complicated runtime regime because the various non-local environments, i.e. the

environment encapsulated inside a closure, are involved in a way that does not reflect the normal execution flow. This might be the reason that the original version of the Lisp programming language (1960) as well as most scripting languages, adopt dynamic scope. However, dynamic scope makes type checking and program optimisations difficult, and allows for very obscure program mistakes. Hence, almost all modern programming languages use static scope. The perl language has both statically and dynamically scope identifiers, declared using the keywords `my` and `local` respectively.

8.1 Higher-order Functions

A remarkable feature of functional programming languages is the ability yo treat functions as first-class values, just like integers and strings. This means that well known programming patterns such us uniform transformation of the elements of a list, can be naturally implemented as a *higher-order* function that takes as argument (or returns) another function. We now show how to extend our first-order functional language to a higher-order one.

The abstract syntax of Figure 8 already allows the function part of an application to be an arbitrary expression, `Apply of exp * exp`. To accomodate higher-order features in our language one has to admit the possibility that an expression evaluates to a function. Note that our language allows anonymous functions and that an identifier may be bound to a function value, i.e. a closure. Hence, the only diffrenece beween the higher-order interprete and the first-order interpreter is the handling of function application. A functional application `apply(eF, eArg)` is evaluated by evaluating the expression `eF` to a functional value and then evaluating the body of the function (extracted from the closure) in the static environment. Here is the corresponding fragment of the interpreter.

```
| Apply(eF, eArg) ->
  let fclosure = eval eF in
     (match fclosure with
       | Closure(arg, fbody, fDecEnv) ->
         let aVal = eval eArg s in
         let aenv = bind fDecEnv arg aVal in
         eval fbody aenv
       | RecClosure(f, arg, fbody, fDecEnv) ->
         let aVal = eval eArg s in
         let rEnv = bind fDecEnv f fclosure in
         let aenv = bind rEnv arg aVal in
         eval fbody aenv
       | _ -> failwith("non functional value")) ;;
```

8.2 eager and Lazy Evaluation

In a functional application such as `f(e)` one can evaluate the actual parameter `e` eagarly to obtain a value `v` before evaluating the function body. This is what we are used in OCaml, Java, C, C#. Alternatively, one might evaluate the actual parameter `e` lazily. This means that the evaluation of `e` is postponed until the actual value of `e` is needed. If it is not needed we never evaluate `e`. The distinction between eager and lazy evaluation makes a big difference in functiuonal programming. For instance, let us consider the following OCaml program

```
# let rec forever n = forever n ;;
val forever : 'a -> 'b = <fun>
# let f x = 1 ;;
val f : 'a -> int = <fun>
# f (forever (2));;
```

The evaluation of `f(forver(2))` will never terminate if we use eager evaluation as in OCaml. With lazy evaluation, however, we do not evaluate the expression `forever(2))` until we have found that function `f` needs

its value to progress. Indeed, function `f` does not need it at all because the formal parameter `x` does not appear in the body of function `f`. Then, with lazy evaluation the program above terminates yielding as result the integer value `1`.

The previous example is somewhat too artificial. To make things more concrete, let us consider the evaluation of the following OCaml program.

```
# let myif b v1 v2 = if b then v1 else v2;;
val myif : bool -> 'a -> 'a -> 'a = <fun>
# let rec fact n = myif (n=0) 1 (n * fact(n-1));;
val fact : int -> int = <fun>
# fact 3;;
Stack overflow during evaluation (looping recursion?).
```

The eager evaluation of the third argument of function `myif` will force the program to enter into an infinite loop. Thus it is an important that the built-in conditional `if-then-else` is not eager.

Our toy functional programming language is eager. The interpreter (the function eval) always evaluates the actual parameter of a function before evaluating the function body. Note also that our conditional is not eager.

Most widely used programming languages (C, C++, Java, C#, Pascal, Ada, Lisp, ...) use eager evaluation. A well known exception is Algol 60, whose call-by-name parameter passing mechanism evaluates an argument only when needed (and re-evaluates it every time it is needed).