

## Chapter 2

# Operational Semantics

### 2.1 A First Look at Operational Semantics

The **syntax** of a programming language is the set of rules governing the formation of expressions in the language. The **semantics** of a programming language is the *meaning* of those expressions.

There are several forms of language semantics. Axiomatic semantics is a set of axiomatic truths in a programming language. Denotational semantics involves modeling programs as static mathematical objects, namely as set-theoretic functions with specific properties. We, however, will focus on a form of semantics called operational semantics.

An operational semantics is a mathematical model of programming language *execution*. It is, in essence, an interpreter defined mathematically. However, an operational semantics is more precise than an interpreter because it is defined mathematically, and not based on the meaning of the programming language in which the interpreter is written. This might sound like a pedantic distinction, but interpreters interpret e.g. a language's **if** statements with the **if** statement of the language the interpreter is written in. This is in some sense a circular definition of **if**. Formally, we can define operational semantics as follows.

**Definition 2.1** (Operational Semantics). *An **operational semantics** for a programming language is a mathematical definition of its computation relation,  $e \Rightarrow v$ , where  $e$  is a program in the language.*

$e \Rightarrow v$  is mathematically a 2-place relation between expressions of the language,  $e$ , and values of the language,  $v$ . Integers and booleans are values. Functions are also values because they don't compute to anything.  $e$  and  $v$  are **metavariables**, meaning they denote an arbitrary expression or value, and should not be confused with the (regular) variables that are part of programs.

An operational semantics for a programming language is a means for understanding in precise detail the meaning of an expression in the language. It is the formal specification of the language that is used when writing compilers and interpreters, and it allows us to rigorously verify things about the language.

## 2.2 BNF grammars and Syntax

Before getting into meaning we need to take a step back and first precisely define language syntax. This is done with formal grammars. *Backus-Naur Form* (BNF) is a standard grammar formalism for defining language syntax. You could well be familiar with BNF since it is often taught in introductory courses, but if not we provide a brief overview. All BNF grammars comprise *terminals*, *nonterminals* (aka *syntactic categories*), and production rules. Terminals are traditionally identified using lower-case letters; non-terminals are identified using upper-case letters. Production rules describe how non-terminals are defined. The general form of production rules is:

$$\langle \text{nonterminal} \rangle ::= \langle \text{form 1} \rangle \mid \cdots \mid \langle \text{form n} \rangle$$

where each “form” above describes a particular language form – that is, a string of terminals and non-terminals. A *term* in the language is a string of terminals which matches the description of one of these rules (traditionally the first).

For example, consider the language Sheep. Let  $\{S\}$  be the set of nonterminals,  $\{a, b\}$  be the set of terminals, and the grammar definition be:

$$S ::= b \mid Sa$$

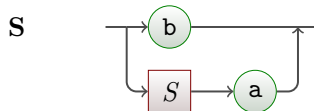
Note that this is a recursive definition. Examples of terms in Sheep are

$$b, ba, baa, baaa, baaaa, \dots$$

That is, any string starting with the character  $b$  and followed by zero or more  $a$  characters is a term in Sheep. The following are examples that are not terms in SHEEP:

- $a$ : Terms in Sheep must start with a  $b$ .
- $bbaaa$ : Sheep does not allow multiple  $b$  characters in a term.
- $baah$ :  $h$  is not a terminal in Sheep.
- $Saaa$ :  $S$  is a non-terminal in Sheep. Terms may not contain non-terminals.

Another way of expressing a grammar is by the use of a **syntax diagram**. Syntax diagrams describe the grammar visually rather than in a textual form. For example, the following is a syntax diagram for the language Sheep:



The above syntax diagram describes all terms of the Sheep language. To generate a form of  $S$ , one starts at the left side of the diagram and moves until one reaches the right. The rectangular nodes represent non-terminals while the rounded nodes represent terminals. Upon reaching a non-terminal node, one must construct a term using that non-terminal to proceed.

As another example, consider the language Frog. Let  $\{F, G\}$  be the set of nonterminals,  $\{r, i, b, t\}$  be the set of terminals, and the grammar definition be:

$$\begin{aligned} F &::= rF \mid iG \\ G &::= bG \mid bF \mid t \end{aligned}$$

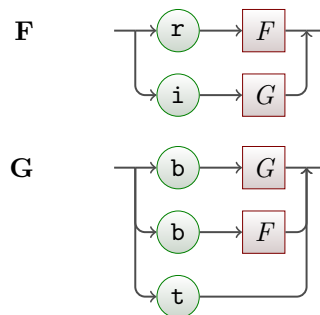
Note that this is a mutually recursive definition. Note also that each production rule defines a syntactic category. Terms in FROG include:

*ibit, ribbit, ribibbbbit . . .*

The following terms are not terms in Frog:

- *rbt*: When a term in Frog starts with  $r$ , the following non-terminal is  $F$ . The non-terminal  $F$  may only be expanded into  $rF$  or  $iG$ , neither of which start with  $b$ . Thus, no string starting with  $rb$  is a term in Frog.
- *rabbit*:  $a$  is not a terminal in Frog.
- *rrrrrrF*:  $F$  is a non-terminal in Frog; terms may not contain non-terminals.
- *bit*: The only forms starting with  $b$  appear as part of the definition of  $G$ . As  $F$  is the first non-terminal defined, terms in Frog must match  $F$  (which does not have any forms starting with  $b$ ).

The following syntax diagram describes Frog:



### 2.2.1 Operational Semantics for Logic Expressions

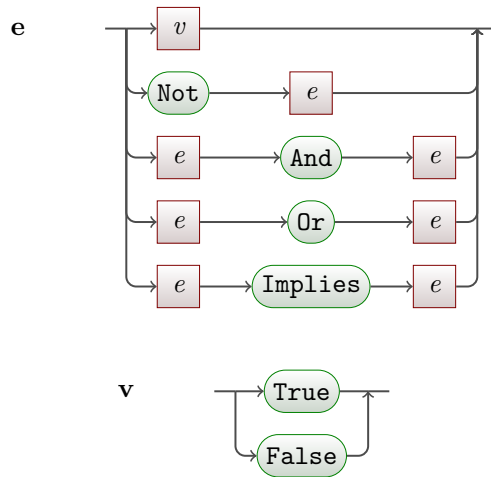
In order to get a feel for what an operational semantics is and how it is defined, we will now examine the operational semantics for a very simple language: propositional boolean logic with no variables. The syntax of this language is as follows. An expression  $e$  is recursively defined to consist of the values **True** and **False**, and the expressions  $e$  **And**  $e$ ,  $e$  **Or**  $e$ ,  $e$  **Implies**  $e$ , and **Not**  $e$ .<sup>1</sup> This syntax is known as the **concrete syntax**,

<sup>1</sup>Throughout the book we use syntax very similar to Caml in our toy languages, but with the convention of capitalizing keywords to avoid potential conflicts with the Caml language.

because it is the syntax that describes the textual representation of an expression in the language. We can express it in a BNF grammar as follows:

$$\begin{aligned} e &::= v \mid \text{Not } e \mid (e \text{ And } e) \mid (e \text{ Or } e) \mid (e \text{ Implies } e) && \text{expressions} \\ v &::= \text{True} \mid \text{False} && \text{values} \end{aligned}$$

The following is an equivalent syntax diagram:



Note that the syntax above breaks tradition somewhat by using lower-case letters for non-terminals. Terminals are printed in fixed-width font. The rationale for this is consistency with the metavariables we will be using in operational semantics below and will become clear shortly.

We can now discuss the operational semantics of the boolean language. Operational semantics are written in the form of logic rules, which are written as a series of pre-conditions above a horizontal line and the conclusion below it. For example, the logic rule

$$\text{(Apple Rule)} \quad \frac{\text{Red}(x) \quad \text{Shiny}(x)}{\text{Apple}(x)}$$

indicates that if a thing is red and shiny, then that thing is an apple. This is, of course, not true; many red, shiny things exist which are not apples. Nonetheless, it is a valid logical statement. In our work, we will be defining logical rules pertaining to a programming language; as a result, we have control over the space in which the rules are constructed. We need not necessarily concern ourselves with intuitive sense so long as the programming language has a mathematical foundation.

Operational semantics rules discuss how pieces of code evaluate. For example, let us consider the **And** rule. We may define the following rule for **And**:

$$\text{(And Rule (Try 1))} \quad \frac{}{\text{True And False} \Rightarrow \text{False}}$$

This rule indicates that the boolean language code `True And False` evaluates to `False`. The absence of any preconditions above the line means that no conditions must be met; this operational semantics rule is always true. Rules with nothing above the line are termed **axioms** since they have no preconditions and so the conclusion always holds.

As a rule, though, it isn't very useful. It only evaluates a very specific program. This rule does not describe how to evaluate the program `True And True`, for instance. In order to generalize our rules to describe a full language and not just specific terms within the language, we must make use of metavariables.

To maintain consistency with the above BNF grammar, we use metavariables starting with  $e$  to represent expressions and metavariables starting with  $v$  to represent values. We are now ready to make an attempt at describing every aspect of the `And` operator using the following new rule:

$$(And\ Rule\ (Try\ 2)) \quad \frac{}{v_1\ And\ v_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

Using this rule, we can successfully evaluate `True And False`, `True and True`, and so on. Note that we have used a textual description to indicate the value of the expression  $v_1\ And\ v_2$ ; this is permitted, although most rules in more complex languages will not use such descriptions.

We very quickly encounter limitations in our approach, however. Consider the program `True And (False And True)`. If we tried to apply the above rule to that program, we would have  $v_1 = \text{True}$  and  $v_2 = (\text{False And True})$ . These two values cannot be applied to logical and as `(False and True)` is not a boolean value; it is an expression. Our boolean language rule does not allow for cases in which the operands to `And` are expressions. We therefore make another attempt at the rule:

$$(And\ Rule\ (Try\ 3)) \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1\ And\ e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

This rule is almost precisely what we want; in fact, the rule itself is complete. Intuitively, this rule says that  $e_1\ And\ e_2$  evaluates to the logical and of the values represented by  $e_1$  and  $e_2$ . But consider again the program `True And False`, which we expect to evaluate to `False`. We can see that  $e_1 = \text{True}$  and that  $e_2 = \text{False}$ , but our evaluation relation does not relate  $v_1$  or  $v_2$  to any value. This is because, strictly speaking, we do not know that `True`  $\Rightarrow$  `True`.

Of course, we would like that to be the case and, since we are in the process of defining the language, we can make it so. We simply need to declare it in an operational semantics rule.

$$(Value\ Rule) \quad \frac{}{v \Rightarrow v}$$

The value rule above is an axiom declaring that any value always evaluates to itself. This satisfies our requirement and allows us to make use of the **And** rule. Using this formal logic approach, we can now prove that **True And (False And True)  $\Rightarrow$  False** as follows:

$$\frac{\frac{\text{True} \Rightarrow \text{True}}{\text{True And (False And True)} \Rightarrow \text{False}} \quad \frac{\frac{\text{False} \Rightarrow \text{False} \quad \text{True} \Rightarrow \text{True}}{\text{False And True} \Rightarrow \text{False}}}{\text{True And (False And True)} \Rightarrow \text{False}}$$

One may read the above **proof tree** as an explanation as to why **True And (False And True)** evaluates to **False**. We can choose to read that proof as follows: “*True And (False And True) evaluates to False by the And rule because we know True evaluates to True, that False And True evaluates to False, and that the logical and of true and false is false. We know that False And True evaluates to False by the And rule because True evaluates to True, False evaluates to False, and the logical and of true and false is false.*”

An equivalent and similarly informal format for the above is:

**True And (False And True)  $\Rightarrow$  False**, because by the **And** rule  
**True  $\Rightarrow$  True**, and  
**(False And True)  $\Rightarrow$  False**, the latter because  
**True  $\Rightarrow$  True**, and  
**False  $\Rightarrow$  False**

The important thing to note about all three of these representations is that they are describing a proof tree. The proof tree consists of nodes which represent the application of logical rules with preconditions as their children. To complete our boolean language, we define the  $\Rightarrow$  relation using a complete set of operational semantics rules:

(Value Rule)  $\frac{}{v \Rightarrow v}$

(Not Rule)  $\frac{e \Rightarrow v}{\text{Not } e \Rightarrow \text{the negation of } v}$

(And Rule)  $\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \text{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$

The rules for **Or** and **Implies** are left as an exercise to the reader (see Exercise 2.4).

These rules form a **proof system** as is found in mathematical logic. Logical rules express incontrovertible logical truths. A **proof** of  $e \Rightarrow v$  amounts to constructing a sequence of rule applications such that, for any given application of a rule, the items above the line appeared earlier in the sequence and such that the final rule application is  $e \Rightarrow v$ . A proof is structurally a tree, where each node is a rule, and the subtree rules have conclusions which exactly match what the parent’s assumptions are. For a proof

tree of  $e \Rightarrow v$ , the root rule has as its conclusion  $e \Rightarrow v$ . Note that *all leaves of a proof tree must be axioms*. A tree with a non-axiom leaf is not a proof.

Notice how the above proof tree is expressing how this logic expression could be computed. Proofs of  $e \Rightarrow v$  corresponds closely to how the execution of  $e$  produces the value  $v$  as result. The only difference is that “execution” starts with  $e$  and produces the  $v$ , whereas a proof tree describes a relation between  $e$  and  $v$ , not a function from  $e$  to  $v$ .

**Lemma 2.1.** *The boolean language is **deterministic**: if  $e \Rightarrow v$  and  $e \Rightarrow v'$ , then  $v = v'$ .*

*Proof.* By induction on the height of the proof tree. □

**Lemma 2.2.** *The boolean language is **normalizing**: For all boolean expressions  $e$ , there is some value  $v$  where  $e \Rightarrow v$ .*

*Proof.* By induction on the size of  $e$ . □

When a proof  $e \Rightarrow v$  can be constructed for some program  $e$ , we say that  $e$  **converges**. When no such proof exists,  $e$  **diverges**. Because the boolean language is normalizing, all programs in that language are said to converge. Some languages (such as Caml) are not normalizing; there are syntactically legal programs for which no evaluation proof exists. An example of a Caml program which is divergent is `let rec f x = f x in f 0;;`.

### 2.2.2 Abstract Syntax

Our operational semantics rules have expressed the evaluation relation in terms of concrete syntax using metavariables. Operators, such as the infix operator `And`, have appeared in textual format. This is a good representation for humans to read because it appeals to our intuition; it is not, however, an ideal computational representation. We read `True And False` as “perform a logical and with operands `True` and `False`”. We read `True And (False And True)` as “perform a logical and with operands `False` and `True` and then perform a logical and with operands `True` and the result of the last operation.” If we are to write programs (such as interpreters) to work with our language, we need a representation which more accurately describes how we think about the program.

The **abstract syntax** of a language is such a representation. A term in an abstract syntax is represented as a **syntax tree** in which each operation to be performed is a node and each operand to that operation is a child of that node. In order to represent abstract syntax trees for the boolean language, we might use the following Caml data type:

```
type boolexp =
  True | False |
  Not of boolexp |
  And of boolexp * boolexp |
  Or of boolexp * boolexp |
  Implies of boolexp * boolexp;;
```

To understand how the abstract and concrete syntax relate, consider the following examples:

**Example 2.1.****Concrete:**

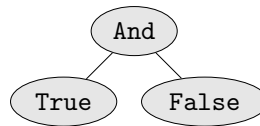
True

**Abstract:**

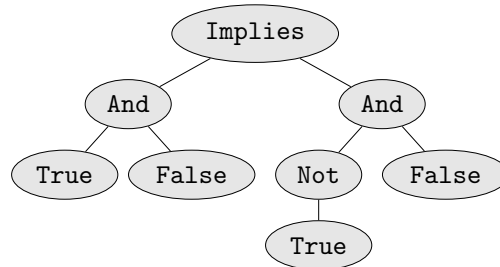
True

**Example 2.2.****Concrete:**

True And False

**Abstract:**

And(True, False)

**Example 2.3.****Concrete:**(True And False) Implies  
((Not True) And False)**Abstract:**Implies( And(True,False) ,  
And(Not(True),False) )

There is a simple and direct relationship between the concrete syntax of a language and the abstract syntax. As mentioned above, the abstract syntax is a form which more directly represents the operations being performed whereas the concrete syntax is the form in which the operations are actually expressed. Part of the process of compiling or interpreting a program is to translate the concrete syntax term (source file) into an abstract syntax term (AST) in order to manipulate it. We define a relation  $\llbracket c \rrbracket = a$  to map concrete syntax form  $c$  to abstract syntax form  $a$  (in this case for the boolean language):

$$\begin{aligned}
 \llbracket \text{True} \rrbracket &= \text{True} \\
 \llbracket \text{False} \rrbracket &= \text{False} \\
 \llbracket \text{Not } e \rrbracket &= \text{Not}(e) \\
 \llbracket e_1 \text{ And } e_2 \rrbracket &= \text{And}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
 \llbracket e_1 \text{ Or } e_2 \rrbracket &= \text{Or}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
 \llbracket e_1 \text{ Implies } e_2 \rrbracket &= \text{Implies}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)
 \end{aligned}$$

For example, this relation indicates the following:



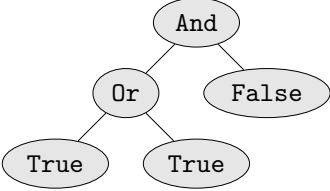
$$\begin{aligned}
 & \llbracket (\text{True And False}) \text{ Implies } ((\text{Not True}) \text{ And False}) \rrbracket \\
 &= \text{Implies}(\llbracket \text{True And False} \rrbracket, \llbracket (\text{Not True}) \text{ And False} \rrbracket) \\
 &= \text{Implies}(\text{And}(\llbracket \text{True} \rrbracket, \llbracket \text{False} \rrbracket), \text{And}(\llbracket \text{Not True} \rrbracket, \llbracket \text{False} \rrbracket)) \\
 &= \text{Implies}(\text{And}(\text{True}, \text{False}), \text{And}(\text{Not}(\llbracket \text{True} \rrbracket), \text{False})) \\
 &= \text{Implies}(\text{And}(\text{True}, \text{False}), \text{And}(\text{Not}(\text{True}), \text{False}))
 \end{aligned}$$

A particularly astute reader will have noticed that the parentheses in the expressions handled until now were not explicitly addressed. This is because parentheses and other parsing meta-operators are not traditionally mentioned in the operational semantics of a language. Such operators merely have the effect of grouping operations. For example, let us assume that binary operations in our boolean language are left-associative; thus, the expressions `True Or True And False` and `(True Or True) And False` are equivalent. Consider the following examples:

**Example 2.4.**

**Concrete:**  
`True Or True And False`

**Abstract:**  
`And(Or(True, True), False)`

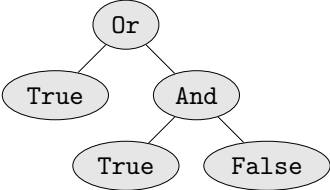


$$\frac{\frac{\text{True} \Rightarrow \text{True} \quad \text{True} \Rightarrow \text{True}}{\text{True Or True} \Rightarrow \text{True}} \quad \frac{\text{False} \Rightarrow \text{False}}{\text{True Or True And False} \Rightarrow \text{False}}$$

**Example 2.5.**

**Concrete:**  
`True Or (True And False)`

**Abstract:**  
`Or(True, And(True, False))`



$$\frac{\frac{\text{True} \Rightarrow \text{True} \quad \frac{\text{True} \Rightarrow \text{True} \quad \text{False} \Rightarrow \text{False}}{\text{True And False} \Rightarrow \text{False}}}{\text{True Or (True And False)} \Rightarrow \text{True}}$$

The expression in example 2.4 will evaluate to `False` because one must evaluate the `Or` operation first and then evaluate the `And` operation using the result. Example 2.5, on the other hand, performs the operations in the opposite order. Note that in both examples, though, the parentheses themselves are no longer overtly present in the abstract syntax.

This is because they are implicitly represented in the structure of the AST; that is, the AST in example 2.5 would not have the shape that it has if the parentheses were not present in the concrete syntax of the form.

In short, parentheses merely change how expressions are grouped. In example 2.5, the only rule we can match to the *entire expression* is the **Or** rule; the **And** rule obviously can't match because the left parentheses would be part of  $e_1$  while the right parenthesis would be part of  $e_2$  (and expressions with unmatched parentheses make no sense). Similarly but less obviously, example 2.4 can only match the **And** rule; the associativity implicitly forces the **Or** rule to happen first, giving the **And** operator that entire expression to evaluate. This distinction is clearly and correspondingly represented in the ASTs of the examples, a fact which is key to the applicability of operational semantics.

### 2.2.3 Operational Semantics and Interpreters

As alluded above, there is a very close relationship between an operational semantics and an actual interpreter written in Caml. Given an operational semantics defined via the relation  $\Rightarrow$ , there is a corresponding (Caml) evaluator function `eval`.

**Definition 2.2** (Faithful Implementation). *A (Caml) interpreter function `eval` faithfully implements an operational semantics  $e \Rightarrow v$  if:*  
 $e \Rightarrow v$  if and only if `eval`( $\llbracket e \rrbracket$ ) returns result  $\llbracket v \rrbracket$ .

To demonstrate this relationship, we will demonstrate the creation of an `eval` function in Caml. Our first draft of the function will, for sake of simplicity, only consist of the **And** rule and the value rule:

```
let eval exp =
  match exp with
  | True -> True
  | False -> False
  | And(exp0,exp1) ->
    begin
      match (exp0, exp1) with
      | (True, True) -> True
      | (_, False) -> False
      | (False, _) -> False
    end
```

At first glance, this function appears to have the behavior we desire. **True** evaluates to **True**, **False** to **False**, and **True And False** to **False**. This is not, however, a complete implementation.

To find out why, consider the concrete term **True And True And True**. As we have seen before, this translates to the abstract term `And(And(True, True), True)`. When the `eval` function receives that value as its parameter, it matches the value to the case **And** and defines `exp0` and `exp1` as `And(True, True)` and `True`, respectively. We then enter the inner match, and this match fails! None of the terms can match the tuple `(exp0, exp1)` because `exp0` is an entire expression and not just a value, as the match expression is expecting.

Looking back at our attempts to write the `And` rule above, we can see why this `eval` function is flawed: this version of the function does not consider the operands of `And` to be expressions - it expects them to be values. We can see, then, that the `And` clause in our function is a faithful implementation of *Try 2* of our `And` rule, a rule which we rejected precisely because it could not handle nested expressions.

How can we correct this problem? We are trying to write a faithful implementation of our final `And` rule, which relies on the evaluation of the `And` rule's operands. Thus, in our implementation, we must evaluate those operands; we make this possible by declaring our evaluation function to be recursive.

```
let rec eval exp =
  match exp with
  | True -> True
  | False -> False
  | And(exp0, exp1) ->
    begin
      match (eval exp0, eval exp1) with
      | (True, True) -> True
      | (_, False) -> False
      | (False, _) -> False
    end
end
```

Observe that, in the above code, we have changed very little. We modified the `eval` function to be recursive. We also added a call to `eval` for each of the operands to the `And` operation. That call alone is sufficient to fix the problem; the process of evaluating those arguments represents the  $e_1 \Rightarrow v_1$  and  $e_2 \Rightarrow v_2$  preconditions on the `And` rule, while the use of the resultings values in the tuple causes the match to be against  $v_1$  and  $v_2$  rather than  $e_1$  and  $e_2$ . The above code is a faithful implementation of the value rule and the `And` rule.

We can now complete the boolean language interpreter by continuing the `eval` function in the same form:

```
let rec eval exp =
  match exp with
  | True -> True
  | False -> False
  | Not(exp0) -> (match eval exp0 with
    | True -> False
    | False -> True)
  | And(exp0, exp1) -> (match (eval exp0, eval exp1) with
    | (True, True) -> True
    | (_, False) -> False
    | (False, _) -> False)
  | Or(exp0, exp1) -> (match (eval exp0, eval exp1) with
    | (False, False) -> False
    | (_, True) -> True
    | (True, _) -> True)
```

```

| Implies(exp0,exp1) -> (match (eval exp0, eval exp1) with
  (False,_) -> True
| (True,True) -> True
| (True,False) -> False)

```

The only difference between the operational semantics and the interpreter is that the interpreter is a function. We start with the bottom-left expression in a rule, use the interpreter to recursively produce the value(s) above the line in the rule, and finally compute and return the value below the line in the rule.

Note that the boolean language interpreter above faithfully implements its operational semantics:  $e \Rightarrow v$  if and only if `eval(⟦e⟧)` returns `⟦v⟧` as result. We will go back and forth between these two forms throughout the book. The operational semantics form is used because it is independent of any particular programming language. The interpreter form is useful because we can interpret real programs for nontrivial numbers of steps, something that is difficult to do “on paper” with an operational semantics.

**Definition 2.3** (Metacircular Interpreter). *A **metacircular interpreter** is an interpreter for (possibly a subset of) a language  $x$  that is written in language  $x$ .*

Metacircular interpreters give you some idea of how a language works, but suffer from the non-foundational problems implied in Exercise 2.5. A metacircular interpreter for Lisp (that is, a Lisp interpreter written in Lisp) is a classic programming language theory exercise.

## 2.3 The $\mathbf{F}^b$ Programming Language

Now that we have seen how to define and understand operational semantics, we will begin to study our first programming language:  $\mathbf{F}^b$ .  $\mathbf{F}^b$  is a shunk (flattened) pure *functional* programming language.<sup>2</sup> It has integers, booleans, and higher-order anonymous functions. In most ways  $\mathbf{F}^b$  is much weaker than Caml: there are no reals, lists, types, modules, state, or exceptions.

$\mathbf{F}^b$  is untyped, and in this way is it actually more powerful than Caml. It is possible to write some programs in  $\mathbf{F}^b$  that produce no runtime errors, but which will not typecheck in Caml. For instance, our encoding of recursion in Section 2.3.5 is not typeable in Caml. Type systems are discussed in Chapter 6. Because there are no types, runtime errors can occur in  $\mathbf{F}^b$ , such as the application (5 3).

Although very simplistic,  $\mathbf{F}^b$  is still **Turing-complete**. The concept of Turing-completeness has been defined in numerous equivalent ways. One such definition is as follows:

**Definition 2.4** (Turing Completeness). *A computational model is **Turing-complete** if every **partial recursive function** can be expressed within it.*

<sup>2</sup>Also, any readers familiar with the programming language  $\mathbf{C}\sharp$  as well as basic music theory should find this at least a bit humorous.

This definition, of course, requires a definition of partial recursive functions (also known as computable functions). Without going into an extensive discussion of foundational material, the following somewhat informal definition will suffice:

**Definition 2.5** (Partial Recursive Function). *A function is a partial recursive function if an algorithm exists to calculate it which has the following properties:*

- *The algorithm must have as its input a finite number of arguments.*
- *The algorithm must consist of a finite number of steps.*
- *If the algorithm is given arguments for which the function is defined, it must produce the correct answer within a finite amount of time.*
- *If the algorithm is given arguments for which the function is not defined, it must either produce a clear error or otherwise not terminate. (That is, it must not appear to have produced an incorrect value for the function if no such value is defined.)*

The above definition of a partial recursive function is a mathematical one and thus does not concern itself with execution-specific details such as storage space or practical execution time. No constraints are placed against the amount of memory a computer might need to evaluate the function, the range of the arguments, or that the function terminate before the heat death of the universe (so long as it would eventually terminate for all inputs for which the function is defined).

The practical significance of Turing-completeness is this: there is no computation that could be expressed in another deterministic programming language that cannot be expressed in  $\mathbf{Fb}$ .<sup>3</sup> In fact,  $\mathbf{Fb}$  is even Turing-complete without numbers or booleans. This language, one with only functions and application, is known as the pure lambda-calculus and is discussed briefly in Section 2.4.4. No deterministic programming language can compute more than the partial recursive functions.

### 2.3.1 $\mathbf{Fb}$ Syntax

We will take the same approach in defining  $\mathbf{Fb}$  as we did in defining the boolean language above. We start by describing the grammar of the  $\mathbf{Fb}$  language to define its **concrete syntax**; the **abstract syntax** is deferred until Section 2.3.7. We can define the grammar of  $\mathbf{Fb}$  using the following BNF:

---

<sup>3</sup>This does not guarantee that the  $\mathbf{Fb}$  representation will be pleasant. Programs written in  $\mathbf{Fb}$  to perform even fairly simplistic computations such as determining if one number is less than another are excruciating, as we will see shortly.

$x ::=$	$(a \mid b \mid \dots \mid z)$	<i>lower-case letters</i>
	$(A \mid B \mid \dots \mid Z)$	<i>capital letters</i>
	$  a \mid b \mid \dots \mid z$	<i>lower-case letters</i>
	$  0 \mid 1 \mid \dots \mid 9$	<i>digits</i>
	$  \_ \mid ' \mid \dots )^*$	<i>other characters</i>
$v ::=$	$x$	<i>variable values</i>
	$  \text{True} \mid \text{False}$	<i>boolean values</i>
	$  0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \dots$	<i>integer values</i>
	$  \text{Function } x \rightarrow e$	<i>function values</i>
$e ::=$	$v$	<i>value expressions</i>
	$  (e)$	<i>parenthesized expressions</i>
	$  e \text{ And } e \mid e \text{ Or } e \mid \text{Not } e$	<i>boolean expressions</i>
	$  e + e \mid e - e \mid e = e \mid$	<i>numerical expression</i>
	$  e e$	<i>application expression</i>
	$  \text{If } e \text{ Then } e \text{ Else } e$	<i>conditional expressions</i>
	$  \text{Let Rec } f \ x = e_1 \ \text{In } e_2$	<i>recursive let expression</i>

Note that in accordance with the above BNF, we will be using metavariables  $e$ ,  $v$ , and  $x$  to represent expressions, values, and variables respectively. Note the last point: the metavariable  $x$  refers to an arbitrary  $\mathbf{Fb}$  variable, not necessarily to the  $\mathbf{Fb}$  variable  $x$ .

Associativity in  $\mathbf{Fb}$  works in a fashion very similar to OCaml. Function application, for instance, is left associative, meaning that  $a \ b \ c$  has the same meaning as  $(a \ b) \ c$ . As with any language, this associativity is significant in that it affects how source code is parsed into an AST.

### 2.3.2 Variable Substitution

The main feature of  $\mathbf{Fb}$  is higher-order functions, which also introduces variables. Recall that programs are computed by rewriting them:

```
(Function x -> x + 2)(3 + 2 + 5) ⇒ 12
  because
    3 + 2 + 5 ⇒ 10
      because
        3 + 2 ⇒ 5
      and
        5 + 5 ⇒ 10
  and
    10 + 2 ⇒ 12
```

Note how in this example, the argument is substituted for the variable in the body—this gives us a rewriting interpreter. In other words,  $\mathbf{Fb}$  functions compute by substituting the actual argument for the for parameter; for example,

```
(Function x -> x + 1) 2
```

will compute by substituting 2 for  $x$  in the function's body  $x + 1$ , i.e. by computing  $2 + 1$ . This is not a very efficient method of computing, but it is a very simple and accurate *description* method, and that is what operational semantics is all about – describing clearly and unambiguously how programs are to compute.

**Bound and Free Occurrences of Variables** We need to be careful about how variable substitution is defined. For instance,

(Function  $x \rightarrow$  Function  $x \rightarrow x$ ) 3

should not evaluate to **Function  $x \rightarrow$  3** since the inner  $x$  is bound by the inner parameter. To correctly formalize this notion, we need to make the following definitions.

**Definition 2.6** (Variable Occurrence). A variable use  $x$  **occurs** in  $e$  if  $x$  appears somewhere in  $e$ . Note we refer only to variable uses, not definitions.

**Definition 2.7** (Bound Occurrence). Any occurrences of variable  $x$  in the expression

*Function  $x \rightarrow e$*

are **bound**, that is, any free occurrences of  $x$  in  $e$  are bound occurrences in this expression. Similarly, in the expression

*Let Rec  $f x = e_1$  In  $e_2$*

occurrences of  $f$  and  $x$  are bound in  $e_1$  and occurrences of  $f$  are bound in  $e_2$ . Note that  $x$  is not bound in  $e_2$ , but only in  $e_1$ , the body of the function.

**Definition 2.8** (Free Occurrence). A variable  $x$  occurs **free** in  $e$  if it has an occurrence in  $e$  which is not a bound occurrence.

Let's look at a few examples of bound versus free variable occurrences.

**Example 2.6.**

Function  $\overset{\curvearrowright}{x} \rightarrow x + 1$

$x$  is bound in the body of this function.

**Example 2.7.**

Function  $\overset{\curvearrowright}{x} \rightarrow$  Function  $\overset{\curvearrowright}{y} \rightarrow x + \overset{\uparrow}{y} + z$

$x$  and  $y$  are bound in the body of this function.  $z$  is free.

**Example 2.8.**

Let  $z = 5$  In Function  $x \rightarrow$  Function  $y \rightarrow x + y + z$

$x$ ,  $y$ , and  $z$  are all bound in the body of this function.  $x$  and  $y$  are bound by their respective function declarations, and  $z$  is bound by the **Let** statement. Note that **Fb** does not contain **Let** as syntax, but it can be defined as a macro, in Section 2.3.4 below, and from that it is clear that binding rules work similarly for **Functions** and **Let** statements.

**Example 2.9.**

Function  $x \rightarrow$  Function  $x \rightarrow x + x$

$x$  is bound in the body of this function. Note that both  $x$  usages are bound to the inner variable  $x$ .

**Definition 2.9** (Closed Expression). *An expression  $e$  is closed if it contains no free variable occurrences. All programs we execute are closed (no link-time errors) – non-closed programs don't diverge, we can't even contemplate executing them because they are not in the domain of the evaluation relation.*

Of the examples above, Examples 2.6, 2.8, and 2.9 are closed expressions. Example 2.7 is not a closed expression.

Now that we have an understanding of bound and free variables, we can give a formal definition of variable substitution.

**Definition 2.10** (Variable Substitution). *The variable substitution of  $x$  for  $e'$  in  $e$ , denoted  $e[e'/x]$ , is the expression resulting from the operation of replacing all free occurrences of  $x$  in  $e$  with  $e'$ . For now, we assume that  $e'$  is a closed expression.*

Here is an equivalent inductive definition of substitution:

$$\begin{aligned}
 x[v/x] &= v \\
 x'[v/x] &= x' && x \neq x' \\
 (\text{Function } x \rightarrow e)[v/x] &= (\text{Function } x \rightarrow e) \\
 (\text{Function } x' \rightarrow e)[v/x] &= (\text{Function } x' \rightarrow e[v/x]) && x \neq x' \\
 n[v/x] &= n \text{ for } n \in \mathbb{Z} \\
 \text{True}[v/x] &= \text{True} \\
 \text{False}[v/x] &= \text{False} \\
 (e_1 + e_2)[v/x] &= e_1[v/x] + e_2[v/x] \\
 (e_1 \text{ And } e_2)[v/x] &= e_1[v/x] \text{ And } e_2[v/x] \\
 &\vdots
 \end{aligned}$$

For example, let us consider a simple application of a function: **(Function  $x \rightarrow x + 1$ ) 2**. We know that, to evaluate this function, we simply replace all instances of  $x$  in the body with 2. This is written  $(x + 1)[2/x]$ . Given the above definition, we can conclude that the result must be 3.



While this may not seem like an illuminating realization, the fact that this is mathematically discernable gives us a starting point for more complex substitutions. Consider the following example.

**Example 2.10.**

**Expression:**  
`(Function x -> Function y -> (x + x + y)) 5`

**Substitution:**  
`(Function y -> (x + x + y))[5/x]`  
`= (Function y -> (x + x + y)[5/x])`  
`= Function y -> (x[5/x] + x[5/x] + y[5/x])`  
`= Function y -> (5 + 5 + y)`

### $\alpha$ -conversion

In Example 2.9, we saw that it is possible for two variables to share the same name. The variables themselves are, of course, distinct and follow the same rules of scope in  $\mathbf{F}^b$  as they do in Caml. But reading expressions which make frequent use of the same variable name for different variables can be very disorienting. For example, consider the following expression.

```
Let Rec f x =
  If x = 1 Then
    (Function f -> f (x - 1)) (Function x -> x)
  Else
    f (x - 1)
In f 100
```

How does this expression evaluate? It is a bit difficult to tell simply by looking at it because of the tricky bindings. We can make it much easier to understand by using different names.  $\alpha$ -conversion is the process of replacing a variable definition and all occurrences bound to it with a variable of a different name.

**Example 2.11.**

`Function  $\bar{x}$  -> x + 1`  
 becomes  
`Function  $\bar{z}$  -> z + 1`

Example 2.11 shows a simple case in which  $x$  is substituted for  $z$ . For cases in which the same variable name is used numerous times, we can use the same approach. Consider Example 2.12 in which the inner variable  $x$  is  $\alpha$ -converted to  $z$ .

**Example 2.12.**

`Function x -> Function  $\bar{x}$  -> x`  
 becomes  
`Function x -> Function  $\bar{z}$  -> z`

Similarly, we could rename the outer variable to  $z$  as shown in Example 2.13. Note that in this case, the occurrence of  $x$  is *not* changed, as it is bound by the inner variable and not the outer one.

**Example 2.13.**

Function  $\lambda x. \lambda x. x$   
 becomes  
 Function  $\lambda z. \lambda x. x$

Let's figure out what variable occurrences are bound to which function in our previous confusing function and rewrite the function in a clearer way by using  $\alpha$ -conversion. One possible result is as follows:

```
Let Rec f x =
  If x = 1 Then
    (Function z -> z (x - 1)) (Function y -> y)
  Else
    f (x - 1)
In f 100
```

Now it's much easier to understand what is happening. If the function  $f$  is applied to an integer which is not 1, it merely applies itself again to the argument which is one less than the one it received. Since we are evaluating  $f\ 100$ , that results in  $f\ 99$ , which in turn evaluates  $f\ 98$ , and so on. Eventually,  $f\ 1$  is evaluated.

When  $f\ 1$  is evaluated, we explore the other branch of the `If` expression. We know that  $x$  is 1 at this point, so we can see that the evaluated expression is  $(\text{Function } z \rightarrow z\ 0)\ (\text{Function } y \rightarrow y)$ . Using substitution gives us  $(\text{Function } y \rightarrow y)\ 0$ , which in turn gives us 0. So we can conclude that the expression above will evaluate to 0.

Observe, however, that we did not formally prove this; so far, we have been treating substitution and other operations in a cavalier fashion. In order to create a formal proof, we need a set of operational semantics which dictates how evaluation works in  $\mathbf{F}^b$ . Section 2.3.3 walks through the process of creating an operational semantics for the  $\mathbf{F}^b$  language and gives us the tools needed to prove what we concluded above.

### 2.3.3 Operational Semantics for $\mathbf{F}^b$

We are now ready to begin defining operational semantics for  $\mathbf{F}^b$ . For the same reasons as in our boolean language, we will need a rule which relates values to values in  $\Rightarrow$ :

(Value Rule)  $\frac{}{v \Rightarrow v}$

We can also define boolean operations for  $\mathbf{F}^b$  in the same way as we did for the boolean language above. Note, however, that not all values in  $\mathbf{F}^b$  are booleans. Fortunately, our definition of the rules addresses this for us, as there is (for example) no logical and of the values 5 and 3. That is, we know that these rule only apply to  $\mathbf{F}^b$  boolean values because they use operations which are only defined for  $\mathbf{F}^b$  boolean values.

$$\begin{array}{l}
\text{(Not Rule)} \quad \frac{e \Rightarrow v}{\text{Not } e \Rightarrow \text{the negation of } v} \\
\text{(And Rule)} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \text{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2} \\
\vdots
\end{array}$$

We can also define operations over integers in much the same way. For sake of clarity, we will explicitly restrict these rules such that they operate only on expressions which evaluate to integers.

$$\begin{array}{l}
\text{(+ Rule)} \quad \frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 + e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2} \\
\text{(- Rule)} \quad \frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 - e_2 \Rightarrow \text{the integer difference of } v_1 \text{ and } v_2}
\end{array}$$

As with the boolean rules, observe that these rules allow the  $\Rightarrow$  relation to be applied recursively:  $5 + (4 - 3)$  can be evaluated using the + rule because  $4 - 3$  can be evaluated using the - rule first.

These rules allow us to write **Fb** programs containing boolean expressions or **Fb** programs containing integer expressions, but we currently have no way to combine the two. There are two mechanisms we use to mix the two in a program: conditional expressions and comparison operators. The only comparison operator in **Fb** is the = operator, which compares the values of integers. We define the = rule as follows.

$$\text{(= Rule)} \quad \frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 = e_2 \Rightarrow \text{True if } v_1 \text{ and } v_2 \text{ are identical, else False}}$$

Note that the = rule is defined only where  $v_1$  and  $v_2$  are integers. Due to this constraint, the expression  $\text{True} = \text{True}$  is not evaluatable in **Fb**. This is, of course, a matter of choice; as a language designer, one may choose to remove that constraint and allow boolean values to be compared directly. To formalize this, however, a change to the rules would be required. A faithful implementation of **Fb** using the above = rule is *required* to reject the expression  $\text{True} = \text{True}$ .

An intuitive definition of a conditional expression is that it evaluates to the value of one expression if the boolean is true and the value of the other expression if the boolean is false. While this is true, the particulars of how this is expressed in the rule are vital. Let us consider the following flawed attempt at a conditional expression rule:

$$\text{(Flawed If Rule)} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \Rightarrow v_2 \text{ if } v_1 \text{ is True, } v_3 \text{ otherwise}}$$

It seems that this rule allows us to evaluate many of the conditional expressions we want to evaluate. But let us consider this expression:

`If True Then 0 Else (True + True)`

If we attempted to apply the above rule to the previous expression, we would find that the precondition  $e_3 \Rightarrow v_3$  would not hold; there is no rule which can relate `True + True`  $\Rightarrow v$  for any  $v$  since the `+` rule only applies to integers. Nonetheless, we want the expression to evaluate to 0. So how can we achieve this result?

In this case, we have no choice but to write two different rules with distinct preconditions. We can capture all of the relationships in the previous rule and yet allow expressions such as the previous to evaluate by using the following two rules:

$$\text{(If True Rule)} \quad \frac{e_1 \Rightarrow \text{True}, \quad e_2 \Rightarrow v_2}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \Rightarrow v_2}$$

$$\text{(If False Rule)} \quad \frac{e_1 \Rightarrow \text{False}, \quad e_3 \Rightarrow v_3}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \Rightarrow v_3}$$

Again, the key difference between these two rules is that they have different sets of preconditions. Note that the `If True` rule does not evaluate  $e_3$ , nor does the `If False` rule evaluate  $e_2$ . This allows the untraveled logic paths to contain unevaluatable expressions without necessarily preventing the expression containing them from evaluating.

### Application

We are now ready to approach one of the most difficult **Fb** rules: application. How can we formalize the evaluation of an expression like `(Function x -> x + 1) (5 + 2)`? We saw in Section 2.3.2 that we can evaluate a function application by using variable substitution. As we have a mathematical definition for the substitution operation, we can base our function application rule around it.

Suppose we wish to evaluate the above expression. We can view application in two parts: the function being applied and the argument to the function. We wish to know to what the expression evaluates; thus, we are trying to establish that  $e_1 e_2 \Rightarrow v$  for some  $v$ .

$$\text{(Application Rule (Part 1))} \quad \frac{?}{e_1 e_2 \Rightarrow v}$$

In our boolean operations, we needed to evaluate the arguments before attempting an operation over them (in order to allow recursive expressions). The same is true of our application rule; in `(Function x -> x + 1) (5 + 2)`, we must evaluate `5 + 2` before it can be used as an argument.<sup>4</sup> We must do likewise with the function we are applying.

<sup>4</sup>Actually, some languages would perform substitution before evaluating the expression, but **Fb** and most traditional languages do not. Discussion of this approach is handled in Section 2.3.6.

$$(Application\ Rule\ (Part\ 2)) \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

We obviously aren't finished, though, as we still don't have any preconditions which allow us to relate  $v$  to something. Additionally, we know we will need to use variable substitution, but we have no metavariables representing  $\mathbf{F}^b$  variables in the above rule. We can fix this by reconsidering how we evaluate the first argument; we know that the application rule only works when applying *functions*. In restricting our rule to applying functions, we can name some metavariables to describe the function's contents.

$$(Application\ Rule\ (Part\ 3)) \quad \frac{e_1 \Rightarrow \mathbf{Function}\ x \rightarrow e \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

In the above rule,  $x$  is the metavariable representing the function's variable while  $e$  represents the function's body. Now that we have this information, we can define function application in terms of variable substitution. When we apply  $\mathbf{Function}\ x \rightarrow x + 1$  to a value such as 7, we wish to replace all instances of  $x$ , the function's variable, in the function's body with 7, the provided argument. Formally,

$$(Application\ Rule) \quad \frac{e_1 \Rightarrow \mathbf{Function}\ x \rightarrow e \quad e_2 \Rightarrow v_2 \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

### $\mathbf{F}^b$ Recursion

We now have a very complete set of rules for the  $\mathbf{F}^b$  language. We do not, however, have a rule for  $\mathbf{Let}\ \mathbf{Rec}$ . As we will see,  $\mathbf{Let}\ \mathbf{Rec}$  is not actually necessary in basic  $\mathbf{F}^b$ ; it is possible to build recursion out of the rules we already have. Later, however, we will create variants of  $\mathbf{F}^b$  with type systems in which it will be impossible for that approach to recursion to work. For that reason as well as our immediate convenience, we will define the  $\mathbf{Let}\ \mathbf{Rec}$  rule now.

Again, we start with an iterative approach. We know that we want to be able to evaluate the  $\mathbf{Let}\ \mathbf{Rec}$  expression, so we provide metavariables to represent the components of the expression itself.

(Recursive Application Rule (Part 1))

$$\frac{?}{\mathbf{Let}\ \mathbf{Rec}\ f\ x = e_1\ \mathbf{In}\ e_2 \Rightarrow v}$$

Let us consider what we wish to accomplish. Consider for a moment a recursive approach to the summation of the numbers between 1 and 5:

```

Let Rec f x =
  If x = 1 Then
    1
  Else
    f (x - 1) + x
In f 5

```

If we focus on the last line (`In f 5`), we can see that we want the body of the recursive function to be applied to the value 5. We can write our rule accordingly by replacing `f` with the function's body. We must make sure to use the same metavariable to represent the function's argument in order to allow the new function body's variable to be captured. We reach the following rule.

(Recursive Application Rule (Part 2))

$$\frac{e_2[(\text{Function } x \rightarrow e_1)/f] \Rightarrow v}{\text{Let Rec } f x = e_1 \text{ In } e_2 \Rightarrow v}$$

We can test our new rule by applying it to our recursive summation above.

$$\frac{\frac{\frac{\text{Function } x \rightarrow \dots \Rightarrow \text{Function } x \rightarrow \dots}{5 \Rightarrow 5} \quad \frac{\frac{\frac{???}{f(5-1) \Rightarrow v'} \quad 5 \Rightarrow 5}{f(5-1) + 5 \Rightarrow v}}{5 = 1 \Rightarrow \text{False}}}{\text{If } 5 = 1 \text{ Then } 1 \text{ Else } f(5-1) + 5 \Rightarrow v}}{(\text{Function } x \rightarrow \text{If } x = 1 \text{ Then } 1 \text{ Else } f(x-1) + x) 5 \Rightarrow v}}{\text{Let Rec } f x = \text{If } x = 1 \text{ Then } 1 \text{ Else } f(x-1) + x \text{ In } f 5 \Rightarrow v}$$

As foreshadowed by its label above, our recursion rule is not yet complete. When we reach the evaluation of `f (5-1)`, we are at a loss; `f` is not bound. Without a binding for `f`, we have no way of repeating the recursion.

In addition to replacing the invocation of `f` with function application in  $e_2$ , we need to ensure that any occurrences of `f` in the body of the function itself are bound. To what do we bind them? We could try to replace them with function applications as well, but this leads us down a rabbit hole; each function application would require yet another replacement. We can, however, replace applications of `f` in  $e_1$  with *recursive* applications of `f` by reintroducing the `Let Rec` syntax. This leads us to the following application rule:

(Recursive Application Rule (Part 3))

$$\frac{e_2[\text{Function } x \rightarrow e_1[(\text{Let Rec } f x = e_1 \text{ In } f)/f]/f] \Rightarrow v}{\text{Let Rec } f x = e_1 \text{ In } e_2 \Rightarrow v}$$

While this makes a certain measure of sense, it isn't quite correct. In Section 2.3.2, we saw that substitution must replace a variable with a *value*, while the `Let Rec` term

above is an expression. Fortunately, we have functions as values; thus, we can put the expression inside of a function and ensure that we call it with the appropriate argument.

(Recursive Application Rule)

$$\frac{e_2[\text{Function } x \rightarrow e_1[(\text{Function } x \rightarrow \text{Let Rec } f x = e_1 \text{ In } f x)/f]/f] \Rightarrow v}{\text{Let Rec } f x = e_1 \text{ In } e_2 \Rightarrow v}$$

Now, instead of encountering `f (5-1)` when we evaluate the summation example, we encounter `Let Rec f x = If x = 1 Then 1 Else f (x-1) + x In f (5-1)`. This allows us to recurse back into the `Let Rec` rule. Eventually, we may reach a branch which does not evaluate the `Else` side of the conditional expression, in which case that `Let Rec` is not expanded (allowing us to terminate). Each application of the rule effectively “unrolls” one level of recursion.

In summary, we have the following operational semantics for  $\mathbf{Fb}$  (excluding some repetitive rules such as the `-` rule):

$$\begin{array}{l} \text{(Value Rule)} \quad \frac{}{v \Rightarrow v} \\ \\ \text{(Not Rule)} \quad \frac{e \Rightarrow v}{\text{Not } e \Rightarrow \text{the negation of } v} \\ \\ \text{(And Rule)} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \text{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2} \\ \\ \text{(+ Rule)} \quad \frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 + e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2} \\ \\ \text{(= Rule)} \quad \frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 = e_2 \Rightarrow \text{True if } v_1 \text{ and } v_2 \text{ are identical, else False}} \\ \\ \text{(If True Rule)} \quad \frac{e_1 \Rightarrow \text{True}, \quad e_2 \Rightarrow v_2}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \Rightarrow v_2} \\ \\ \text{(If False Rule)} \quad \frac{e_1 \Rightarrow \text{False}, \quad e_3 \Rightarrow v_3}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \Rightarrow v_3} \\ \\ \text{(Application Rule)} \quad \frac{e_1 \Rightarrow \text{Function } x \rightarrow e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow v}{e_1 e_2 \Rightarrow v} \end{array}$$

(Let Rec)

$$\frac{e_2[\text{Function } x \rightarrow e_1[(\text{Function } x \rightarrow \text{Let Rec } f x = e_1 \text{ In } f x)/f]/f] \Rightarrow v}{\text{Let Rec } f x = e_1 \text{ In } e_2 \Rightarrow v}$$

Let us consider a few examples of proof trees using the  $\mathbf{Fb}$  operational semantics.

**Example 2.14.**

**Expression:**

If 3 = 4 Then 5 Else 4 + 2

**Proof:**

$$\frac{\frac{3 \Rightarrow 3}{3 = 4 \Rightarrow \text{False}} \quad \frac{4 \Rightarrow 4}{4 + 2 \Rightarrow 6}}{4 \Rightarrow 4} \quad \frac{2 \Rightarrow 2}{4 + 2 \Rightarrow 6}}{\text{If } 3 = 4 \text{ Then } 5 \text{ Else } 4 + 2 \Rightarrow 6}$$

**Example 2.15.**

**Expression:**

(Function x -> If 3 = x Then 5 Else x + 2) 4

**Proof:**

$$\frac{\text{Function } x \rightarrow \dots \Rightarrow \text{Function } x \rightarrow \dots \quad \frac{4 \Rightarrow 4}{\text{If } 3 = 4 \text{ Then } 5 \text{ Else } 4 + 2 \Rightarrow 6}}{\text{Function } x \rightarrow \text{If } 3 = x \text{ Then } 5 \text{ Else } x + 2) 4 \Rightarrow 6} \quad \text{by Example 2.14}$$

**Example 2.16.**

**Expression:**

(Function f -> Function x -> f(f x))(Function y -> y - 1) 4

**Proof:**

*Due to the size of the proof, it is broken into multiple parts. We use  $v \Rightarrow \star$  as an abbreviation for  $v \Rightarrow v$  (when  $v$  is lengthy) for brevity.*

**Part 1:**

$$\frac{\text{Function } f \rightarrow \text{Function } x \rightarrow f(f x) \Rightarrow \star \quad \text{Function } y \rightarrow y - 1 \Rightarrow \star \quad \text{Function } x \rightarrow (\text{Function } y \rightarrow y - 1) ((\text{Function } y \rightarrow y - 1) x) \Rightarrow \star}{(\text{Function } f \rightarrow \text{Function } x \rightarrow f(f x))(\text{Function } y \rightarrow y - 1) \Rightarrow (\text{Function } x \rightarrow (\text{Function } y \rightarrow y - 1) ((\text{Function } y \rightarrow y - 1) x)) \Rightarrow \star}$$

**Part 2:**

$$\frac{\frac{\text{Function } y \rightarrow y - 1 \Rightarrow \star \quad \frac{\frac{4 \Rightarrow 4}{4 - 1 \Rightarrow 3} \quad \frac{3 \Rightarrow 3}{3 - 1 \Rightarrow 2}}{(\text{Function } y \rightarrow y - 1) 4 \Rightarrow 3}}{\text{Function } y \rightarrow y - 1 \Rightarrow \star} \quad \frac{4 \Rightarrow 4}{(\text{Function } y \rightarrow y - 1) ((\text{Function } y \rightarrow y - 1) 4) \Rightarrow 2}}{\text{Function } y \rightarrow y - 1 \Rightarrow \star} \quad \frac{4 \Rightarrow 4}{(\text{Function } f \rightarrow \text{Function } x \rightarrow f(f(x)))(\text{Function } y \rightarrow y - 1) 4 \Rightarrow 2}}{\text{(by part 1)} \quad 4 \Rightarrow 4}$$



**Interact with Fb.** Tracing through recursive evaluations is difficult, and therefore the reader should invest some time in exploring the semantics of **Let Rec**. A good way to do this is by using the **Fb** interpreter. Try evaluating the expression we looked at above:

```
# Let Rec f x =
  If x = 1 Then 1 Else x + f (x - 1)
  In f 3;;
==> 6
```

Another interesting experiment is to evaluate a recursive function without applying it. Notice that the result is equivalent to a single application of the **Let Rec** rule. This is a good way to see how the “unwrapping” actually takes place:



```

# Let Rec f x =
  If x = 1 Then 1 Else x + f (x - 1)
  In f;;
==> Function x ->
  If x = 1 Then
    1
  Else
    x + (Let Rec f x =
      If x = 1 Then
        1
      Else
        x + (f) (x - 1)
    In
      f) (x - 1)

```

---

As we mentioned before, one of the main benefits of defining an operational semantics for a language is that we can rigorously verify claims about that language. Now that we have defined the operational semantics for  $\mathbf{Fb}$ , we can prove a few things about it.

**Lemma 2.3.**  *$\mathbf{Fb}$  is deterministic.*

*Proof.* By inspection of the rules, at most one rule can apply at any time.  $\square$

**Lemma 2.4.**  *$\mathbf{Fb}$  is not normalizing.*

*Proof.* To show that a language is not normalizing, we simply show that there is some  $e$  such that there is no  $v$  with  $e \Rightarrow v$ . Let  $e$  be  $(\text{Function } x \rightarrow x \ x)(\text{Function } x \rightarrow x \ x)$ .  $e \not\Rightarrow v$  for any  $v$ . Thus,  $\mathbf{Fb}$  is not normalizing.  $\square$

The expression in this proof is a very interesting one which we will examine in more detail in Section 2.3.5. It does not evaluate to a value because each step in its evaluation produces itself as a precondition. This is roughly analogous to trying to prove proposition  $A$  by using  $A$  as a given.

In this case, the expression does not evaluate because it never runs out of work to do. This is not the only kind of non-normalizing expression which appears in  $\mathbf{Fb}$ ; another kind consists of those expressions for which no evaluation rule applies. For example,  $(4 \ 3)$  is a simpler expression that is non-normalizing. No rule exists to evaluate  $(e_1 \ e_2)$  when  $e_1$  is not a function expression.

Both of these cases look like divergence as far as the operational semantics are concerned. In the case of an interpreter, however, these two kinds of expressions behave differently. The expression which never runs out of work to do typically causes an interpreter to loop infinitely (as most interpreters are not clever enough to realize that they will never finish). The expressions which attempt application to non-functions, on the other hand, cause an interpreter to provide a clear error in the form of an exception. This is because the error here is easily detectable; the interpreter, in attempting to evaluate these expressions, can quickly discover that none of its available rules apply to the expression and respond by raising an exception to that effect. Nonetheless, both are theoretically equivalent.

### 2.3.4 The Expressiveness of $\mathbf{Fb}$

$\mathbf{Fb}$  doesn't have many features, but it is possible to do much more than it may seem. As we said before,  $\mathbf{Fb}$  is Turing complete, which means, among other things, that any Caml operation may be encoded in  $\mathbf{Fb}$ . We can informally represent encodings in our interpreter as macros using Caml `let` statements. A macro is equivalent to a statement like “let  $F$  be `Function x -> ...`”

**Logical Combinators** First, there are the classic **logical combinators**, simple functions for recombining data.

```
combI = Function x -> x
combK = Function x -> Function y -> x
combS = Function x -> Function y -> Function z -> (x z) (y z)
combD = Function x -> x x
```

**Tuples** Tuples and lists are encodable from just functions, and so they are not needed as primitives in a language. Of course for an *efficient* implementation you would want them to be primitives; thus doing this encoding is simply an exercise to better understand the nature of functions and tuples. We will define a 2-tuple (pairing) constructor; From a pair you can get a  $n$ -tuple by building it from pairs. For example,  $(1, (2, (3, 4)))$  represents the 4-tuple  $(1, 2, 3, 4)$ .

First, we need to define a pair constructor, `pr`. A first (i.e., slightly buggy) approximation of the constructor is as follows.

```
pr (e1, e2)  $\stackrel{\text{def}}{=} \text{Function } x \rightarrow x \ e_1 \ e_2$ 
```

We use the notation  $a \stackrel{\text{def}}{=} b$  to indicate that  $a$  is an abbreviation for  $b$ . For example, we might have a problem in which the incrementer function is commonly used; it would make sense, then, to define `incx`  $\stackrel{\text{def}}{=} \text{Function } x \rightarrow x + 1$ . Note that such abbreviations do not change the underlying meaning of the expression; it is simply for convenience. The same concept applies to macro definitions in programming languages. By creating a new macro, one is not changing the math behind the programming language; one is merely defining a more terse means of expressing a concept.

Based on the previous definition of `pr`, we can define the following macros for projection:

```
left (e)  $\stackrel{\text{def}}{=} e \ (\text{Function } x \rightarrow \text{Function } y \rightarrow x)$ 
right (e)  $\stackrel{\text{def}}{=} e \ (\text{Function } x \rightarrow \text{Function } y \rightarrow y)$ 
```

Now let's take a look at what's happening. `pr` takes a left expression,  $e_1$ , and a right expression,  $e_2$ , and packages them into a function that applies its argument  $x$  to  $e_1$  and

$e_2$ . Because functions are values, the result won't be evaluated any further, and  $e_1$  and  $e_2$  will be packed away in the body of the function until it is applied. Thus `pr` succeeds in "storing"  $e_1$  and  $e_2$ .

All we need now is a way to get them out. For that, look at how the projection operations `left` and `right` are defined. They're both very similar, so let's concentrate only on the projection of the left element. `left` takes one of our pairs, which is encoded as a function, and applies it to a curried function that returns its first, or leftmost, element. Recall that the pair itself is just a function that applies its argument to  $e_1$  and  $e_2$ . So when the curried `left` function that was passed in is applied to  $e_1$  and  $e_2$ , the result is  $e_1$ , which is exactly what we want. `right` is similar, except that the curried function returns its second, or rightmost, argument.

Before we go any further, there is a technical problem involving our encoding of `pr`. Suppose  $e_1$  or  $e_2$  contain a free occurrence of  $x$  when `pr` is applied. Because `pr` is defined as `Function x -> x e1 e2`, any free occurrence  $x$  contained in  $e_1$  or  $e_2$  will become bound by  $x$  after `pr` is applied. This is known as variable **capture**. To deal with capture here, we need to change our definition of `pr` to the following.

$$\text{pr } (e_1, e_2) \stackrel{\text{def}}{=} (\text{Function } e1 \rightarrow \text{Function } e2 \rightarrow \text{Function } x \rightarrow x e1 e2) e_1 e_2$$

This way, instead of textually substituting for  $e_1$  and  $e_2$  directly, we pass them in as functions. This allows the interpreter evaluate  $e_1$  and  $e_2$  to values before passing them in, and also ensures that  $e_1$  and  $e_2$  are closed expressions. This eliminates the capture problem, because any occurrence of  $x$  is either bound by a function declaration *inside*  $e_1$  or  $e_2$ , or was bound outside the entire `pr` expression, in which case it must have already been replaced with a value at the time that the `pr` subexpression is evaluated. Variable capture is an annoying problem that we will see again in Section 2.4.

Now that we have polished our definitions, let's look at an example of how to use these encodings. First, let's create create the pair  $p$  as  $(4, 5)$ .

$$p \stackrel{\text{def}}{=} \text{pr } (4, 5) \Rightarrow \text{Function } x \rightarrow x 4 5$$

Now, let's project the left element of  $p$ .

$$\text{left } p \equiv (\text{Function } x \rightarrow x 4 5) (\text{Function } x \rightarrow \text{Function } y \rightarrow x)$$

We use the notation  $a \equiv b$  to indicate the expansion of a specific macro instance. In this case, `left p` expands to what we see above, which then becomes

$$(\text{Function } x \rightarrow \text{Function } y \rightarrow x) 4 5 \Rightarrow 4.$$

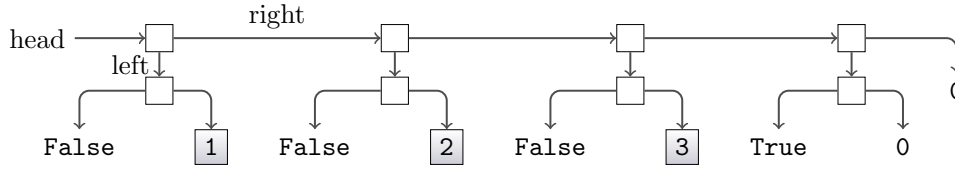


Figure 2.1: Lists Implemented Via Pairs

This encoding works, and has all the expressiveness of real tuples. There are, nonetheless, a few problems with it. First of all, consider

```
left (Function x -> 0) ⇒ 0.
```

We really want the interpreter to produce a run-time error here, because a function is not a pair.

Similarly, suppose we wrote the program `right(pr(3, pr(4, 5)))`. One would expect this expression to evaluate to `pr(4, 5)`, but remember that pairs are not values in our language, but simply encodings, or macros. So in fact, the result of the computation is `Function x -> x 4 5`. We can only guess that this is intended to be a pair. In this respect, the encoding is flawed, and we will, in Chapter 3, introduce “real”  $n$ -tuples into an extension of  $\mathbf{Fb}$  to alleviate these kinds of problems.

**Lists** Lists can also be implemented via pairs. In fact, pairs of pairs are technically needed because we need a flag to mark the end of list. The list `[1; 2; 3]` is represented by `pr (pr(false,1), pr (pr(false,2), pr (pr(false,3), emptylist)))` where `emptylist`  $\stackrel{\text{def}}{=} \text{pr}(\text{pr}(\text{true},0),0)$ . The true/false flag is used to mark the end of the list: only the empty list is flagged true. The implementation is as follows.

```

cons (x, y)  $\stackrel{\text{def}}{=} \text{pr}(\text{pr}(\text{Bool false}, x), y)$ 
emptylist  $\stackrel{\text{def}}{=} \text{pr}(\text{pr}(\text{Bool true}, \text{Int } 0), \text{Int } 0)$ 
head x  $\stackrel{\text{def}}{=} \text{right}(\text{left } x)$ 
tail x  $\stackrel{\text{def}}{=} \text{right } x$ 
isempty l  $\stackrel{\text{def}}{=} (\text{left } (\text{left } l))$ 
length  $\stackrel{\text{def}}{=} \text{Let Rec len } x =$ 
  If isempty(x) Then 0 Else 1 + len (tail x) In len
  
```

In addition to tuples and lists, there are several other concepts from Caml that we can encode in  $\mathbf{Fb}$ . We review a few of these encodings below. For brevity and readability, we will switch back to the concrete syntax.

**Functions with Multiple Arguments** Functions with multiple arguments are done with currying just as in Caml. For example

```
Function x -> Function y -> x + y
```

**The Let Operation** Let is quite simple to define in  $\mathbf{Fb}$ .

$$(\text{Let } x = e \text{ In } e') \stackrel{\text{def}}{=} (\text{Function } x \text{ -> } e') e$$

For example,

$$(\text{Let } x = 3 + 2 \text{ In } x + x) \equiv (\text{Function } x \text{ -> } x + x) (3 + 2) \Rightarrow 10.$$

**Sequencing** Notice that  $\mathbf{Fb}$  has no sequencing ( $;$ ) operation. Because  $\mathbf{Fb}$  is a pure functional language, writing  $e; e'$  is really just equivalent to writing  $e'$ , since  $e$  will never get used. Hence, sequencing really only has meaning in languages with side-effects. Nonetheless, sequencing is definable in the following manner.

$$e; e' \stackrel{\text{def}}{=} (\text{Function } n \text{ -> } e') e,$$

where  $n$  is chosen so as not to be free in  $e'$ . This will first execute  $e$ , throw away the value, and then execute  $e'$ , returning its result as the final result of  $e; e'$ .

**Freezing and Thawing** We can stop and re-start computation at will by freezing and thawing.

$$\text{Freeze } e \stackrel{\text{def}}{=} \text{Function } n \text{ -> } e$$

$$\text{Thaw } e \stackrel{\text{def}}{=} e 0$$

We need to make sure that  $n$  is a fresh variable so that it is not free in  $e$ . Note that the 0 in the application could be any value. **Freeze**  $e$  freezes  $e$ , keeping it from being computed. **Thaw**  $e$  starts up a frozen computation. As an example,

```
Let x = Freeze (2 + 3) In (Thaw x) + (Thaw x)
```

This expression has same value as the equivalent expression without the freeze and thaw, but the  $2 + 3$  is evaluated twice. Again, in a pure functional language the only difference is that freezing and thawing is less efficient. In a language with side-effects, if the frozen expression causes a side-effect, then the freeze/thaw version of the function may produce results different from those of the original function, since the frozen side-effects will be applied as many times as they are thawed.

### 2.3.5 Russell's Paradox and Encoding Recursion

**Fb** has a built-in `Let Rec` operation to aid in writing recursive functions, but its actually not needed because recursion is definable in **Fb**. The encoding is a non-obvious one, and so before we introduce it, we present some background information. As we will see, the encoding of recursion is closely related to a famous set-theoretical paradox due to Russell.

Let us begin by posing the following question. *How can programs compute forever in **Fb** without recursion?* The answer to this question comes in the form of a seemingly simple expression:

```
(Function x -> x x)(Function x -> x x)
```

Recall from Lemma 2.2, that a corollary to the existence of this expression is that **Fb** is not normalizing. This computation is odd in some sense.  $(x x)$  is a function being applied to itself. There is a logical paradox at the heart of this non-normalizing computation, namely Russell's Paradox.

#### Russell's Paradox

In Frege's set theory (circa 1900), sets were written as predicates  $P(x)$ . We can view predicates as single-argument functions which return a boolean value: true if the argument is in the set represented by the predicate and false if it is not. Testing membership in a set is done via application of the predicate. For example, consider the predicate

```
Function x -> (x = 2 Or x = 3 Or x = 5)
```

This predicate represents the integer set  $\{2, 3, 5\}$  since it will return `True` for any of the elements in that set and `False` for all other arguments. If we were to extend **Fb** to include a native integer less-than operator, the predicate

```
Function x -> x < 2
```

would represent an infinitely-sized set containing all integer values less than 2 (as **Fb** still has no notion of real numbers). In general, given a predicate  $P$  representing a set  $S$ ,

$e \in S$  iff  $P e \Rightarrow \text{True}$

Russell discovered a paradox in Frege’s set theory, and it can be expressed in the following way.

**Definition 2.11** (Russell’s Paradox). *Let  $P$  be the set of all sets that do not contain themselves as members. Is  $P$  a member of  $P$ ?*

Asking whether or not a set is a member of itself seems like strange question, but in fact there are many sets that are members of themselves. The infinitely receding set  $\{\{\{\dots\}\}\}$  has itself as a member. The set of things that are not apples is also a member of itself (clearly, a set of non-apples is not an apple). These kinds of sets arise only in “non-well-founded” set theory.

To explore the nature of Russell’s Paradox, let us try to answer the question it poses: Does  $P$  contain itself as a member? Suppose the answer is yes, and  $P$  does contain itself as a member. If that were the case then  $P$  should not be in  $P$ , which is the set of all sets that do *not* contain themselves as members. Suppose, then, that the answer is no, and that  $P$  does not contain itself as a member. Then  $P$  should have been included in  $P$ , since it doesn’t contain itself. In other words,  $P$  is a member of  $P$  if and only if it isn’t. Hence Russell’s Paradox is indeed a paradox.

This can also be illustrated by using **Fb** functions as predicates. Specifically, we will write a predicate for  $P$  above. We must define  $P$  to accept an argument (which we know to be a set - a predicate in our model) and determine if it contains itself (pass it to itself as an argument). Thus, our representation of  $P$  is **Function**  $x \rightarrow \text{Not}(x x)$ . We merely apply  $P$  to itself to get our answer.

**Definition 2.12** (Computational Russell’s Paradox). *Let*

$P \stackrel{\text{def}}{=} \text{Function } x \rightarrow \text{Not}(x x)$ .

*What is the result of  $P P$ ? Namely, what is*

*$(\text{Function } x \rightarrow \text{Not}(x x)) (\text{Function } x \rightarrow \text{Not}(x x))$ ?*

If this **Fb** program were evaluated, it would run forever. We can informally detect the pattern just by looking at a few passes of an evaluation proof:

$$\begin{array}{c} \vdots \\ \hline \text{Not } (\text{Not } ((\text{Function } x \rightarrow \text{Not } (x x))(\text{Function } x \rightarrow \text{Not } (x x)))) \\ \hline \text{Not } ((\text{Function } x \rightarrow \text{Not } (x x)) (\text{Function } x \rightarrow \text{Not } (x x))) \\ \hline (\text{Function } x \rightarrow \text{Not } (x x)) (\text{Function } x \rightarrow \text{Not } (x x)) \end{array}$$

We know that `Not (Not (e))` evaluates to the same value as  $e$ .<sup>5</sup> We can see that we're going in circles. Again, this statement tells us that  $P \Rightarrow \text{True}$  if and only if  $P \Rightarrow \text{False}$ .

This is not how Russell viewed his paradox, but it has the same core structure; it is simply rephrased in terms of computation, and not set theory. The computational realization of the paradox is that the predicate doesn't compute to true or false, so it's not a sensible logical statement. Russell's discovery of this paradox in Frege's set theory shook the foundations of mathematics. To solve this problem, Russell developed his ramified theory of types, which is the ancestor of types in programming languages. The program

```
(function x -> not(x x)) (function x -> not(x x))
```

is not typeable in Caml for the same reason the corresponding predicate is not typeable in Russell's ramified theory of types. Try typing the above code into the Caml top-level and see what happens.

More information on Russell's Paradox may be found in [15].

### Encoding Recursion by Passing Self

In the logical view, passing a function to itself as argument is a bad thing. From a programming view, however, it can be an extremely powerful tool: passing a function to itself allows recursive functions to be defined without the use of `Let Rec`. We now show how we can modify the paradoxical combinator to do useful work. First, let's start with the original Mr. Bad:

```
(Function x -> Not (x x)) (Function x -> Not (x x))
```

The first step to making Mr. Bad do some good is rather than making an unbounded number of `Nots`, `Not (Not (Not (...)))`, let's make an unbounded number of some predefined function `F`:

```
(Function x -> F (x x)) (Function x -> F (x x))
```

Well, that makes unbounded `F (F (F (...)))` but it never terminates so they don't do us any good. The next step is to *freeze* the computation at certain points to directly stop it from continuing; recall from our `freeze` macro above all we need to do is wrap some expression in a `Function _ -> ...` to freeze it:

```
makeFroFs  $\stackrel{\text{def}}{=} \\ (\text{Function } x \rightarrow F (\text{Function } \_ \rightarrow x x)) (\text{Function } x \rightarrow F (\text{Function } \_ \rightarrow x x))$ 
```

---

<sup>5</sup>In this case, anyway, but not in general. General assertions about the equivalence of expressions are hard to prove. In Section 2.4, we will explore a formal means of determining if two expressions are equivalent.



Now, we have postponed the infinite execution; supposing **F** were say

```
Function froF -> True
```

the above would compute to

```
F (Function _ -> makeFroFs)
```

and since the **F** we defined throws away its argument this computation would in turn terminate with value **True**.

This particular example terminated *too* well, it threw away all the copies of **F** we painstakingly made. The way we can get recursion is to make an **F** which sometimes uses its argument **Function \_ -> makeFroFs** to make recursive calls by *thawing* it. Consider the following revised **F**, aiming to ultimately be a function that sums the integers  $\{0, 1, \dots, n\}$  for argument  $n$ :

```
F def == Function froFs -> Function n ->
  If n = 0 Then 0 Else n + froFs 0 (n - 1)
```

If **makeFroFs** were to use this **F**, it would again compute to **F (Function \_ -> makeFroFs)**, but the argument **Function \_ -> makeFroFs** is not thrown out by this new **F**. Consider the computation of **makeFroFs 5**, by the above it is equivalent to computing

```
(F (Function _ -> makeFroFs)) 5
```

and so for the above definition of **F** we see parameter **froFs** will be instantiated with **Function \_ -> makeFroFs**, and parameter **n** with 5. Because  $5 = 0$  is **False** we then compute the else clause which after the above instantiation is

```
5 + (Function _ -> makeFroFs) 0 (5-1)
```

And, to compute the right-hand side of the addition, first the 0 dummy argument is applied to un-freeze **makeFroFs**, so we are now setting to compute **makeFroFs (5-1)**. Look above – this is nearly the exact spot we started at except the argument is one smaller! So, **makeFroFs** is now a recursive summation function and this example will ultimately compute to 15. We have succeeded in repurposing the paradoxical combinator to write recursive functions.

There are a few minor clean-up steps we can perform on the above. First, since the **F** we care about are curried functions of two arguments (we need the additional argument e.g.  $n$  for the recursive call at a different value), we can make a revised freezer **Function n -> F n** which doesn't need to use 0 as the thawer but can “pun” and use the argument itself to do the thawing. So, we can redo the above definitions as

```
makeFs def == (Function x -> F' (Function n -> (x x) n))
(Function x -> F' (Function n -> (x x) n))
```

```
F'  $\stackrel{\text{def}}{=} \text{Function fs } \rightarrow \text{Function n } \rightarrow$ 
  If n = 0 Then 0 Else n + fs (n - 1)
```

– we can remove the 0 argument from the recursive call here since the `n-1` argument is doing the unfreezing work via our pun.

One last refactoring we can do to clean this up is to make a generic recursive-function-maker, by pulling out the `F'` in `makeFs` above as an explicit parameter `f`. This gives us

```
Y  $\stackrel{\text{def}}{=} \text{Function f } \rightarrow$ 
(Function x  $\rightarrow$  f (Function n  $\rightarrow$  (x x) n))
(Function x  $\rightarrow$  f (Function n  $\rightarrow$  (x x) n))
```

and we can apply to some concrete `F`, e.g. `Y F'`, to create our recursive summing function. We call the above expression `Y` because this recursive function creator was discovered by logicians many years ago and given that name.

### Another route to `Y` via direct self-passing

Since the construction of `Y` is complex we now present another path to its construction. This approach is more based on how object-oriented languages such as C++ implement messaging to self: every time an object method is invoked, (a pointer to) the object itself is passed as an additional parameter.

We demonstrate this approach by again writing a *summate* function. This time we follow the C++ idea and write the function to accept *two* arguments: `arg`, which is the number to which we will sum, and `this`, which is the copy of the function we require to be passed in so that recursion may occur. We will name our function `summate0` to reflect the fact that it is not quite the *summate* we want. It can be defined as

```
summate0  $\stackrel{\text{def}}{=} \text{Function this } \rightarrow \text{Function arg } \rightarrow$ 
  If arg = 0 Then 0 Else arg + this this (arg - 1)
```

Note the use of `this this (arg - 1)`. The first use of `this` names the function to be applied; the second use of `this` is one of the arguments to that function. The argument `this` allows the recursive call to invoke the function again, thus allowing us to recurse as much as we need.

We can now sum the integers  $\{0, 1, \dots, 7\}$  with the expression

```
summate0 summate0 7
```

`summate0` always expects its first argument `this` to be itself. It can then use one copy for the recursive call (the first `this`) and pass the other copy on for future duplication. So `summate0 summate0` “primes the pump”, so to speak, by giving the process an initial extra copy of itself.

Better yet, recall that currying allows us to obtain the inner function without applying it. In essence, a function with multiple arguments could be partially evaluated, with some arguments fixed and others waiting for input. We can use this to our advantage to define the `summate` function we want:

```
summate  $\stackrel{\text{def}}{=} \text{summate0 summate0}$ 
```

This allows us to hide the self-passing from the individual using our `summate` function, which cleans things up considerably. We can summarize the entire process as follows using the `Let` macro we described before:

```
summate  $\stackrel{\text{def}}{=} \text{Let sum} = \text{Function this} \rightarrow \text{Function arg} \rightarrow$   

  If arg = 0 Then 0 Else arg + this this (arg - 1)  

  In Function arg -> sum sum arg
```

We now have a model for defining recursive functions without the use of the `Let Rec` operator. This means that untyped languages with no built-in recursion can still be Turing-complete. While this is an accomplishment, we can do even better; we can abstract the idea of self-passing almost entirely out of the body of the function itself.

**The Y-Combinator** The *Y*-combinator is a further abstraction of self-passing. The idea is that the *Y*-combinator does the self-application with an abstract body of code that is passed in as an argument. We first define a function called `almostY`, and then revise that definition to arrive at the real *Y*-combinator.

```
almostY  $\stackrel{\text{def}}{=} \text{Function body} \rightarrow$   

  Function arg -> body body arg
```

using `almostY`, we can define `summate` as follows.

```
summate  $\stackrel{\text{def}}{=} \text{almostY (Function this} \rightarrow \text{Function arg} \rightarrow$   

  If arg = 0 Then 0 Else arg + this this (arg - 1))
```

The true *Y*-combinator actually goes one step further and allows us to write recursive calls in the more natural style of just “`this (arg - 1)`”, avoiding the extra `this` parameter. To do this, we assume that the `body` argument is already in this simple form. We then define a new form, `wrapper`, which replaces `this` with `(this this)` in `body`:

**Definition 2.13** (*Y-Combinator*).

```

comby def = Function body ->
  Let wrapper = Function this -> Function arg -> body (this this) arg
  In Function arg -> wrapper wrapper arg

```

Recalling `Let` is a macro, this is the same `Y` as was defined in the previous subsection.

The transformation steps performed in these examples are also good examples of the power of higher-order functional programming: “code surgery” is performed on `body` to produce `wrapper` by simply using function abstraction and application. The `Y`-combinator can then be used to define `summate` as

```

summate def = comby (Function this -> Function arg ->
  If arg = 0 Then 0 Else arg + this (arg - 1))

```

### 2.3.6 Call-By-Name Parameter Passing

In **call-by-name** parameter passing, the argument to the function is not evaluated at function call time, but rather is only evaluated if it is used. This style of parameter passing is largely of historical interest now; Algol uses it but no modern languages are call-by-name by default (The Digital Mars D language does allow parameters to be treated as call-by-name via use of the `lazy` qualifier). The reason is that it is much harder to write efficient compilers if call-by-name parameter passing is used. Nonetheless, it is worth taking a brief look at call-by-name parameter passing.

Let us define the operational semantics for call-by-name.

$$(\text{Call-By-Name Application}) \quad \frac{e_1 \Rightarrow \text{Function } x \rightarrow e, \quad e[e_2/x] \Rightarrow v}{e_1 \ e_2 \Rightarrow v}$$

Freezing and thawing, defined in Section 2.3.4, is a way to get call-by-name behavior in a call-by-value language. Consider, then, the computation of

```
(Function x -> Thaw x + Thaw x) (Freeze (3 - 2))
```

`(3 - 2)` is not evaluated until we are inside the body of the function where it is thawed, and it is then evaluated two separate times. This is precisely the behavior of call-by-name parameter passing, so `Freeze` and `Thaw` can encode it by this means. The fact that `(3 - 2)` is executed twice shows the main weakness of call by name, namely repeated evaluation of the function argument.

Lazy or **call-by-need** evaluation is a version of call-by-name that caches evaluated function arguments the first time they are evaluated so it doesn’t have to re-evaluate them in subsequent uses. Haskell [14, 7] is a pure functional language with lazy evaluation.

### 2.3.7 F<sup>b</sup> Abstract Syntax

The previous sections thoroughly describe the operational semantics of **F<sup>b</sup>** in terms of its concrete syntax. Recall from our boolean language, however, that an interpreter operates over a representation of a program which is more conducive to programmatic manipulation; this representation is termed its **abstract syntax**. To define the abstract syntax of **F<sup>b</sup>** for a Caml interpreter, we need to define a variant type that captures the expressiveness of **F<sup>b</sup>**. The variant types we will use are as follows.

```

type ident = Ident of string

type expr =
  Var of ident | Function of ident * expr | Appl of expr * expr |
  LetRec of ident * ident * expr * expr |
  Plus of expr * expr | Minus of expr * expr | Equal of expr * expr |
  And of expr * expr | Or of expr * expr | Not of expr |
  If of expr * expr * expr | Int of int | Bool of bool

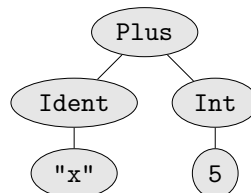
type fbtype = TInt | TBool | TArrow of fbtype * fbtype | TVar of string;;

```

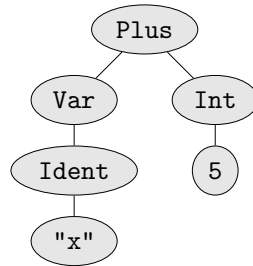
One important point here is the existence of the `ident` type. Notice where `ident` is used in the `expr` type: as variable identifiers, and as function parameters for `Function` and `Let Rec`. The `ident` type attaches additional semantic information to a string, indicating that the string specifically represents an identifier.

Note, though, that `ident` is used in two different ways: to signify the *declaration* of a variable (such as in `Function (Ident "x", ...)`) and to signify the *use* of a variable (such as in `Var (Ident "x")`). Observe that the use of a variable is an expression and so can appear in the AST anywhere that any expression can appear. The declaration of a variable, on the other hand, is not an expression; variables are only declared in functions. In this way, we are able to use the Caml type system to help us keep the properties of the AST nodes straight.

For example, consider the following AST:



At first glance, it might appear that this AST represents the **F<sup>b</sup>** expression `x + 5`. However, the AST above cannot actually exist using the variants we defined above. The `Plus` variation accepts two expressions upon construction and `Ident` is not a variant; thus, the equivalent Caml code `Plus (Ident "x", Int 5)` would not even typecheck. The **F<sup>b</sup>** expression `x + 5` is represented instead by the AST



which is represented by the Caml code `Plus(Var(Ident "x"), Int 5)`.

Being able to convert from abstract to concrete syntax and vice versa is an important skill for one to develop, but it takes some time to become proficient at this conversion. Let us look at some examples  $F_b$  in pursuit of refining this skill.

**Example 2.17.**

**Concrete:**  
1 + 2

**Abstract:**  
Plus(Int 1, Int 2)

```

graph TD
    Plus(Plus) --- Int1(Int)
    Plus --- Int2(Int)
    Int1 --- 1(1)
    Int2 --- 2(2)
    
```

**Example 2.18.**

**Concrete:**  
True Or False

**Abstract:**  
Or(Bool true, Bool false)

```

graph TD
    Or(Or) --- Bool1(Bool)
    Or --- Bool2(Bool)
    Bool1 --- true(true)
    Bool2 --- false(false)
    
```

**Example 2.19.**

**Concrete:**  
If Not(1 = 2) Then 3 Else 4

**Abstract:**  
If(Not(Equal(Int 1, Int 2)),  
Int 3, Int 4)

```

graph TD
    If(If) --- Not(Not)
    If --- Int3(Int)
    If --- Int4(Int)
    Not --- Equal(Equal)
    Equal --- Int1(Int)
    Equal --- Int2(Int)
    Int1 --- 1(1)
    Int2 --- 2(2)
    Int3 --- 3(3)
    Int4 --- 4(4)
    
```

**Example 2.20.**

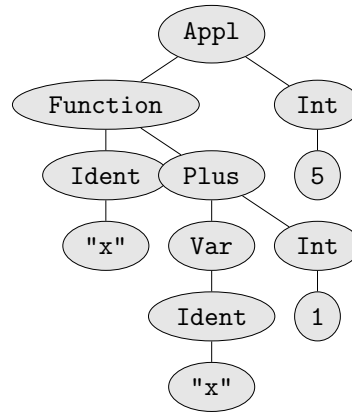
**Concrete:**

(Function x -> x + 1) 5

**Abstract:**

```

Appl(
  Function(
    Ident "x",
    Plus(Var(Ident "x"),
          Int 1)),
  Int 5)
    
```



**Example 2.21.**

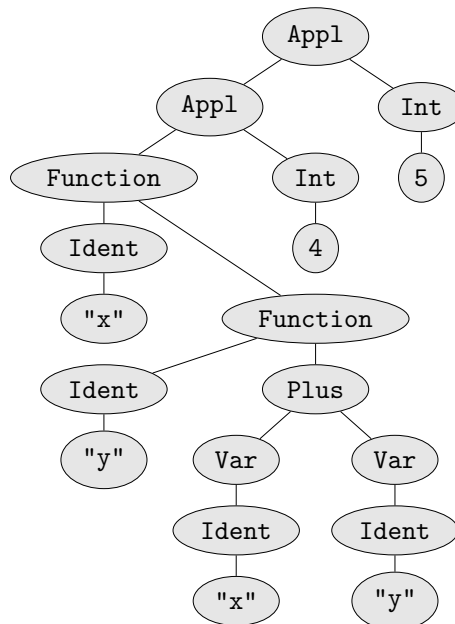
**Concrete:**

(Function x -> Function y ->  
x + y) 4 5

**Abstract:**

```

Appl(
  Appl(
    Function(
      Ident "x",
      Function(
        Ident "y",
        Plus(
          Var(Ident "x"),
          Var(Ident "y")))),
    Int 4),
  Int 5)
    
```



**Example 2.22.****Concrete:**

```

Let Rec fib x =
  If x = 1 Or x = 2 Then 1 Else fib (x - 1) + fib (x - 2)
In fib 6

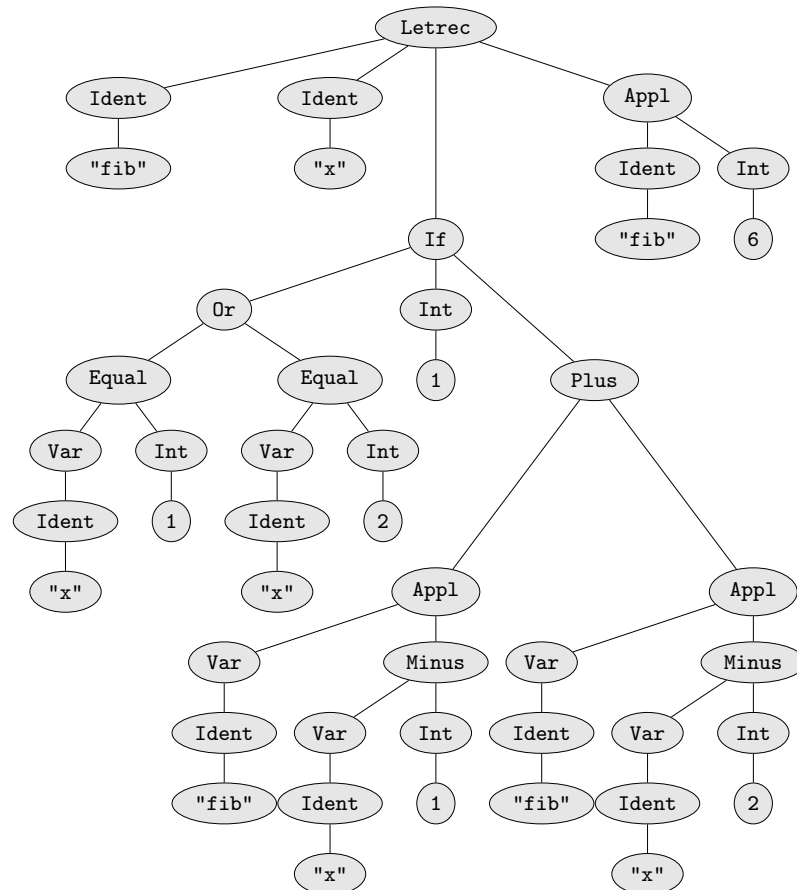
```

**Abstract:**

```

Letrec(Ident "fib", Ident "x", If(Or(Equal(Var(Ident "x"), Int 1),
Equal(Var(Ident "x"), Int 2)), Int 1, Plus(Impl(Var(Ident "fib"),
Minus(Var(Ident "x"), Int 1)), Impl(Var(Ident "fib"), Minus(Var(Ident
"x"), Int 2)))), Impl(Var(Ident "fib"), Int 6))

```



Notice how lengthy even simple expressions can become when represented in the abstract syntax. Review the above examples carefully, and try some additional examples of your own. It is important to be able to comfortably switch between abstract and concrete syntax when writing compilers and interpreters.



## 2.4 Operational Equivalence

One of the most basic operations defined over a space of mathematical objects is the equivalence relation ( $\cong$ ). We have, using operational semantics, defined programs as mathematical entities. As a result, equivalence makes sense for programs too.

First, we are compelled to consider what we mean intuitively by “equivalent” programs. Two things which are equivalent are interchangeable; thus, two equivalent programs must do the same thing. It is not necessary that they do so in exactly the same way however. For example, we can easily see that the programs  $(1 + 1 + 1 - 1)$  and  $(2)$  are equivalent, but the prior requires more computation to evaluate.

Because two equivalent programs can be substituted for each other but may have different execution-time properties, we can use operational equivalence when optimizing compilers and other such tools. Operational equivalence provides a rigorous and reliable foundation for such work; we do not need to worry about an optimization changing the behavior of an application because we can prove mathematically that such a change is impossible. Operational equivalence also defines the process of refactoring, a process by which developers can change the structure of a program for maintainability or enhancement without changing the program’s behavior.

Defining an equivalence relation for programs is actually not as straightforward as one might expect. The initial idea is to define the relation such that two programs are equivalent if they always lead to the same results when used. As we will see, however, this definition is not sufficient, and we will need to do some work to arrive at a satisfactory definition.

Let us begin by looking at a few sample equivalences to get a feel for what they are.  $\eta$ -conversion (or *eta*-conversion) is one example of an interesting equivalence. It is defined as follows.

```
Function  $x \rightarrow e \cong$ 
  Function  $z \rightarrow (\text{Function } x \rightarrow e) z$ , for  $z$  not free in  $e$ 
```

$\eta$ -conversion is similar to the proxy pattern in object oriented programming[12]. A closely related law for our freeze/thaw syntax is

```
Thaw (Freeze  $e$ )  $\cong e$ 
```

In both examples, one of the expressions may be replaced by the other without ill effects (besides perhaps changing execution time), so we say they are equivalent. To write formal proofs, however, we will need to develop a more rigorous definition of equivalence.

### 2.4.1 Defining Operational Equivalence

Let’s begin by informally strengthening our definition of operational equivalence. We define equivalence in a manner dating all the way back to Leibniz[19]:

**Definition 2.14** (Operational Equivalence (Informal)). *Two programs expressions are equivalent if and only if one can be replaced with the other at any place, and no external change in behavior will be noticed.*

We wish to study equivalence for possibly open programs, because there are good equivalences such as  $x + 1 - 1 \cong x + 0$ . We define “at any place” by the notion of a **program context**, which is, informally, a **Fb** program with some holes ( $\bullet$ ) in it. Using this informal definition, testing if  $e_1 \cong e_2$  would be roughly equivalent to performing the following steps (for all possible programs and all possible holes, of course).

1. Select any program context (that is, program containing a hole).
2. Place  $e_1$  in the  $\bullet$  position and run the program.
3. Do the same for  $e_2$ .
4. If the observable result is different,  $e_1$  is not equivalent to  $e_2$ .
5. Repeat steps 1-4 for *every possible context*. If none of these infinitely many contexts produces different results, then  $e_1$  is equivalent to  $e_2$ .

Now let us elaborate on the notion of a program context. Take an **Fb** program with some “holes” ( $\bullet$ ) punched in it: replace some subterms of any expression with  $\bullet$ . Then “hole-filling” in this program context  $C$ , written  $C[e]$ , means replacing  $\bullet$  with  $e$  in  $C$ . Hole filling is like substitution, but without the concerns of bound or free variables. It is direct replacement with no conditions.

Let us look at an example of contexts and hole-filling using  $\eta$ -conversion as we defined above. Let

$$C \stackrel{\text{def}}{=} (\text{Function } z \text{ -> Function } x \text{ -> } \bullet) z$$

Now, filling the hole with  $x + 2$  is simply written

$$\begin{aligned} ((\text{Function } z \text{ -> Function } x \text{ -> } \bullet) z)[x + 2] = \\ (\text{Function } z \text{ -> Function } x \text{ -> } x + 2) z \end{aligned}$$

Finally, we are ready to rigorously define operational equivalence.

**Definition 2.15** (Operational Equivalence).  *$e \cong e'$  if and only if for all contexts  $C$ ,  $C[e] \Rightarrow v$  for some  $v$  if and only if  $C[e'] \Rightarrow v'$  for some  $v'$ .*

Another way to phrase this definition is that two expressions are equivalent if in any possible context,  $C$ , one terminates if the other does. We call this *operational equivalence* because it is based on the interpreter for the language, or rather it is based on the operational semantics. The most interesting, and perhaps nonintuitive, part of this definition is that nothing is said about the relationship between  $v$  and  $v'$ . In fact,

they may be different in theory. However, intuition tells us that  $v$  and  $v'$  must be very similar, since equivalence holds for any possible context.

The only problem with this definition of equivalence is its “incestuous” nature—there is no absolute standard of equivalence removed from the language. **Domain theory** is a mathematical discipline which defines an algebra of programs in terms of existing mathematical objects (complete and continuous partial orders). We are not going to discuss domain theory here, mainly because it does not generalize well to programming languages with side effects. [17] explores the relationship between operational semantics and domain theory.

### 2.4.2 Properties of Operational Equivalence

In this section, we present some general equivalence principles for  $\mathbf{F}^b$ .

**Definition 2.16** (Reflexivity).

$$e \cong e$$

**Definition 2.17** (Symmetry).

$$e \cong e' \text{ if } e' \cong e$$

**Definition 2.18** (Transitivity).

$$e \cong e'' \text{ if } e \cong e' \text{ and } e' \cong e''$$

**Definition 2.19** (Congruence).

$$C[e] \cong C[e'] \text{ if } e \cong e'$$

**Definition 2.20** ( $\beta$ -Equivalence).

$$(\text{Function } x \rightarrow e) v \cong (e[v/x])$$

*provided  $v$  is closed (if  $v$  had free variables they could be captured when  $v$  is placed deep inside  $e$ ).*

**Definition 2.21** ( $\eta$ -Equivalence).

$$(\text{Function } x \rightarrow e) \cong ((\text{Function } z \rightarrow \text{Function } x \rightarrow e) z)$$

**Definition 2.22** ( $\alpha$ -Equivalence).

$$(\text{Function } x \rightarrow e) \cong ((\text{Function } y \rightarrow e)\{y/x\})$$

**Definition 2.23.**

$$(n + n') \cong \text{the sum of } n \text{ and } n'$$

*Similar rules hold for  $\neg$ , **And**, **Or**, **Not**, and  $=$ .*

**Definition 2.24.**

(If True Then  $e$  Else  $e'$ )  $\cong e$

A similar rule holds for If False...

**Definition 2.25.**

If  $e \Rightarrow v$  then  $e \cong v$

Equivalence transformations on programs can be used to justify results of computations instead of directly computing with the interpreter; it is often easier. An important component of compiler optimization is applying transformations, such as the ones above, that preserve equivalence.

**2.4.3 Examples of Operational Equivalence**

To solidify the concept of operational equivalence (one which reliably boggles newcomers to programming language theory), we provide a number of examples of equivalent and non-equivalent expressions. We start with a very simple example of two expressions which are not equivalent.

**Lemma 2.5.**  $2 \not\cong 3$ 

*Proof.* By example. Let  $C \stackrel{\text{def}}{=} \text{If } \bullet = 2 \text{ Then } 0 \text{ Else } (0 \ 0)$ .  $C[2] \Rightarrow 0$  while  $C[3] \not\Rightarrow v$  for any  $v$ . Thus, by definition,  $2 \not\cong 3$ .  $\square$

Note that, in the above proof, we used the expression  $(0 \ 0)$ . This expression cannot evaluate; the application rule applies only to functions. As a result, this expression makes an excellent tool for intentionally making code get stuck when building a proof about operational equivalence. Other similar get-stuck expressions exist, such as  $\text{True} + \text{True}$ ,  $\text{Not } 5$ , and the ever-popular  $(\text{Function } x \rightarrow x \ x)(\text{Function } x \rightarrow x \ x)$ .

It should be clear that we can use the approach in Lemma 2.5 to prove that any two integers which are not equal are not operationally equivalent. But can we make a statement about a non-value expression?

**Lemma 2.6.**  $x \not\cong x + 1 - 1$ .

At first glance, this inequivalence may seem counterintuitive. But the proof is fairly simple:

*Proof.* By example. Let  $C \stackrel{\text{def}}{=} (\text{Function } x \rightarrow \bullet) \text{ True}$ . Then  $C[x] \Rightarrow \text{True}$ .  $C[x + 1 - 1] \equiv (\text{Function } x \rightarrow x + 1 - 1) \text{ True}$ , which cannot evaluate because no rule allows us to evaluate  $\text{True} + 1$ . Thus, by definition,  $x \not\cong x + 1 - 1$ .  $\square$

Similar proofs could be used to prove inequivalences such as  $\text{Not}(\text{Not}(e)) \not\cong e$ . The key here is that some of the rules in  $\mathbf{Fb}$  distinguish between integer values and boolean values. As a result, equivalences cannot hold if one side makes assumptions about the type of value to which it will evaluate and the other side does not.

We have proven inequivalences; can we prove an equivalence? It turns out that some equivalences can be proven but that this process is much more involved. Thus far, our proofs of inequivalence have relied on the fact that they merely need to demonstrate an example context in which the expressions produce different results. A proof of equivalence, however, must demonstrate that no such example exists among infinitely many contexts. For example, consider the following:

**Lemma 2.7.** *If  $e \not\Rightarrow v$  for any  $v$ ,  $e' \not\Rightarrow v'$  for any  $v'$ , and both  $e$  and  $e'$  are closed expressions, then  $e \cong e'$ . For example,  $(0\ 0) \cong (\text{Function } x \rightarrow x\ x)(\text{Function } x \rightarrow x\ x)$ .*

First, we need a definition:

**Definition 2.26** (Touch). *An expression is said to **touch** one of its subexpressions if the evaluation of the expression causes the evaluation of that subexpression.*

That is, `If True Then 0 Else 1` *touches* the subexpression 0 because it is evaluated when the whole expression is evaluated. 1 is not touched because the evaluation of the expression never causes 1 to be evaluated.

We can now prove Lemma 2.7.

*Proof.* By contradiction. Without loss of generalization, suppose that there is some context  $C$  for which  $C[e] \Rightarrow v$  while  $C[e'] \not\Rightarrow v'$  for any  $v'$ .

Because  $e$  is a closed expression,  $C'[e] \not\Rightarrow v$  for all  $v$  and all contexts  $C'$  which touch  $e$ . Because  $C[e] \Rightarrow v$ , we know that  $C$  does not touch the hole.

Because  $C$  does not touch the hole, how the hole evaluates cannot affect the evaluation of  $C$ ; that is,  $C[e^*] \Rightarrow v^*$  for some  $v^*$  must be true for all  $e^*$  or for no  $e^*$ .

Because  $C[e] \Rightarrow v$ ,  $C[e^*] \Rightarrow v^*$  for some  $v^*$  for all  $e^*$ . But  $C[e'] \not\Rightarrow v'$  for any  $v'$ . Thus, by contradiction,  $C$  does not exist. Thus, by definition,  $e \cong e'$ .  $\square$

The above proof is much longer and more complex than any of the proofs of inequivalence that we have encountered and it isn't even very robust. It could benefit, for example, from a proof that a closed expression cannot be changed by a replacement rule (which is taken for granted above). Furthermore, the above proof doesn't even prove a very useful fact! Of far more interest would be proving the operational equivalence of two expressions when one of them is executed in a more desirable manner than the other (faster, fewer resources, etc.) in order to motivate compiler optimizations.

Lemma 2.7 is, however, effective in demonstrating the complexities involved in proving operational equivalence. It is surprisingly difficult to prove that an equivalence holds; even proving  $1 + 1 \cong 2$  is quite challenging. See [17] for more information on this topic.

#### 2.4.4 The $\lambda$ -Calculus

We briefly consider the  $\lambda$ -calculus. In Section 2.3, we saw how to encode tuples, lists, `Let` statements, freezing and thawing, and even recursion in `Fb`. The encoding approach is very powerful, and also gives us a way to understand complex languages based on our understanding of simpler ones. Even numbers, booleans, and if-then-else statements are encodable, although we will skip these topics. Thus, all that is needed is functions

and application to make a Turing-complete programming language. This language is known as the **pure  $\lambda$  calculus**, because functions are usually written as  $\lambda x.e$  instead of **Function**  $x \rightarrow e$ .

Execution in  $\lambda$  calculus is extremely straightforward and concise. The main points are as follows.

- Even programs with free variables can execute (or *reduce* in  $\lambda$ -calculus terminology).
- Reduction can happen anywhere, e.g. inside a function body that hasn't been called yet.
- $(\lambda x.e)e' \Rightarrow e[e'/x]$  is the only reduction rule, called  $\beta$ -reduction. (It has a special side-condition that it must be *capture-free*, i.e. no free variables in  $e'$  become bound in the result. Capture is one of the complications of allowing reduction anywhere.)

This form of computation is conceptually very interesting, but is more distant from how actual computer languages execute and so we do not put a strong focus on it here.

## Exercises

**Exercise 2.1.** How would you change the Sheep language to allow the terms *bah*, *baah*,  $\dots$  without excluding any terms which are already allowed?

**Exercise 2.2.** Is the term *it* in the Frog language? Why or why not?

**Exercise 2.3.** Is it possible to construct a term in Frog without using the terminal *t*? If so, give an example. If not, why not?

**Exercise 2.4.** Complete the definition of the operational semantics for the boolean language described in section 2.2.1 by writing the rules for **Or** and **Implies**.

**Exercise 2.5.** Why not just use interpreters and forget about the operational semantics approach?