Chapter 4

Definelang: A Language with Global Variables

Our goal in this chapter is to learn about global definitions and usage, and to contrast them with locally scoped definitions discussed in the previous chapter.

4.1 Local vs. Global Definitions

So far each variable that we have defined in a let expression has had a local lexical scope. So in the let expression `(let ((a 3)(b 4)(c 2)) (+ a b c))`, variables a, b, and c are defined only during the evalution of `(+ a b c)` and not outside it. Sometimes it is useful to define a variable and have it available for the entire duration of our interaction with the interpreter. For example, we can define the roman numerals using such a feature.

```
$ (define i 1)
```

We will call this feature a *define declaration*. A define declaration has three parts: the `define` keyword, name that is being defined (here `i`), and the initial value given to the name (here `1`). We can similarly define other roman numerals.

```
$ (define ii 2)
$ (define iii 3)
$ (define iv 4)
$ (define v 5)
```

We can then use these definitions in all of our programs as shown below.

```
$ (+ (* iii iv v v) (* ii iv v) ii)
342
$ (- (* v v v v) (+ i ii iii iv) (* iii v v))
540
$ (+ (* v ii iv) i (* v v iv v))
541
$ (+ (+ v v v i) (* v v v v))
641
```

Note that this is different from extending the initial environment of programs in Varlang with predefined values. Those definitions would have effect on each run of the interpreter, whereas names defined by the `define` form during a particular run of the interpreter would not be automatically valid for the next run of the same interpreter.

4.2 Define, define

To support the define form, we extend the syntax of Varlang as shown in figure 4.1 to create a new language that we will call *Definelang*.

There are two major syntax changes. First, the syntax of the program changes in this language. A program in Definelang consists of zero or more definitions, represented as `DefineDecl*` followed by an optional expression, represented as `Exp?`. We have also added the syntax for define declarations. The syntax of `DefineDecl` takes an identifier (name being defined) and an expression (value of the name being defined). Here are some additional examples.

```
$ (define R 8.3145) // The gas constant R
$ (define n 2) // 2 moles of gas
$ (define V 0.0224) // Volume of gas 0.0224 m^2
$ (define T 273) // Temperature of gas 273 K
$ (define P (/ (* n R T) V)) // Using Boyles law to compute pressure
$ P     //What is the pressure?
202665.93750000003
```

Notice that the expression in a define declaration can make use of previously defined names, but it cannot utilize those that might appear later.

`Program`	::=	`DefineDecl`* `Exp?`	*Program*
`DefineDecl`	::=	`(define Identifier Exp)`	*Define*
`Exp`	::=		*Expressions*
		`Number`	*NumExp*
	|	`(+ Exp Exp`$^+$`)`	*AddExp*
	|	`(- Exp Exp`$^+$`)`	*SubExp*
	|	`(* Exp Exp`$^+$`)`	*MultExp*
	|	`(/ Exp Exp`$^+$`)`	*DivExp*
	|	`Identifier`	*VarExp*
	|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
`Number`	::=	`Digit`	*Number*
	|	`DigitNotZero Digit`$^+$	
`Digit`	::=	`[0-9]`	*Digits*
`DigitNotZero`	::=	`[1-9]`	*Non-zero Digits*
`Identifier`	::=	`Letter LetterOrDigit`*	*Identifier*
`Letter`	::=	`[a-zA-Z$_]`	*Letter*
`LetterOrDigit`	::=	`[a-zA-Z0-9$_]`	*LetterOrDigit*

Figure 4.1: Grammar for the Definelang Language

Each line in the previous example is a separate `Definelang` program according to the syntax in figure 4.1. Each of these programs defines a single variable, but it is possible to define multiple variables at once, e.g. both Faraday and Rydberg constant[1].

```
$ (define F 96454.56) (define R 10973731.6)
```

The Definelang language also permits defining one or more constants and then computing the value of an expression.

```
$ (define R 8.3145) (/ (* 2 R 273) 0.0224)
202665.93750000003
```

We will now explore the semantics and implementation of this new language feature, and changes in the semantics of programs to add define declarations.

[1]For the curious reader, Faraday's constant is the charge on a mole of electrons.

Extending the AST, the visitor, and the formatter

Besides the grammar, we would also need to extend the AST, and the visitor infrastructure to accomodate these language changes. These changes include adding a new AST node `DefineDecl`, modifying the AST node for `Program` to also store define declarations, modifying the visitor interface to support new AST node, and finally modifying the formatter to print the new AST node. The reader is encouraged to review these changes in the companion code before proceeding further.

4.3 Semantics and Interpretation of Programs with Define Declarations

A correct Varlang program cannot have any free variables. A Definelang program can have free variables. For example, after starting the interpreter if a Definelang programmer types the following program the interpreter will evaluate the program to a dynamic error.

```
$ (/ (* 2 R 273) 0.0224)
No binding found for name: R
```

On the other hand, if the interaction of the programmer was like the following the same program would produce the anticipated value.

```
$ (define R 8.3145)
unit
$ (/ (* 2 R 273) 0.0224)
202665.93750000003
```

To understand the difference between these two behaviors, it would help to ask: when both of these programs start running, what was the value of the environment? In Varlang, when a program starts running, no variables have been defined yet. If you would recall from the realization of Varlang, we started evaluating every program in an empty environment. The semantics of Definelang is slightly different — when an interpreter starts running, no variables have been defined yet; when a program starts running all variables that have been declared since the interpreter started running are defined. To a reader familiar with interpreters or similar systems, this distinction will not come as a surprise, but it is important to make a note of it as we develop the realization of Definelang.

Recall from previous chapters that in our interpeter, a single `Evaluator` object persists through the lifetime of the interpreter. In Definelang, we want definitions to be retained across runs, which can be achieved by making the initial environment an attribute of the `Evaluator` object. Following implementation changes model that.

```
1 class Evaluator implements Visitor<Value> {
2   Env initEnv = new EmptyEnv(); //New for definelang

4   Value valueOf(Program p) {
5     return (Value) p.accept(this, initEnv);
6   }

8   public Value visit(Program p, Env env) {
9     for(DefineDecl d: p.decls())
10      d.accept(this, initEnv);
11    return (Value) p.e().accept(this, initEnv);
12  }
13  ...
14 }
```

The set of legal values for Varlang was limited to `NumVal`, however, for Definelang this set needs to be extended as shown in figure 4.2 to model the semantics that define declarations do not produce any values.

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`UnitVal`	*Unit Values*
`NumVal`	::=	`(NumVal n)`, where `n` ∈ the set of doubles	*NumVal*
`UnitVal`	::=	`(UnitVal)`	*NumVal*

Figure 4.2: The Set of Legal Values for the Definelang Language

The figure defines a new kind of value `UnitVal`, for unit values. A `UnitVal` is like a `void` type in Java. It allows programming language definitions and implementations to uniformly treat programs and expressions as evaluating to 'a value', which could be a `UnitVal` when producing other kinds of value is not sensible.

The semantics of define declarations is very similar to the let expressions, except that each definition changes the global `initEnv` to add a new binding from name to value.

```
1  public Value visit (DefineDecl d, Env env) { // New for definelang.
2    String name = d.name();
3    Exp value_exp = d.value_exp();
4    Value value = (Value) value_exp.accept(this, env);
5    initEnv = new ExtendEnv(initEnv, name, value);
6    return new Value.UnitVal();
7  }
```

The implementation evaluates the value expression on line 4, creates a new environment by extending the current value of `initEnv` on line 5. The value of a define declaration is a `UnitVal`.

Exercise

4.3.1. *[Volume of Sphere]* Define a constant `pi` with the usual value of `3.14159265359`. Define a constant `fourByThree` with the value of `1.33333`. Using the definition of `pi` and `fourByThree`, compute the volume of a sphere with radius `1.42`. Recall that the volume of a sphere is `4/3 * pi * radius * radius * radius`.

4.3.2. *[Lazy Definitions]* Extend the Definelang programming language from this chapter such that it supports a variation of the define declaration, say `ldefine` declaration (for lazy define). In a regular define declaration in an expression such as the following, first the value of the expression `(/ (* n R T) V)` is computed, then the global environment is extended with a mapping from name `P` to this value.

```
$ (define P (/ (* n R T) V))
```

In a lazy define declaration, the value of the expression `(/ (* n R T) V)` will be computed when the name `P` is used. An example appears below.

```
$ (define P (/ (* n R T) V))
$ (define R 8.3145) // The gas constant R
$ (define n 2) // 2 moles of gas
$ (define V 0.0224) // Volume of gas 0.0224 m^2
$ (define T 273) // Temperature of gas 273 K
$ P      //What is the pressure?
202665.93750000003
```

Notice that the interaction above would not have worked with the regular define declaration because the names `n`, `R`, `T`, and `V` would not be defined. In the lazy define declaration, since the value of the expression `(/ (* n R T) V)` is computed when `P` is looked up the interaction works.

4.3.3. *[Macros]* Extend the Definelang programming language from this chapter such that it supports declaring macros and macro expansion.

A macro definition has the following form.

```
$ (define (macro_name argument1, argument2, ...)  expression )
```

Example:

```
$ (define (square x)  (* x x) )
$ (square 2) // This is expanded to (* 2 2)
4

$ (define ( pressure  n R T V)  (/ (* n R T) V))
$ ( pressure  2 8.3145 273 0.0224 )
202665.93750000003
```

4.4 Further Reading

Template Metaprogramming

The macros described in the exercise were a simplified form of a more general idea known as template metaprogramming. Template metaprogramming, broadly, refers to a collection of technique that consume user facing syntax to generate more source code at compile-time. Template metaprogramming is utilized by a number of programming languages most notably C++, but also in Haskell and Curl.

Chapter 5

Funclang: A Language with Functions

Our goal in this chapter is to learn about functions in full detail. We will especially focus on first-class functions, lambda abstraction, closures, higher-order functions, currying, and on functional data structures. We will also learn about the essence of pairs and lists and about techniques for flat recursion.

5.1 Function as Abstraction

We have already discussed variables, the most elementary abstraction mechanism in computer programming languages, in previous chapters. A variable name is a proxy for a piece of computation. For example, the variables `x` and `y` below stand for the definitions on the right.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad y = \frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'}$$

which allows us to write $x * (y - x)$ instead of the following more complex form.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} * (\frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'} - \frac{-b \pm \sqrt{b^2 - 4ac}}{2a})$$

We can think of these variables as an opaque, fixed abstraction, once we define them we cannot customize their functionalities. For some abstractions it might be desirable to alter some portion of its functionality.

For example, for variable definitions `x` and `y` above it may be sensible to change values of `a`, `b`, and `c` and get a different instantiation of the abstraction. Functions, procedures, and methods in computer programming languages provide such an abstraction. This ability to instantiate an abstraction by assigning concrete values in place of formal parameters is known as *parametrization*.

There are many variations of programming language features that can be used for parameterization of a computation. Each such feature is a variation of two elementary concepts: the ability to define a procedure, and the ability to call a procedure. In this chapter, we first focus on these concepts and then study some variations. We will gradually define a programming language with these features that we will call *Funclang*.

5.2 Function Definition and Calls

We will start by studying two features: lambda abstraction, and call. Think of a lambda abstraction as a tool for defining anonymous function. Let us start with simple examples of such feature. For example, the listing below defines an anonymous function that takes a single parameter `x` and the value of the function is `x`. It may be useful for you to try out these examples using the interpreter for this chapter.

```
(
  lambda      //Lambda special function for defining functions
  (x)         //List of formal parameter names of the function
   x          //Body of the function
)
```

If you are familiar with the notion of procedure or methods in ALGOL family of languages like C, C++, Java, C#, etc..., you should notice three key differences in syntax and one key difference in semantics. Key differences in syntax are:

1. We are not specifying the name of the function.

2. The formal parameter name is neither preceded nor followed by the type of the formal parameter.

3. We don't have to write an explicit "return" statement to specify the value returned by the function. In the function definition above, the value of the formal parameter `x` is the return value of the function.

In ALGOL family of languages as well as in lower-level languages like assembly language, procedures and methods are thought of as a subsection of the code section, procedure/method names are thought of as a proxy for the location of that subsection in the code section, and procedure/method call as a jump to that location after adjusting the environment. Unlike this mental model, it is perhaps best to think of a lambda abstraction as a generator of runtime values that represent functions. A lambda abstraction can be used to create many such function values, and each of these function values have a different identity. Each such function value can be used multiple times.

The following listing shows another anonymous function that takes a single parameter `x` and the value of the function is `x + 1`.

```
(
 lambda          //Lambda special form for defining functions
 (x)             //List of formal parameter names of the function
 (+ x 1)         //Body of the function
)
```

The listing belows defines an anonymous function that takes two parameters `x`, `y` and the value of the function is `x + y`.

```
(lambda (x y) (+ x y))
```

We can call an anonymous function also. Here, is an example of calling the identity function.

```
(                   //Begin function call syntax
  (lambda (x) x)    //Operator: function being called
  1                 //Operands: list of actual parameters
)                   //End function call syntax
```

The value of this program is `1`. Notice the prefix notation for function calls. The operator here is the function `(lambda (x) x)`. Here, is another example of calling the addition function.

```
(                       //Begin function call syntax
  (lambda (x) (+ x 1))  //Operator: function being called
```

```
  1                    //Operands: list of actual parameters
)                    //End function call syntax
```

The value of this program is `2`. The operator here is the function `(lambda (x) (+ x 1))`. Here, is a third example of calling the function that adds two numbers. The value of this program is `2`.

```
(
  (lambda (x y) (+ x y))
  1 1
)
```

If we desire, we can also give these lambda abstractions lexically scoped name using the `let` expression. For example, we can give the name identity to the lambda abstraction that we defined previously.

```
(let
  (( identity  (lambda (x) x)))          //Naming the function
  ( identity  1)                         //Function call
)
```

We can also give these functions globally scoped name using the `define` declaration.

```
$ (define identity (lambda (x) x))
$ ( identity  1)
1
```

As is usual, functions can be defined in terms of other helper functions.

```
$ (define square (lambda (x) (* x x)))
$ (square 1.2)
1.44
$ (define cube (lambda (x) (* (square x) x)))
$ (cube 1.2)
1.728
```

Exercise

5.2.1. *[Area]* Using the `define` declaration, define a constant `pi` with the usual value of `3.14159265359`. Use the definition of `pi` to define a function `area` of a circle that takes a radius and computes the area using the standard formula `pi * radius * radius`.

5.2.2. *[sumsquares]* Define a function sumsquares that takes two integers as a parameter and computes the sum of square of numbers from the first number to the second number.

```
$ (sumsquares 1 2)
5
$ (sumsquares 3 5)
50
```

5.2.3. *[sumseries]* Define a function `sumseries` that takes a number n as argument and computes the series below. Take the value of `(sumseries 0)` to be 0.

$$(1/2) - (1/4) + (1/8) - (1/16) + ...$$

5.3 Functions for Pairs and Lists

Most programming languages provide some built-in functions, i.e. functions that programmers do not have to define from scratch and can assume to be available. In Funclang, we have several such built-in functions, mostly related to list manipulation. Funclang programmers have access to `list`, `car`, `cdr`, `cons`, and `null?` built-in functions. These functions `car`, `cdr`, `cons`, and `null?` operate on both pairs and lists in Funclang.

A pair in Funclang is a 2-tuple written as `(fst.snd)`. A list in Funclang is a pair, where the second element is a list. There is a special list, *empty list*, which is not a pair. Not all pairs are lists. Each list except for the empty list is a pair.

The function `list` is a constructor for list and it takes zero or more values as arguments, and produces a list containing these values. The function `car` takes a single argument, a pair or a list, and produces the first element of that pair or list. The function `cdr` also takes a single argument, a pair or a list, and produces the second element of that pair or list. The function `cons` takes two values as arguments. If the second value is a list, it produces a new list with the first value appended to the front of the second value list. Otherwise, it produces a pair of two argument values. The function `null?` takes a single argument and evaluates to `#t` if that argument is an empty list. Exercises in this section will help you become more familiar with the semantics of these functions.

Using these basic functions, other functions over lists can also be defined. For example, the function `cadr` defined below returns the second element of the argument list.

```
(define cadr
  (lambda (lst)
    (car (cdr lst))
  )
)
```

Similarly the function `caddr` defined below produces the third element of the argument list.

```
(define caddr
  (lambda (lst)
    (car (cdr (cdr lst)))
  )
)
```

The `length` function below computes the size of the list.

```
(define length
  (lambda (lst)
    (if (null? lst) 0
      (+ 1 (length (cdr lst)))
    )
  )
)
```

The `length` function is an excellent first example of a recursive function over list. *A recursive function's definition should mirror the definition of the input data types.* For example, a list can be thought of as follows.

$$List := (list) \mid (cons\ \texttt{val}\ List),\ \texttt{where val} \in \texttt{Value}$$

The definition says that a list is either an empty list, or a pair constructed by joining an element and another list. Notice that the function `length` is structured just like the definition of list into two cases. First case handles empty lists. Second case handles non-empty list.

The `append` function below combines two lists.

```
(define append
  (lambda (lst1 lst2)
```

```
    (if (null? lst1) lst2
      (if (null? lst2) lst1
        (cons (car lst1) (append (cdr lst1) lst2))
      )
    )
  )
)
```

Notice that for `append` the definition of `list` suggests that the function's structure should have four cases.

1. `lst1` is empty, `lst2` is empty.
2. `lst1` is empty, `lst2` is not empty.
3. `lst1` is not empty, `lst2` is empty.
4. `lst1` is not empty, `lst2` is not empty.

On closer observation, we realize that the value of the function `append` in the first and second case will be the same, so to optimize function definition we merge these two cases. Thus, we arrive at a suggested recursive function structure with three cases. In summary, when writing recursive functions that process data types such as lists, the structure of the data type provides useful hint for creating the structure of the function.

Exercise

5.3.1. Experiment with built-in functions for lists.

1. Use (list) to create an empty list,
2. Use (list 342) to create a list with a single element 342.
3. Use (car (list 342)) to get the first element of the previously defined list.
4. Use (cdr (list 342)) to get the rest of the elements of the previously defined list.
5. Use (null? (list)) to check if the list created in part 1 is an empty list.

6. Use (cons 541 (list 342)) to append an element at the beginning of the list created in part 2.
7. Use (cadr (list 541 342)) to get second element of the list.
8. Use (caddr (list 641 541 342)) to get third element of the list.
9. Use function length to find length of the list created by expression (list).
10. Use function append to concatenate a list with a single element 3, with another list with a single element 4.

5.3.2. *[Sum of Even Numbers]* Write a function, sumeven, that takes a list of numbers and returns the summation of the even numbers in this list. For example

```
$ (sumeven (list 1 1 1 1 1))
0
$ (sumeven (list 1 1 1 1 2))
2
```

5.3.3. *[Frequency]* Write a function, frequency, which takes a list, `lst`, and an element, `elem`, and returns the frequency of that element in that list. For example:

```
$ (frequency ( list ) 5)
0
$ (frequency ( list #t "hello") 5)
0
$(frequency ( list #t 5 "hello") 5)
1
$(frequency ( list #t 5 "hello" 5) 5)
2
```

5.3.4. *[Reverse]* Write a function, reverse, which takes a list, `lst` and returns the reverse of that list. For example:

```
$( reverse ( list ))
()
$( reverse ( list 3))
(3)
$( reverse ( list 3 4))
```

```
(4 3)
$( reverse ( list 3 4 2))
(2 4 3)
```

5.3.5. *[Sum of 2^n]* Write a recursive function called `sumPower`, that takes a list of numbers, and computes the sum of 2 to the power of each element in the list.

The following interactions log illustrates semantics of `sumPower`:

```
$ (sumPower (list))
0
$ (sumPower (list 0))
1
$ (sumPower (list 1))
2
$ (sumPower (list 3))
8
$ (sumPower (list 1 3))
10
```

5.3.6. *[Get books]* Write a function, getbooks that takes in a list of lists with author/book string pairs and returns a single list of only the books. Assume the author is the first element and the book is the second

```
$ (get-books (list (list "C. S. Lewis" "The Last Battle")
                   (list "Charles Dickens" "A Christmas Carol")
                   (list "Arthur C. Clarke" "Rama")))
("The Last Battle" "A Christmas Carol" "Rama")
```

5.3.7. *[Triangle]* Write a function, triangle, which takes a number and produces a list each element of which is a list of symbols.

When triangle is called with a non-negative integer, n, it returns a list containing n number of lists. The first inner list has n elements, the second inner list has n-1 element, and so on until you reach the top with only one element list, which forms the shape of a triangle. Each of the inner lists contain only the numbers 0 and 1 and they alternate across lists. The result always has the 0 as the first element of the first inner list, if any.

In the following examples, we have formatted the output to show the result more clearly, but your output will not look the same; it is sufficient to just get the outputs that are equal to those shown. Spaces in the lists are just for displaying purposes and you are not required to print them.

```
$ ( triangle 0)
  ()

$ ( triangle 1)
  ((0))

$ ( triangle 2)
  ((0 1)
    (1))

$ ( triangle 3)
    ((0 1 0)
      (1 0)
       (0))

$ ( triangle 4)
    ((0 1 0 1)
      (1 0 1)
       (0 1)
        (1))

$ ( triangle 5)
    ((0 1 0 1 0)
      (1 0 1 0)
       (0 1 0)
        (1 0)
         (0))

$ ( triangle 6)
    ((0 1 0 1 0 1)
      (1 0 1 0 1)
       (0 1 0 1)
```

```
(1 0 1)
 (0 1)
  (1))
```

5.3.8. *[Board]* Write a procedure, board, which takes a integer number as input and produces a list as an output, each element of which is a list of 0s and 1s.

When board is called with a non-negative integer, n, it returns a list containing n lists each of which have n elements. These lists form the shape of a square board. Each of the inner lists contain only numbers 0 and 1 and they alternate across lists. The result always has 0 as the first element of the first inner list, if any.

The following examples, shows a formatted output of the board, but your output does not need look the same regarding spaces. For your output it is sufficient to just produce the list. Spaces in the lists are just for displaying purposes and you are not required to print them.

```
$ (board 0)
()

$ (board 1)
((0))

$ (board 2)
((0 1)
(1 0))

$ (board 3)
((0 1 0)
 (1 0 1)
 (0 1 0))

$ (board 4)
((0 1 0 1)
 (1 0 1 0)
 (0 1 0 1)
 (1 0 1 0))
```

```
$ (board 5)
((0 1 0 1 0)
 (1 0 1 0 1)
 (0 1 0 1 0)
 (1 0 1 0 1)
 (0 1 0 1 0))

$ (board 6)
((0 1 0 1 0 1)
 (1 0 1 0 1 0)
 (0 1 0 1 0 1)
 (1 0 1 0 1 0)
 (0 1 0 1 0 1)
 (1 0 1 0 1 0))
```

5.4 Higher-order Functions

A higher-order function is a function that accepts a function as argument or returns a function as value. In Funclang we can define higher-order functions. For example, here is a function that returns `(lambda (x) c)`, which is also a function.

```
(lambda (c)          //Lambda abstraction with a single argument
  (lambda (x) c)     //Result of running this lambda expression
)
```

If we call this function with argument `1` as follows:

```
(                                //Function call
  (lambda (c)(lambda (x) c))     //Operator
  1                              //Operand
)
```

the result would be `(lambda (x) 1)`. If we call this function with argument `2` as follows:

```
(                                 //Function call
  (lambda (c) (lambda (x) c))     //Operator
  2                               //Operand
)
```

the result would be `(lambda (x) 2)`. So we can think of (lambda (c) (lambda (x) c)) as a constant function generator.

```
(define
  constgen
  (lambda (c) (lambda (x) c))
)
```

We can also define functions that take other functions as parameter. For example, the listing below defines a function that takes a function `f` and applies it to `1`.

```
(define
  applytoone
  (lambda (f) (f 1))
)
```

To try out this function, we can define another function.

```
(define add3
  (lambda (x) (+ x 3))
)
```

We can then use `add3` as an argument to `applytoone`.

```
$ (applytoone add3)
4
```

We can also create a function on the fly and give it as an argument to `applytoone`.

```
$ (applytoone (lambda (x) x))
1
$ (applytoone (constgen 342))
342
```

The second call to function `applytoone` is specially noteworthy. It makes use of the previous higher-order function `constgen` to create a function that when called returns `342`, and calls `applytoone` with this function as argument.

Higher-order functions can be particularly useful for defining reusable algorithmic structures. For example, the listing below shows a higher-order function that accepts an operation and a list and applies the operation on each element of the list.

```
(define map
  (lambda (op lst)
    ( if  (null?  lst )  ( list )
      (cons (op (car  lst ))  (map op (cdr lst )))
    )
  )
)
```

Here are some examples of using this function.

```
$ (define num1to10 (list  1 2 3 4 5 6 7 8 9 10))
$ (define  identity  (lambda (x) x))
$ (map identity  num1to10)
(1 2 3 4 5 6 7 8 9 10)
$ (define square  (lambda (x) (* x x)))
$ (map square num1to10)
(1 4 9 16 25 36 49 64 81 100)
```

Exercise

5.4.1. Define a function `filter` with the signature.

```
(define  filter  (lambda (test_op  lst ) ...) )
```

The function takes two inputs, an operator `test_op` that should be a single argument function that returns a boolean, and `lst` that should be a list of elements. The function outputs a list containing all the elements of "lst" for which the test_op function returned `#t`.

```
$ (define gt5? (lambda (x) (if (> x 5) #t #f)))
$ ( filter  gt5? ( list ))
()
$ ( filter  gt5? ( list  1))
()
$ ( filter  gt5? ( list  1 6))
(6)
$ ( filter  gt5? ( list  1 6 2 7))
(6 7)
$ ( filter  gt5? ( list  1 6 2 7 5 9))
(6 7 9)
```

5.4.2. Define a function `foldl` (fold left) with three parameters `op`, `zero_element` and `lst`. The parameter `op` itself is a two argument function, zero_element is the zero element of the operator function, e.g. 0 for plus function or 1 for multiply function, and lst is a list of elements. (plus function takes two parameters and adds them, multiply function takes two parameters and multiplies them.)

The function successively applies the `op` function to each element of the list and the result of previous `op` function (where no such results exists the zero element is used). The following interaction log illustrates foldl function

```
$ (define plus (lambda (x y) (+ x y)))
$ ( foldl  plus 0 ( list ))
0
$ ( foldl  plus 0 ( list  1))
1
$ ( foldl  plus 0 ( list  1 2))
3
$ ( foldl  plus 0 ( list  1 2 3))
6
$ ( foldl  plus 0 ( list  1 2 3 4))
10
```

5.4.3. Define a function `foldr` (fold right) with three parameters `op`, `zero_element` and `lst`. The parameter `op` itself is a two argument function, zero_element is the zero element of the operator function, e.g. 0 for '+' operator or 1 for multiply operator, and lst is a list of elements. (plus function takes two parameters and adds them, multiply function takes two parameters and multiplies them.)

The function successively applies the `op` function to each element of the list and the result of previous `op` function (where no such results exists the zero element is used). The following interaction log illustrates foldr function

```
$ (define minus (lambda (x y) (- x y)))
$ ( foldr  minus 0 ( list ))
0
$ ( foldr  minus 0 ( list  1))
```

```
1
$ ( foldr minus 0 ( list 1 2 3 4) )
$ -2
$ ( foldr minus 0 ( list 4 3 2 1) )
$ 2
```

5.4.4. *[Repeated]* If `f` is a numerical function and `n` is a positive integer, then we can form the `nth` repeated application of `f`, which is defined to be the function whose value at `x` is `f(f(...(f(x))...))`. For example, if `f` is the function `f(x) = x + 1`, then the `nth` repeated application of `f` is the function `f(x) = x + n`.

Define a function `repeated` that takes as inputs a procedure that computes `f` and a positive integer `n` and returns a function that computes the `nth` repeated application of `f`. Your function `repeated` should permit following usage.

```
$ (define inc (lambda (x) (+ x 1)))
$ ((repeated inc 2) 3)
5
$ ((repeated inc 10) 3)
13
$ ((repeated inc 100) 3)
103
$ ((repeated inc 3) 0)
3
$ ((repeated inc 3) -1)
2
$ ((repeated inc 3) -23)
-20
```

```
$ (define quad (lambda (x) (+ (* 2 x) 1)))
$ ((repeated quad 2) 2)
11
```

In this example `quad` is a function which given a number `x` returns `(+ (* 2 x) 1)`. Repeated is a function which takes a function (`quad` in this case) and another number (2 in this case) and returns a function which applies `quad` function twice on 2.

```
$ ((repeated quad 2) 3)
15
$ ((repeated quad 3) 2)
23
```

5.4.5. *[Smooth]* The idea of smoothing a function is an important concept in signal processing. If `f` is a function and `dx` is some small number, then the smoothed version of `f` is the function whose value at a point `x` is the average of `f(x - dx)`, `f(x)`, and `f(x + dx)`.

Write a function smooth that takes two inputs: a function that computes `f`, and a smoothing parameter `dx` and returns a function that computes the smoothed value of `f`. Your function `smooth` should permit following usage.

```
$(define quad (lambda (x) (+ (* 2 x) 1)))
$ ((smooth quad 1) 10)
21
$ ((smooth quad 1) 1)
3
$ ((smooth quad 1) 0)
1
$ ((smooth quad 1) 30)
61
```

5.4.6. *[GCD]* Greatest common divisor (GCD) of two numbers a and b is defined as follows. If `a > b` then `(gcd a b)` is gcd of `a - b` and `b`. Else if `a < b` then `(gcd a b)` is gcd of `a` and `b - a`. Otherwise, it is `a`.

1. Define a function gcd that computes greatest common divisor according the definition above.

   The following interaction log illustrates the function:

   ```
   $ (gcd 4 2)
   2

   $ (gcd 12 15)
   3
   ```

2. Use the function gcd to define gcds that takes two list of numbers and produces a third list that contains the list of gcd of corresponding elements from the first list and the second list. The following interaction log illustrates the function:

```
$(gcds ( list ) ( list ))

()

$ (gcds ( list  4) ( list  2))
(2)

$ (gcds ( list  4 12) ( list  2 15))
(2 3)
```

5.5 Functional Data Structures

Our ability to define abstractions that represent a piece of computation is essential to creating meaningful software systems, but we also need to be able to represent data structures. Fortunately, first-class functions of Funclang can serve both purposes equally well.

To illustrate, imagine that we want to create a data type that holds a pair of values. In order to define the data type, from the perspective of the client of the data type, it would be sufficient to provide definition for all operations that can be applied to that data type: a constructor for creating pairs, an observer for getting the first element of the pair, and another observer for getting the second element of the pair. We can define a constructor operation that can be used to create values of that data type as follows.

```
(define pair
  (lambda (fst snd)
    (lambda (op)
      (if op fst snd)
    )
  )
)
```

We can then use this operator to create a new pair value.

```
$ (pair 3 4)
(lambda ( op ) (if op fst snd))
```

Here, `pair` is a higher-order function. It returns a *function value* that takes a single parameter `op` and based on the value of `op` evaluates to `fst` or `snd`. We can also store this pair to use later.

```
$ (define apair (pair 3 4))
```

An important property to re-emphasize here is that `apair` is a function value. This function can be called by providing a value for the parameter `op`, which will then run its body `(if op fst snd)`. The body of the function has two variables `fst` and `snd`. So in addition to the code, the function value `apair` also stores mappings from `fst` to `3` and `snd` to `4`.

Now, we can define observer operations `first` and `second`.

```
$ (define first (lambda (p) (p #t)))
$ (define second (lambda (p) (p #f)))
$ ( first apair)
3
$ (second apair)
4
```

The function `first` and `second` assume that their argument `p` is a function. We can implement runtime checks, but for simplicity let us avoid them at the moment. These functions then call `p` with argument `#t` and `#f` respectively. Recall from before that when a function value created by the constructor `pair` is called with argument `#t` and `#f` it returns the value of `fst` or `snd`.

Exercise

5.5.1. *[Procedural representation of records]* Define a function, `record`, which takes a list of strings, `fields`, and a list of values, `values`, and returns a procedural representation of record. Write a second procedure `lookup`, which takes a procedural representation of record and a string and and returns the ith element of the list `values` if i is the index of `name` in list `fields`. For example:

```
$ (define roman (record ( list "i" "v" "x") ( list 1 5 10)))
$ (lookup roman "i")
```

```
1
$ (lookup roman "v")
5
$ (define empty (record ( list ) ( list )))
$ (lookup empty "a")
" error"
$ (define bad (record ( list ) ( list 1 2 3)))
$ (lookup bad "a")
" error"
$ (define bad2 (record ( list a b c) ( list )))
$ (lookup bad2 "a")
" error"
```

5.5.2. *[Procedural representation of tree]* Define a procedural representation of binary tree. A binary tree is either a terminal node that contains a value or a non-terminal node containing two children that are both binary trees. Define two constructors `leaf` and `interior` for creating terminal and non-terminal tree. Define an observer `traverse` that traverses the tree in a depth-first manner, applies `op` to each value stored in the tree, and applies `combine` to combine values from root node, left subtree, and right subtree to produce a single value.

```
(define leaf (lambda (leafval) ... ))
(define interior (lambda (rootval lefttree righttree ) ... ))
(define traverse (lambda (tree op combine) ... ))
```

5.6 Currying

All functions that we have written are defined using lambda abstractions that could take zero or more arguments. The ability to take zero or more arguments isn't an essential property of a lambda abstracton. In fact in a programming language with support for first-class functions it is possible to model multiple argument lambda abstractions as a combination of single argument lambda abstraction. Take the function `plus` that we defined previously as follows.

```
(define plus
  (lambda (x y)
```

```
    (+ x y)
  )
)
```

We can easily redefine plus using single argument lambda abstractions as follows.

```
(define plusCurry
  (lambda (x)
    (lambda (y)
      (+ x y)
    )
  )
)
```

This form where a function is defined using only single argument lambda abstraction is called its *curried form*. Since a curried form accepts only a single argument, calling it is slightly different. An example appears below.

```
$ ((plusCurry 3) 4)
7
```

Note that the expression `(plusCurry 3)` only partially evaluates the function. The value of this expression is a function, which when applied on 4 evaluates to the final value 7.

Exercise

5.6.1. *[volume]* Write a curried form of a function using lambda abstraction that takes length, width, and height of a cuboid, and computes the volume of cuboid by multiplying length, width and height.

Use that function, to compute volume of a cuboid with length equal to 3, width equal to 4 and height equal to 2.

5.6.2. *[speed-mph]* Define a curried version of the following procedure, call it speed-mph.

```
(define (speed kms kmToMile hours)
  (/ (* kms kmToMile) hours) )
```

Test your code by executing the following.

```
$ //Speed of carA that traveled 150 km in 2 hours in mph
$ (speed 150 0.62 2)
46.5
$ //Speed of carB that traveled 250 miles in 3 hours in mph
$ (speed 250 0.62 3)
51.666666666666664
```

5.6.3. *[speed-miles-per-two-hours]* Using your solution from the previous problem, define a procedure,

```
speedMilesPerTwoHours
```

which takes a single number (kilometers) as an argument and computes the speed of a car in miles per two hours.

Test your code by executing the following.

```
$ (speedMilesPerTwoHours 50)
$ (speedMilesPerTwoHours 120)
```

In the rest of this chapter, we will understand different aspects of the semantics of function definitions and calls by building an implementation of Funclang. As with Varlang, we will build an interpreter as opposed to a compiler. Fortunately, we have the Definelang implementation to work with. So the interpreter discussed in this chapter will build on that.

5.7 Syntax of Lambda and Call Expressions

We will first build support for lambda abstractions and calls, and differ discussions about other features such as the list-related functions, if expression, conditional expressions, etc...

- Like the grammar for Definelang, in the grammar for Funclang a program consists of zero or more definitions (`define` declarations) followed by an optional expression `(exp)?`. We also have two new expressions `lambdaexp` and `callexp`. We have used the syntax of these expressions often in previous sections of this chapter. We also need to extend the AST representation to support these two new nodes as shown in figure 5.2.

The AST node for a lambda expression has fields to store the formal parameter names and the body of the lambda expression, whereas the call

`Program`	::=	`DefineDecl`* `Exp?`	*Program*
`DefineDecl`	::=	`(define Identifier Exp)`	*Define*
`Exp`	::=		*Expressions*
		`Number`	*NumExp*
	|	`(+ Exp Exp`$^+$`)`	*AddExp*
	|	`(- Exp Exp`$^+$`)`	*SubExp*
	|	`(* Exp Exp`$^+$`)`	*MultExp*
	|	`(/ Exp Exp`$^+$`)`	*DivExp*
	|	`Identifier`	*VarExp*
	|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
	|	`( Exp Exp`$^+$`)`	***CallExp***
	|	`(lambda (Identifier`$^+$`) Exp)`	***LambdaExp***
`Number`	::=	`Digit`	*Number*
	|	`DigitNotZero Digit`$^+$	
`Digit`	::=	`[0-9]`	*Digits*
`DigitNotZero`	::=	`[1-9]`	*Non-zero Digits*
`Identifier`	::=	`Letter LetterOrDigit`*	*Identifier*
`Letter`	::=	`[a-zA-Z$_]`	*Letter*
`LetterOrDigit`	::=	`[a-zA-Z0-9$_]`	*LetterOrDigit*

Figure 5.1: Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are exactly the same as that in Definelang.

expression has fields to store the operator expression and zero or more operand expressions.

We would also need to extend the visitor infrastructure to accomodate these language changes. That will include adding new methods to the `Visitor` interface for `CallExp` and `LambdaExp` types. The standard visitors such as the `Formatter` would need to provide functionality to handle AST nodes of these new types. The reader is encouraged to review these changes in the companion code before proceeding further.

5.8 Value of a Lambda Expression

Functions are a *first-class feature* in Funclang, since values of type functions are treated just like numeric, string, boolean and list values. They can be

```
1  class LambdaExp extends Exp {
2    List<String> _formals;
3    Exp _body;
4    LambdaExp(List<String> formals, Exp body) {
5      _formals = formals;
6      _body = body;
7    }
8    List<String> formals() { return _formals; }
9    Exp body() { return _body; }
10   Object accept( Visitor visitor , Env env) {
11     return visitor . visit (this, env);
12   }
13 }

15 class CallExp extends Exp {
16   Exp _operator;
17   List<Exp> _operands;
18   CallExp(Exp operator, List<Exp> operands) {
19     _operator = operator;
20     _operands = operands;
21   }
22   Exp operator() { return _operator; }
23   List<Exp> operands() { return _operands; }
24   Object accept( Visitor visitor , Env env) {
25     return visitor . visit (this, env);
26   }
27 }
```

Figure 5.2: New AST nodes for the Funclang Language.

passed as parameters, returned as values, and stored in environments[1]. This enables us to program higher-order functions, and procedural representation of data structure as discussed in previous sections.

Since Funclang program and expressions can produce functions as val-

[1]An expression `(let ((identity (lambda (x) x))) ...)` stores a 2-tuple: a name `identity` and a value of type `FunVal` in the environment

ues, it becomes essential to extend the set of legal values to include a new kind of value: `FunVal`.

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`FunVal`	*Function Values*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`FunVal`	::=	`(FunVal` $\texttt{var}_0$`,..,` $\texttt{var}_n$ `e env)`	*FunVal*
		where $\texttt{var}_0$`,..,` $\texttt{var}_n \in$ Identifier,	
		`e` $\in$ Exp, `env` $\in$ Env	

Figure 5.3: The Set of Legal Values for the Funclang Language

From our previous discussion about the lambda expression, recall that a lambda expression evaluates to a function value.

VALUE OF LAMBDAEXP

$$\frac{\texttt{(FunVal var}_i\texttt{,for i = 0...k exp}_b\texttt{ env) = v}}{\texttt{value (LambdaExp var}_i\texttt{,for i = 0...k exp}_b\texttt{) env = v}}$$

Here, `FunVal` is a new kind of value for Funclang. It encapsulates the names of the formal parameter, the body of the lambda expression, and the current environment. A realization of `FunVal` is shown in figure 5.4.

The implementation of the lambda expression case in the interpreter is shown below.

```
Value visit (LambdaExp e, Env env) {
  return new Value.FunVal(env, e.formals(), e.body());
}
```

This implementation exactly models the semantics. It creates a function value that encapsules formal parameter names, function body, and the current environment.

5.9 Value of a Call Expression

Evaluating a call expression has three key steps.

1. *Evaluate operator.* Evaluate the expression whose value will be the function value. For example, for the call expression `(identity i)` the variable expression `identity`'s value will be the function value.

```
1  class FunVal implements Value {
2    private Env _env;
3    private List<String> _formals;
4    private Exp _body;
5    public FunVal(Env env, List<String> formals, Exp body) {
6      _env = env;
7      _formals = formals;
8      _body = body;
9    }
10   public Env env() { return _env; }
11   public List<String> formals() { return _formals; }
12   public Exp body() { return _body; }
13 }
```

Figure 5.4: FunVal: A New Kind of Value for Functions

2. *Evaluate operands.* For each expression that is in place of a formal parameter, evaluate it to a value. For example, for the call expression `(identity i)` the variable expression `i`'s value will be the only operand value.

3. *Evaluate function body.* This step has three parts.

   a) Find the expression that is the body of the function value,

   b) create a suitable environment for that body to evaluate, and

   c) evaluate the body.

   For example, for the call expression `(identity i)` if the function value is `(lambda (x) x)`, then the body of the function would be `x`. A suitable environment for running the body of this function would have a binding from formal parameter name `x` to actual parameter value, `1` for our example call expression. Evaluating that body would result in the value `1`.

The value relation below models this intent.

VALUE OF CALLEXP

$$\frac{\begin{array}{c}\texttt{value exp}_b\ \texttt{env}_{k+1} = \texttt{v} \\ \texttt{value exp env} = (\texttt{FunVal var}_i\texttt{,for i = 0...k exp}_b\ \texttt{env}_0) \\ \texttt{value exp}_i\ \texttt{env} = \texttt{v}_i, \text{for i} = 0...\text{k} \\ \texttt{env}_{i+1} = (\texttt{ExtendEnv var}_i\ \texttt{v}_i\ \texttt{env}_i), \text{for i} = 0...\text{k}\end{array}}{\texttt{value (CallExp exp exp}_i\texttt{,for i = 0...k) env = v}}$$

First condition above the line says that the value of a call expression is the value of `body` in a new environment $\texttt{env}_{k+1}$. Second condition says that evaluating `exp` results in a function value, and that function value has $\texttt{exp}_b$ as the function body. We also get the names of the formal parameters $\texttt{var}_i$`,for i = 0...k` from the function value. Third condition says that evaluating each of the $\texttt{exp}_0$ to $\texttt{exp}_k$ in the original envirnoment results in values $\texttt{v}_0$ to $\texttt{v}_k$ respectively. The fourth condition of this relation defines the extended environment $\texttt{env}_{k+1}$ that maps formal parameter names to corresponding actual parameter values.

The case for `CallExp` in the interpreter implements this semantics as shown below. To improve user experience, it also implements some checks. For example, if the result of evaluating the operator is not a function value, e.g. in expression `(1 2)`, the result is a dynamic error. Similarly, if the number of formal parameters do not match the number of actual arguments, the call expression results in a dynamic error also.

Dynamic Errors To support dynamic error as a legal value, we redefine the set of legal values in Funclang as shown in figure 5.5. A dynamic error encapsulates a string `s` e.g. for describing the cause of the error.

There are no expressions that the programmer can use to directly produce dynamic errors with their own custom string. However, programmers can easily produce dynamic errors indirectly, e.g. by using a call expression incorrectly.

Implementation of Call Expression The implementation also mirrors the value relation and the informal semantics of call expressions.

```
Value visit(CallExp e, Env env) {
  //Step 1: Evaluate operator
  Object result = e.operator().accept(this, env);
```

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
		| `FunVal`	*Function Values*
		| `DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`FunVal`	::=	`(FunVal` $\texttt{var}_0$`,..,` $\texttt{var}_n$ `e env)`	*FunVal*
		where $\texttt{var}_0$`,..,` $\texttt{var}_n \in$ Identifier,	
		`e` $\in$ Exp, `env` $\in$ Env	
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where `s` $\in$ the set of Java strings	

Figure 5.5: The set of Legal Values for the Funclang Language with new dynamic error value

```
    if (!( result  instanceof Value.FunVal))
      return new Value.DynamicError("Operator not a function");
    Value.FunVal operator =  (Value.FunVal)  result ;
    List<Exp> operands = e.operands();

    //Step 2: Evaluate operands
    List<Value> actuals = new ArrayList<Value>(operands.size());
    for(Exp exp : operands)
      actuals.add((Value)exp.accept(this, env));

    //Step 3: Evaluate function body
    List<String> formals = operator.formals();
    if (formals.size()!=actuals.size())
      return new Value.DynamicError("Argument mismatch in call");
    Env fenv = appendEnv(operator.env(), initEnv);
    for (int i = 0; i < formals.size(); i++)
      fenv = new ExtendEnv(fenv, formals.get(i), actuals.get(i));
    return (Value) operator.body().accept(this, fenv);
  }
```

In the third step, evaluate function body, an environment for running the function body `fenv` is created by first appending the bindings from function value and the initial environment, and then successively adding mapping from formal parameters to actual parameters. First step makes

use of a recursively defined helper function `appendEnv` below.

```
Env appendEnv(Env fst, Env snd){
  if(fst.isEmpty())
    return snd;
  ExtendEnv f = (ExtendEnv) fst;
  return new ExtendEnv(appendEnv(f.saved_env(),snd), f.var(), f.val());
}
```

In summary, to support function definition and calls we included a new kind of value, `FunVal` for encapsulating aspects of a function definition, added support for `lambda` and call expressions in the read phase, supported new AST nodes to store lambda and call expressions, and changed the evaluator to add semantics for lambda and call expressions. We also added a new kind of value `DynamicError` for representing error conditions in the evaluation of function calls.

Exercise

5.9.1. *[Environment optimization]* Optimize the function call semantics by reducing the size of the environment saved in FunVal so that it only contains mappings from free variables in the function body to their bindings in the current environment.

5.9.2. *[Substitution-based Call Expression]* Extend the Funclang programming language from previous question to implement a *substitution-based variation* of the call expression, say `[ ]` expression. Recall that a substitution-based semantics works as follows. The value of `[ (lambda (x y) (+ x y)) 3 4]` is the value of a new expression created from the original function body `(+ x y)` by replacing `x` with `3` and `y` with `4`. According to the substitution-based semantics the value of the call expression is the value of `(+ 3 4)` that is `7`.

- The grammar of this new language feature should be exactly the same as the grammar of the `call` expression in the Funclang language, except for the syntax `[ ]`.
- Implement substitution as a method `subst` for each AST node such that given a list of variable names, and a list of values the `subst` method returns a copy of current AST node with each free variable name substituted with corresponding value.

5.9.3. *[Tracing Lambda Abstractions]* Extend the Funclang programming language to implement a tracing lambda expression `tlambda` that prints "`Entering a function`" before starting to run the function body and "`Exiting a function`" after returning.

5.9.4. *[Dynamic Binding]* An alternative to static binding is *dynamic binding* in which the function body is evaluated in an environment obtained by extending the environment at the function call point (instead of the environment at the function declaration point that is used for static binding).

Extend FungLang with a dynamic call expression that has syntax `'(' 'dynamic' expr expr* ')'`. A dynamic call expression is evaluated using dynamic binding.

The following examples illustrate the difference between dynamic and static bindings during function calls. Consider the program below.

```
(let
  ((a 3))
  (let
    ((p  (lambda (x) (- x a))))
    (let
      ((a  5))
      (- (p 11) (p 11))
    )
  )
)
```

The value of this program is 0. In the expression above variable a is bound to 3.

Now consider another version of the program that uses dynamic call expression.

```
(let
  ((a 3))
  (let
    ((p  (lambda (x) (- x a))))
    (let
      ((a  5))
      (- (p 11)  (dynamic p 11))
```

```
    )
  )
)
```

The value of this program is 2. Here variable a is bound to 5.

Consider another program that uses static binding only.

```
(let
  ((a 3))
  (let
    ((p  (lambda (x) (- x a))))
    (let
      ((a  5))
      (- a (p 2))
    )
  )
)
```

The value of this program is 6. Here variable a is bound to 3.

```
(let
  ((a 3))
  (let
    ((p  (lambda (x) (- x a))))
    (let
      ((a  5))
      (- a (dynamic p 2))
    )
  )
)
```

If the program used dynamic binding, since the variable a will be bound to 5, value of this program will be 8.

5.9.5. *[Default parameters]* In this question you will explore a semantic variation of function call and return.

A function definition can provide default values for the function's parameters. Extend the Funclang interpreter to support functions with the default value for their last parameters (not all parameters).

The following interaction log illustrates the syntax and semantics for our desired extension:

```
$ (define func (lambda ((v = 342)) v) )
$ (func)
342

$ (func 541)
541

$ (define add (lambda (a (b = 5)) (+ a b)) )
$ (add 8)
13

$ (add 8 4)
12
```

5.9.6. *[Implicit parameters]* In this question you will explore a semantic variation of function call and return.

A function can take an implicit parameter. Extend the Funclang language to support function calls that can (optionally) provide an implicit parameter 'this'.

The following interaction log illustrates the syntax and semantics of our desired extension:

```
$ (define f (lambda () this))
```

Here, the function f is making use of the implicit parameter 'this' in its body.

```
$ (define obj ( list 3 4 2))
$ (obj.f)
(3 4 2)
```

In the function call, obj is being provided as the value of the implict parameter 'this' to be used in the function body f.

Here are some more examples of defining and using such functions.

```
$ (define first (lambda () (car this)))
$ (obj. first )
```

```
3
$ (define second (lambda () (car (cdr this))))
$ (obj.second)
4
```

5.9.7. *[Variable argument functions]* In this question you will explore a semantic variation of function call and return.

A function can take variable number of parameters. Such functions are known as varargs or varidiac function. Extend the Funclang language to support variable argument functions with the following syntax.

'(' **lambda** '(Identifier ... ')' exp ')'

This syntax (especially `...` three dots) allows a lambda expression with variable number of parameters. Parameters of variadic functions are treated as a list in the body of the function. For example `(lambda (x ...) (car x))` defines an anonymous function with variable parameters x where the body of the function, returns the first parameter of the function.

Variable number of arguments could be passed when calling a variadic function. For example, in the program `((lambda (x ...) (car x)) 8 2 3)` the three parameters `8`, `2` and `3` are passed to the function. The program returns the first argument in the list of arguments, i.e. it returns `8`.

```
$ ( (lambda (x ...) (car x)) 8 2 3)
8
$ ( (lambda (x ...) (cdr x)) 8 2 3)
(2 3)
$ ( define vardiacFunc (lambda (y ...) (car y)))
$ (vardiacFunc 2)
2
$ (vardiacFunc 2 3 4)
2
$ (vardiacFunc 2 3 4 5)
2
```

5.10 Semantics of Conditional Expressions

We have also added a conditional expression, `if` and three comparison expressions `<`, `=`, and`>` to the Funclang language to improve expressiveness of the language. The syntax for these extensions is given in figure 5.6.

An if expression is different from if statement in ALGOL-like languages. In Java for example, an if statement can have a condition, a then block of statements and optionally an else block of statements. An if expression in function consists of three expressions: the condition expression, the then expression, and the else expression. All parts are mandatory, and all three of them are expressions.

The comparison expressions are standard, except like other expressions in Funclang are written using the prefix form.

`Exp`	::=	*Expressions*
	`Number`	*NumExp*
|	`(+ Exp Exp`$^+$`)`	*AddExp*
|	`(- Exp Exp`$^+$`)`	*SubExp*
|	`(* Exp Exp`$^+$`)`	*MultExp*
|	`(/ Exp Exp`$^+$`)`	*DivExp*
|	`Identifier`	*VarExp*
|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
|	`( Exp Exp`$^+$`)`	*CallExp*
|	`(lambda (Identifier`$^+$`) Exp)`	*LambdaExp*
|	`(if Exp Exp Exp)`	***IfExp***
|	`(< Exp Exp)`	***LessExp***
|	`(= Exp Exp)`	***EqualExp***
|	`(> Exp Exp)`	***GreaterExp***
|	`#t` | `#f`	***BoolExp***

Figure 5.6: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

To add these new expressions to the language, we should first decide about the legal values produced by these expressions. Here, we have two options.

1. Encode using `NumVal`: We could choose to use the domain of `NumVal` as legal values produced by the comparison expressions and that con-

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`BoolVal`	*Boolean Values*
	|	`FunVal`	*Function Values*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`BoolVal`	::=	`(BoolVal true)`	***BoolVal***
	|	`(BoolVal false)`	
`FunVal`	::=	`(FunVal` $\texttt{var}_0$`,.., `$\texttt{var}_n$ `e env)`	*FunVal*
		where $\texttt{var}_0$`,.., `$\texttt{var}_n$ $\in$ Identifier,	
		`e` $\in$ Exp, `env` $\in$ Env	
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where `s` $\in$ the set of Java strings	

Figure 5.7: The set of Legal Values for the Funclang Language with new **boolean value**

sumed by the if expressions. For example, a greater expression `(> a b)` could produce `0` as value when `a` is not greater than `b`, otherwise it can produce a nonzero value. Similarly, an if expression `(if (> a b) ..)` could assume that any value of its condition expression `(> a b)` greater than zero would be taken as true, and false otherwise.

2. Introduce new kind of values: We could also choose to introduce a new kind of value to model boolean conditions. A disadvantage of this option is that the language definition and implementation must support an additional feature. A distinct advantage is that the results of a boolean comparison may not be confused with results of arithmetic operations since they would be values of different kind. This has the potential to reduce errors, therefore most modern languages introduce boolean as a new kind of value.

In Funclang, following recent trends, we will also extend the set of legal values produced by Funclang programs to include boolean values. This extended set of values is shown in figure 5.7. A true literal is represented as `#t` and false literal as `#f` as shown in figure 5.6. A `BoolVal` can either be true or false.

With new set of legal values, and syntax, we can give the semantics of the three comparison expressions in Funclang as shown below.

VALUE OF GREATEREXP

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \quad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{>}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (GreaterExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

VALUE OF EQUALEXP

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \quad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{==}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (EqualExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

VALUE OF LESSEXP

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \quad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{<}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (LessExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

The semantics of if expression is a bit more involved, and is given by the two rules below. First rule handles the case when the boolean condition $\texttt{exp}_{cond}$ evaluates to true, and second rule handles the false case.

VALUE OF IFEXP - TRUE

$$\frac{\texttt{value exp}_{cond}\ \texttt{env = (BoolVal true)} \quad \texttt{value exp}_{then}\ \texttt{env = v}}{\texttt{value (IfExp exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{) env = v}}$$

VALUE OF IFEXP - FALSE

$$\frac{\texttt{value exp}_{cond}\ \texttt{env = (BoolVal false)} \quad \texttt{value exp}_{else}\ \texttt{env = v}}{\texttt{value (IfExp exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{) env = v}}$$

The reader is encouraged to review the implementation of these expressions in the companion code before proceedings further.

Exercise

5.10.1. *[Logical conjunction and disjunction expressions]* Extend the syntax and semantics of the Funclang language to add support for the logical conjunction (and) and logical disjunction (or) expressions.

Examples:

```
$ (&& (< 3 4) (> 4 2))
#t
$ (|| (> 3 4) (> 4 2))
#t
```

5.10.2. *[Switch expression]* Extend the syntax and semantics of the Funclang language to add support for a switch expression.

Examples:

```
$ (define x 0)
$ (switch (x) (case 0 3) (case 1 4) (case 2 2))
3
$ (define x 1)
$ (switch (x) (case 0 3) (case 1 4) (case 2 2))
4
$ (switch ((+ x 1)) (case 0 3) (case 1 4) (case 2 2))
2
$ (switch (x) (case 0 3) (case 1 4) (case 2 (+ 1 1)))
2
```

5.11 Semantics of Pairs and Lists

A Funclang programmers also has access to pair values, list values and related expressions. Recall that a pair in Funclang is a 2-tuple. A list is either an empty list or a pair, where the second element is a list.

Funclang supports several built-in expressions such as `list` for creating a new list, `car` for getting the first element of a pair, `cdr` for getting the second element of a pair, `cons` for constructing a pair, and `null?` for checking for empty list. Support for these expressions is orthogonal to function-related features (as we have discussed in section 5.5), but Funclang includes this support to make it easier to write interesting programs.

To support these expressions, we include two additional kinds of value `PairVal` and `Null`. This is an inductively-defined value. A `ListVal` is either an empty list, represented as the type `EmptyList`, or a pair that consists of a value as the first element and a `ListVal` as a second element, represented as the type `ExtendList`. New AST nodes `ListExp`, `CarExp`, `CdrExp`, `ConsExp`, and `NullExp` are also added to store these expressions.

Exp	::=	*Expressions*
	Number	*NumExp*
|	(+ Exp Exp$^+$)	*AddExp*
|	(- Exp Exp$^+$)	*SubExp*
|	(* Exp Exp$^+$)	*MultExp*
|	(/ Exp Exp$^+$)	*DivExp*
|	Identifier	*VarExp*
|	(let ((Identifier Exp)$^+$) Exp)	*LetExp*
|	(Exp Exp$^+$)	*CallExp*
|	(lambda (Identifier$^+$) Exp)	*LambdaExp*
|	(if Exp Exp Exp)	*IfExp*
|	(< Exp Exp)	*LessExp*
|	(= Exp Exp)	*EqualExp*
|	(> Exp Exp)	*GreaterExp*
|	#t | #f	*BoolExp*
|	(car Exp)	***CarExp***
|	(cdr Exp)	***CdrExp***
|	(null? Exp)	***NullExp***
|	(cons Exp Exp)	***ConsExp***
|	(list Exp*)	***ListExp***

Figure 5.8: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

The semantics of list-related expressions is given in terms of the list values. The value of a `ListExp` is given by the following relation.

$$\texttt{value (ListExp exp}_0 \ \ldots \ \texttt{exp}_n\texttt{) env = (ListVal val}_0 \ \texttt{lval}_1\texttt{)}$$

$$\texttt{where exp}_0 \ \ldots \ \texttt{exp}_n \in \texttt{Exp} \quad \texttt{env} \in \texttt{Env}$$

$$\texttt{value exp}_0 \ \texttt{env = val}_0\texttt{, } \ldots\texttt{, value exp}_n \ \texttt{env = val}_n$$

$$\texttt{lval}_1 \ \texttt{= (ListVal val}_1 \ \texttt{lval}_2\texttt{), } \ldots\texttt{,}$$

$$\texttt{lval}_n \ \texttt{= (ListVal val}_n \ \texttt{(EmptyList))}$$

A corollary of the relation is:

$$\texttt{value (ListExp) env = (EmptyList)}$$

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`BoolVal`	*Boolean Values*
	|	`FunVal`	*Function Values*
	|	`PairVal`	*Pair Values*
	|	`NullVal`	*Null Value*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`BoolVal`	::=	`(BoolVal true)`	*BoolVal*
	|	`(BoolVal false)`	
`FunVal`	::=	`(FunVal` $\texttt{var}_0$`,..,` $\texttt{var}_n$ `e env)`	*FunVal*
		where $\texttt{var}_0$`,..,` $\texttt{var}_n \in$ Identifier,	
		`e` $\in$ Exp, `env` $\in$ Env	
`PairVal`	::=	`(PairVal` $\texttt{v}_0$ $\texttt{v}_1$`)`	***PairVal***
		where $\texttt{v}_0$`,` $\texttt{v}_1 \in$ `Value`	
`NullVal`	::=	`(NullVal)`	***NullVal***
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where `s` $\in$ the set of Java strings	

Figure 5.9: The set of Legal Values for the Funclang Language with new **pair and null values**

The value of a `CarExp` is given by:

```
value (CarExp exp) env = val
  where exp ∈ Exp  env ∈ Env
value exp env = (ListVal val lval) where lval ∈ ListVal
```

The value of a `CdrExp` is given by:

```
value (CdrExp exp) env = lval
  where exp ∈ Exp  env ∈ Env
value exp env = (ListVal val lval) where lval ∈ ListVal
```

The value of a `ConsExp` is given by:

```
value (ConsExp exp exp') env = (ListVal val lval)
```

```
where exp,exp' ∈ Exp  env ∈ Env  value exp env = val
                value exp' env = lval
```

The value of a `NullExp` is given by:

```
value (NullExp exp) env = #t if value exp env = (EmptyList)
                value (NullExp exp) env = #f
if value exp env = (ListVal val lval') where lval' ∈ ListVal
                where exp ∈ Exp  env ∈ Env
```

Exercise

5.11.1. [Pairs and Lists] In some programming languages such as Scheme, pairs are the basic values and lists are defined in terms of pairs. Define a pair as a 2-tuple, and redefine the value relations for `list`, `car`, `cdr`, `cons`, and `null?` in terms of pair values.

5.11.2. *[equal? expression]* Extend the FuncLang language to support a new expression `equal?` that takes two subexpressions and returns `#t` if their values are equal and `#f` otherwise.

The following interaction log illustrates the semantics of equal?

```
$(equal? #t #t)
#t
$(equal? #t 1)
#f
$(equal? #t "Hello")
#f
$(equal? (list) (list))
#t
$(equal? (list) (list 1))
#f
$(equal? (list (list 1)) (list (list 0)))
#f
$(equal? (list (list 1)) (list (list 1)))
#t
```

```
$(equal? (+ 2 3) (+ 2 3))
#t
$(equal? (let ((x 2)) x) (let ((y 2)) y))
#t
$(equal? (let ((x 2)) y) (let ((y 2)) x))
funclang.Env$LookupException: No binding found for name: y
```

5.11.3. *[List comprehension]* Extend the Funclang programming language to support a new list comprehension expression. A list comprehension is a concise way of representing higher-order functions that operate over list. It should have the following syntax:

```
comp-exp : '[' exp '|' identifier '<-' exp ',' exp ']'
```

The following interaction log illustrates the list comprehension expression:

```
$ (define add2 (lambda (x) (+ x 2)))

$ (define list1 (list 3 4 2))

$ [ (add2 x) | x <- list1 , #t]
(5 6 4)

$ [ (add2 x) | x <- (list) , #t]
()

$ [ (add2 x) | x <- list1 , (> x 2)]
(5 6)
```