Programming Languages: An Active Learning Approach ©2008, Springer Publishing

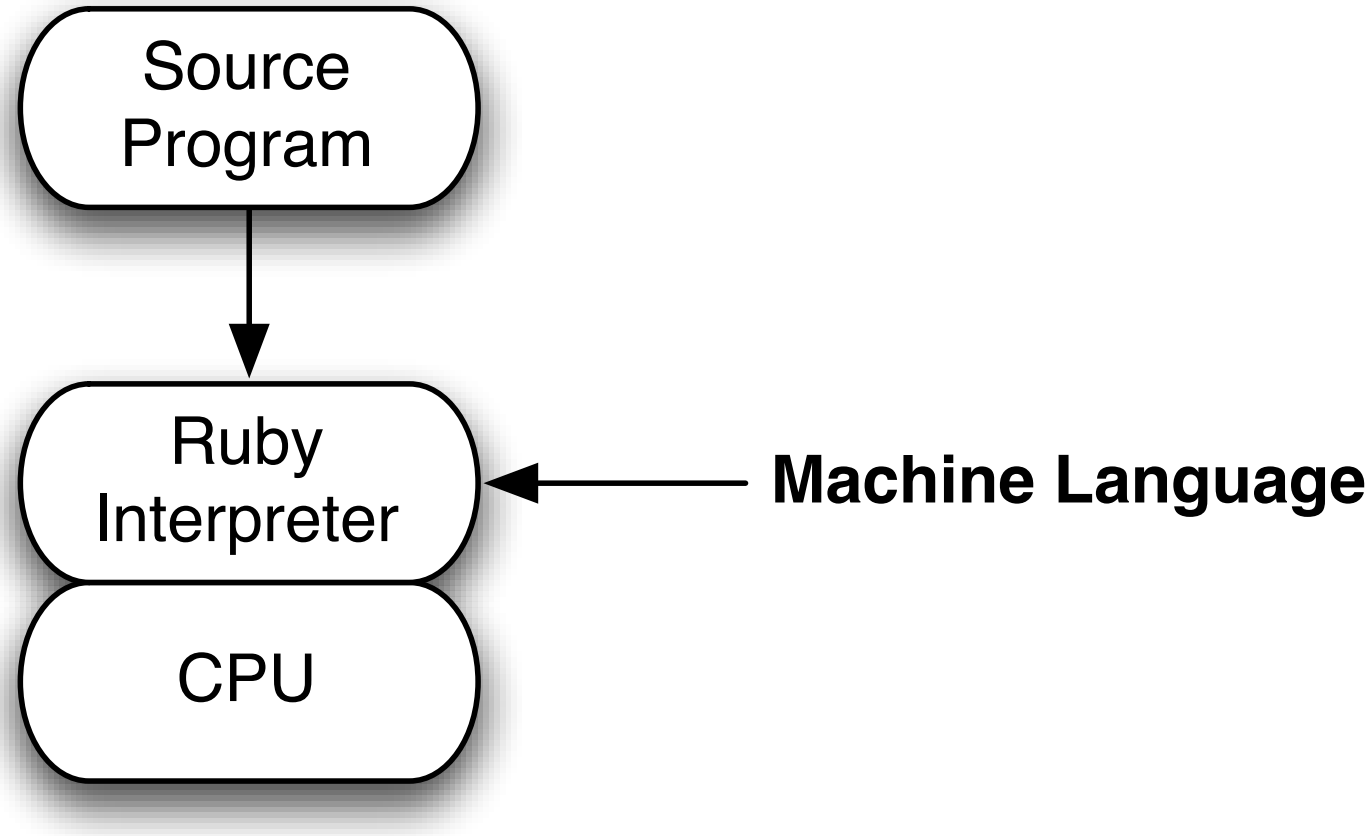
Chapter 4 Object-Oriented Programming with Ruby

Kent D. Lee

Object-Oriented Programming with Ruby

- This chapter introduces you to the Ruby language through the development of an interpreter for the calculator language like the C++ version
- Unlike C++, Ruby is an interpreted language.
- Ruby is also dynamically typed.
- Using Ruby and C++ to develop the same project can give you some insight into the differences and similarities of the two languages.
- Ruby was influenced by several older (relative to Ruby anyway) programming languages including Smalltalk and Perl.

Interpreting Ruby Programs



Object-Oriented Programming with Ruby

Example 4.1

In Linux and Unix there is a simple command called *echo* that will echo back to the screen the text you type. To write a similar tool, we can create a file called *recho* and enter this into the file.

```
#!/usr/local/bin/ruby
s = gets
puts s
```

The first line of the program may change slightly depending on where the Ruby interpreter is located. If you are not sure, you can type *which ruby* and Linux will tell you where the interpreter is located. After creating the file called *recho*, Linux needs to be told that the file is executable. To do this you enter the command

```
chmod +x recho
```

which tells Linux that echo is eXecutable. Then you can run this program by typing *recho* at the command line. That's it, you've written your first Ruby program. It doesn't work exactly like the Linux *echo* command, but it's close. You only have to make the file executable once. If you make more changes to the file, the program is still executable and you can change and try your program to your heart's content.

Designing Calc

- As with the C++ version, we'll call the calculator interpreter **calc**.
- It will be invoked from the command-line as described in example 4.2.

Designing Calc

Example 4.2

Here is a typical session with the completed calc interpreter.

```
%>calc
Please enter a calculator expression: (4S+5)*(7-R)
The result is 27
%>
```

Designing Calc

- The **S** represents the store operator.
- It stores the value that is computed to the left of it.
- The **R** represents the value that is stored in the memory location.

Designing Calc

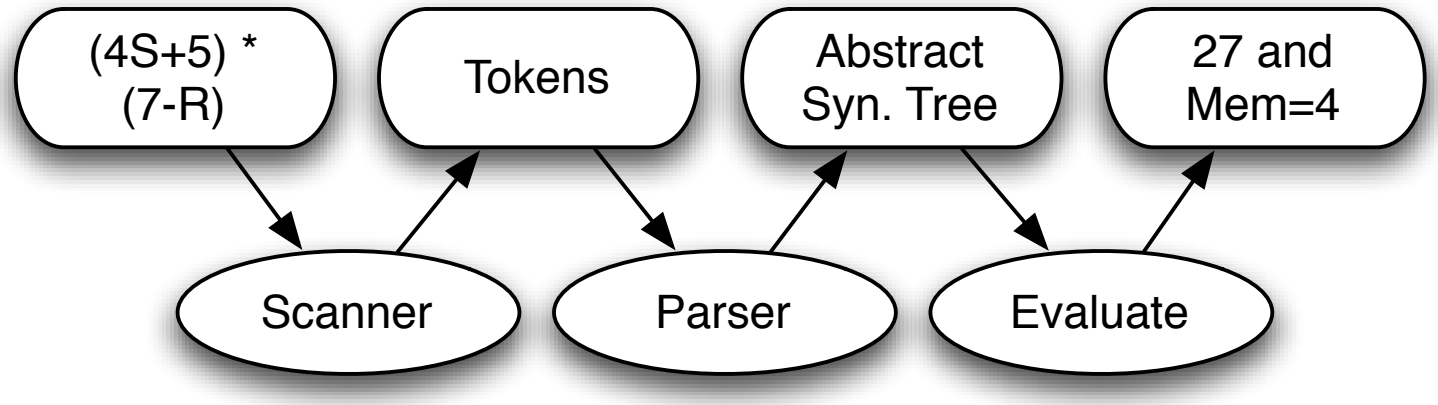
👉 Practice 4.1

Evaluate the following calculator expressions by hand.

1. (4+5)S*R
2. 3S + R
3. R + 3S
4. 2S*4 + R

▸ View Solution

Data flow through the Calc Interpreter



The Token Class

- Each part of an expression that is read by the scanner becomes a token in the interpreter.
- The calculator language has several types of tokens.
- The complete list is *number*,+,-,*,/,(,),**S**,and **R**.

The Token Class

👉 Practice 4.2

Identify the tokens in these expressions. Refer to the paragraph above to be sure you find them all.

1. 3S + R
2. (4+5)S*R

▸ View Solution

The Token Class

Example 4.3

Here is the Ruby code representing token objects in the project. A Ruby class starts with the `class` keyword and ends with the keyword `end`. The constructor of a class is called `initialize`. Instance variables are preceded with `@` when used in an expression or statement. The keyword `attr_reader` is something you likely haven't seen before. It is syntactic sugar. Ruby is designed as a language for programmers to relieve them of menial tasks. The `attr_reader` keyword will create accessor methods for the list of instance variables that follow it.

```
1  class Token
2     attr_reader :type, :line, :col
3
4     def initialize(type,lineNum,colNum)
5        @type = type
6        @line = lineNum
7        @col = colNum
8     end
9  end
```

The Token Class

- The lack of a garbage collector in C++ puts more burden on the programmer.
- A C++ programmer must write a destructor for a class when an object of that type points to a value that is heap allocated.
- Ruby is garbage collected, and therefore destructors are not relevant.
- The `@` sign in the code above is used to distinguish a local variable from an instance variable.
- Instance variables in Ruby must be preceded by the `@` to tell the Ruby interpreter that it is an instance variable.

Parameter Passing in Ruby vs C++

- Because C++ allows parameters to be passed by value, pointer, or reference, it is sometimes necessary to declare functions to be **const** or constant, meaning that the function is an accessor method.
- In Ruby there is only one way to pass parameters. They may only be passed by reference which is the way Java passes object parameters.
- It should be noted that some of the objects in Ruby are immutable.
- For example, the integer class is immutable.
- When an object, or all objects of a class, are immutable, then the distinction of passing parameters by reference or by value is irrelevant.

Parameter Passing in Ruby vs C++

👉 Practice 4.3

Why doesn't the following code mutate the integer from 5 to 6?

```
x = 5
x = x + 1
```

▸ View Solution

Accessor and Mutator methods in Ruby

- Given the `attr_reader` declaration in example 4.3 you can access the `type` instance variable of a token, `t`, by writing `t.type`.
- You cannot change an instance variable from outside the class using an `attr_reader`. If you intend to let code outside the class change an instance variable you must declare an `attr_writer`.
- In most languages an object is immutable if there exist no methods that mutate the object.
- However, Ruby contains an interesting method defined on all objects called `freeze` that takes no parameters.
- When this method is called on an object it is frozen at run-time making any mutator method calls on the object throw an exception.

Inheritance

Example 4.4

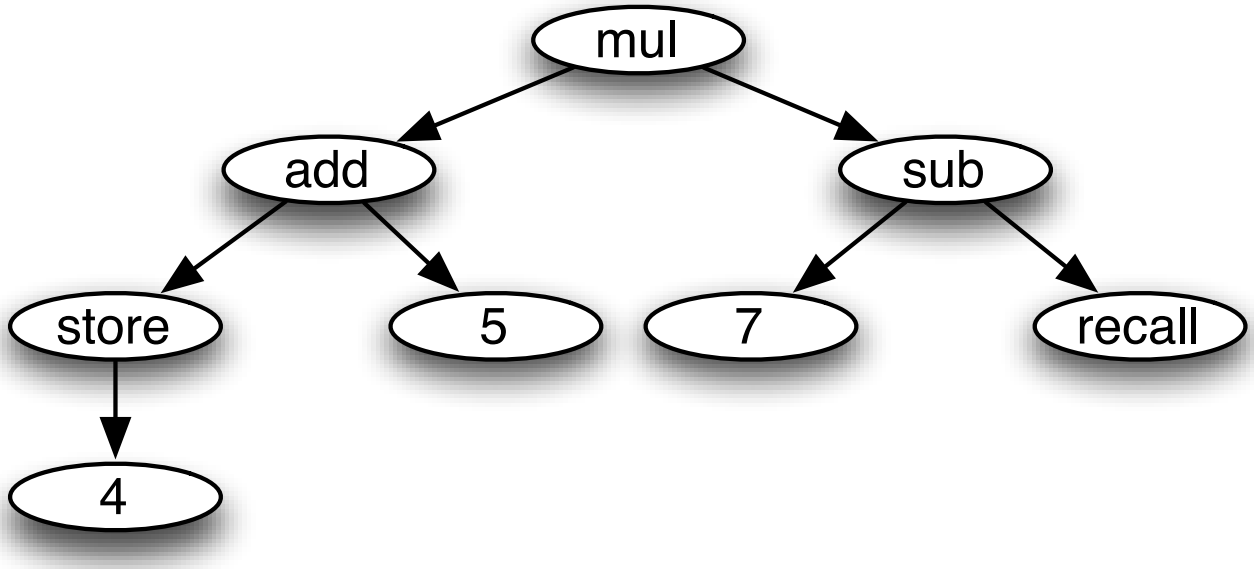
Here is the code for the LexicalToken class. The `< Token` indicates that the class `LexicalToken` inherits from `Token`. Line 2 adds a `lex` accessor function. Line 5 calls the super class' constructor to initialize the inherited part of the object, while line 7 initializes the new part of the object.

```
1  class LexicalToken < Token
2     attr_reader :lex
3
4     def initialize(type,lex,lineNum,colNum)
5        super(type,lineNum,colNum)
6
7        @lex = lex
8     end
9  end
```

The AST Classes

- The AST classes can be designed as a hierarchy of classes where each type of AST class represents one type of node in an abstract syntax tree.
- If an evaluate method is defined for each type of node in an AST as well, then the evaluate methods can recursively compute the value associated with a calculator expression's AST.

Abstract Syntax Tree of expression in figure **??**



The AST Classes

1. The traversal begins by recursively descending the left side of the tree down to the 4 node. Visiting that node returns 4.
2. The store node takes the 4 and stores it in the calculator's memory. It also returns the 4.
3. The add can't be visited yet since it has a right child (the 5). The 5 node is visited and returns the 5.
4. The add node can now be visited. It take the 4 and the 5, adds them together and returns the 9.
5. The mul node can't be visited until its right child is visited. Postorder traversal of the sub node calls the traversal on the 7 node, which returns 7.
6. The sub node still can't be visited yet. The recall node is traversed and returns the value in the calculator memory, the 4.
7. The 7-4 is computed by visiting the sub node and returns 3.
8. Visiting the mul node computes 9*3 or 27.

The AST Classes

Example 4.5

Here is the code for the two base classes, UnaryNode and BinaryNode.

```
1  class BinaryNode
2     attr_reader :left, :right
3
4     def initialize(left,right)
5        @left = left
6        @right = right
7     end
8  end
9
10 class UnaryNode
11    attr_reader :subTree
12
13    def initialize(subTree)
14       @subTree = subTree
15    end
16 end
```

The AST Classes

Example 4.6

Here is the code for the AddNode and SubNode classes. The other classes are left as an exercise.

```
1  class AddNode < BinaryNode
2      def initialize(left, right)
3          super(left,right)
4      end
5
6      def evaluate()
7          return @left.evaluate() + @right.evaluate()
8      end
9  end
10
11 class SubNode < BinaryNode
12     def initialize(left, right)
13         super(left,right)
14     end
15
16     def evaluate()
17         return @left.evaluate() - @right.evaluate()
```

```
18          end
19      end
```

The AST Classes

👉 Practice 4.4

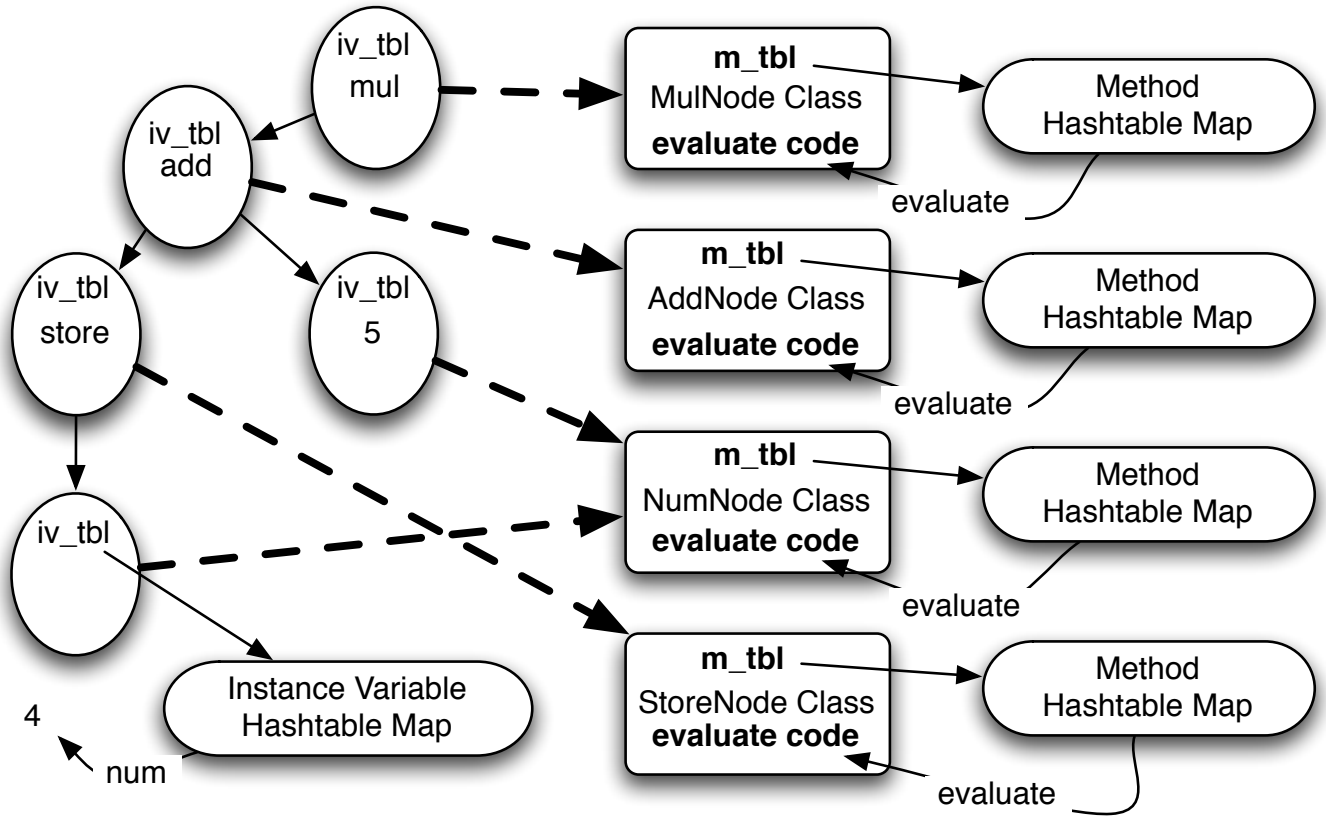
Write the NumNode class for AST nodes that contain numbers.

▸ View Solution

Polymorphism in Ruby

- Ruby is a dynamically typed language.
- This means that types are not determined at compile-time, since Ruby is not compiled.
- Checking types at run-time slows down the execution of the program, although not significantly for most applications.
- The second problem relates to testing code. If a program has a type error, run-time type checking doesn't detect it until the program evaluates the offending expression in the code.
- Consider the abstract syntax tree in figure 2 on page 21.
- To evaluate the tree, the `evaluate` method is called on the root's `MulNode` object.
- How does the correct evaluate get called?

Ruby object organization



Polymorphism in Ruby

- Figure 2 depicts the organization of objects and classes within Ruby.
- Each object in Ruby contains a pointer to its corresponding class (the bold dashed lines in the figure).
- Within each object is a field called `iv_tbl`. This field points to a hash table, one per object, that points to the instance variables of the object.
- Each time evaluate is called on an object, the `m_tbl` hash table maps the name `"evaluate"` to the code that implements it.

Polymorphism in Ruby

👉 Practice 4.5

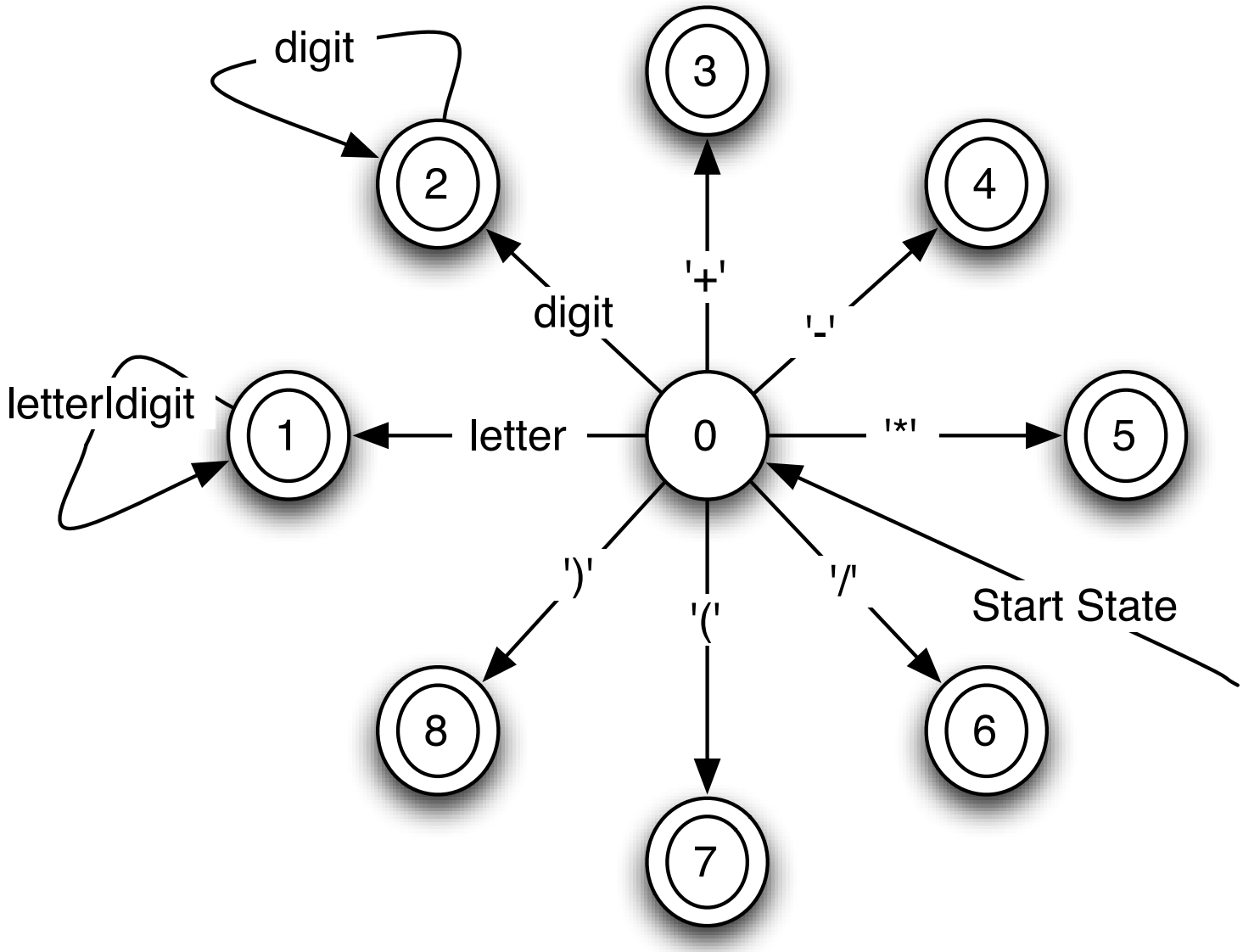
Recalling that every value in Ruby is an object, draw a picture of the store node object shown in figure 2 given what you now know about objects in the Ruby model. In your picture draw all the objects that the store node refers to in this example. Be careful when you do this. Remember that every value is an object in Ruby.

▸ View Solution

The Scanner

- Referring back to figure 5 the scanner reads characters from the input and builds Token objects that are used by the parser.
- The parser will get tokens from the scanner by calling a getToken method.
- Sometimes the parser gets a token and needs to put it back to get again later.
- In that case a putBackToken method will put back the last token that was returned by getToken.

A Finite State Machine for the Scanner



The Scanner

- Internally, the scanner object is a finite state machine.
- A finite state machine (fsm) consists of a set of states and a set of transitions from one state to another based on the current character in the input.
- The fsm starts in state zero, reads one character and transitions to one of the eight states depending on the character.
- An fsm is a model of computation for recognizing strings of characters.
- Fsm's are used in many contexts including network protocol implementations, pattern recognition, simulations, and of course language implementation.

The Parser

- Figure 5 shows the parser reading tokens and producing an abstract syntax tree as its output.
- The parser that is discussed in this section is a top-down parser.

The Parser

Example 4.7

This is the Calculator language's grammar.

$$Prog \rightarrow Expr \;\; EOF$$

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$

$$Term \rightarrow Term * Storable \mid Term / Storable \mid Storable$$

$$Storable \rightarrow Factor \;\; S \mid Factor$$

$$Factor \rightarrow number \mid R \mid (Expr)$$

The Parser

- The implementation of the parser is given to us by its grammar.
- In the implementation, each nonterminal becomes a function in the parser.
- Each rule in the grammar is part of a function that is named by the nonterminal on the left side of the arrow in the rule.
- In the grammar above each line would correspond to a function in the parser.
- Each appearance of a nonterminal on the right hand side of a production is a function call.
- Each appearance of a token on the right hand side of a production is a call to the scanner to get a token.

A First Attempt at Writing the Parser

Example 4.8

The Prog and Expr functions for the Parser.

```
1     def Prog()
2        result = Expr()
3        t = @scan.getToken()
4
5        if t.type != :eof then
6           print "Expected EOF. Found ", t.type, ".\n"
7           raise "Parse Error"
8        end
9
10       return result
11    end
12
13    def Expr()
14       ast = Expr()
15       t = @scan.getToken()
16       ...
17    end
```

A Better Attempt at Writing a Top-Down Parser

1. Eliminate left recursion.
2. Perform left factorization where appropriate.

A Better Attempt at Writing a Top-Down Parser

Example 4.9

An LL(1) Calculator Language Grammar

$Prog \rightarrow Expr\ EOF$

$Expr \rightarrow Term\ RestExpr$

$RestExpr \rightarrow +\ Term\ RestExpr \mid -\ Term\ RestExpr \mid <null>$

$Term \rightarrow Storable\ \ RestTerm$

$RestTerm \rightarrow *\ Storable\ RestTerm \mid /\ Storable\ RestTerm \mid <null>$

$Storable \rightarrow Factor\ S \mid Factor$

$Factor \rightarrow number \mid R \mid (Expr)$

Translating the LL(1) Grammar to Ruby

1. Construct a function for each nonterminal. Each of these functions should return a node in the abstract syntax tree.
2. Depending on your grammar, some nonterminal functions may require an input parameter of an abstract syntax tree (ast) to be able to complete a partial ast that is recognized by the nonterminal function.
3. Each nonterminal function should call getToken on the scanner to get the next token as needed. If after getting the token the code determines it didn't need the token after all, the nonterminal function should call the scanner's putBackToken function to put back the token. If the parser is based on an LL(1) grammar, it should never have to put back more than one token at a time.

4. The body of each nonterminal function is a series of if statements that choose which production to expand upon depending on the value of the next token. The body of the function is determined by the productions of the grammar with the given nonterminal on the left hand side of the arrow.

Translating the LL(1) Grammar to Ruby

Example 4.10

The Parser's Prog and Expr Functions

```
1     def Prog()
2        result = Expr()
3        t = @scan.getToken()
4
5        if t.type != :eof then
6           print "Expected EOF. Found ", t.type, ".\n"
7           raise "Parse Error"
8        end
9
10       return result
11    end
12
13    def Expr()
14       return RestExpr(Term())
15    end
```

Translating the LL(1) Grammar to Ruby

Practice 4.6

The RestExpr function is slightly different from the Prog and Expr functions. The RestExpr function has an AST parameter which we'll call `e`. The RestExpr function first gets a token and then decides what to do based on that token. If it is an **add** token it builds a new AST AddNode with the part of the tree given to it (i.e. **e**) as the left subtree and the result of calling Term as the right subtree. The subtract AST nodes are handled similarly. Otherwise, there wasn't a token that the RestExpr knows about so the token is put back and the AST **e** is returned as its AST.

Write the RestExpr function described here. Remember you can refer to the grammar in example 4.9.

▸ View Solution

Putting It All Together

- One more class is required to tie together the pieces that have been developed in this chapter.
- The Calculator class contains a memory location that can hold a stored value.
- The value can also be retrieved on demand.
- The calculator can evaluate an expression that is given to it as a string.

Putting It All Together

Example 4.11

The Calculator class implementation

```
1  class Calculator
2     attr_reader :memory
3     attr_writer :memory
4
5     def initialize()
6       @memory = 0
7     end
8
9     def eval(expr)
10       parser = Parser.new(StringIO.new(expr))
11       ast = parser.parse()
12       return ast.evaluate()
13    end
14 end
```

Putting It All Together

Example 4.12

The code to start it all.

```
1  text = gets
2  $calc = Calculator.new()
3
4  puts "The result is " + $calc.eval(text).to_s
```

Static vs Dynamic Type Checking

- In this chapter we have learned how to design and implement a calculator in Ruby.
- In the last chapter the same project was tackled in C++.
- While the two projects have many similarities, there are important differences between them as well.
- The primary difference between the two projects stems from the way type checking is handled in the two languages and how polymorphism is implemented.

Static vs Dynamic Type Checking

Example 4.13

Here is the AST, BinaryNode, UnaryNode, and AddNode class declarations in C++.

```
1  class AST {
2   public:
3     AST();
4     virtual ~AST() = 0;
5     virtual int evaluate() = 0;
6  };
7
8  class BinaryNode : public AST {
9   public:
10    BinaryNode(AST* left, AST* right);
11    ~BinaryNode();
12
13    AST* getLeftSubTree() const;
14    AST* getRightSubTree() const;
15  private:
16    AST* leftTree;
17    AST* rightTree;
```

```
18  };
19
20  class UnaryNode : public AST {
21   public:
22     UnaryNode(AST* sub);
23     ~UnaryNode();
24
25     AST* getSubTree() const;
26   private:
27     AST* subTree;
28  };
29
30  class AddNode : public BinaryNode {
31   public:
32     AddNode(AST* left, AST* right);
33
34     int evaluate();
35  };
```

Static vs Dynamic Type Checking

- For this code to compile in C++ all nodes must inherit from a common ancestor.
- C++ requires this because for polymorphism to work the compiler needs to be able to locate the evaluate method in the **vtable** for each possible node in a tree.
- Contrast this to the way Ruby works.
- In the Ruby AST implementation there is no common ancestor.

Static vs Dynamic Type Checking

Example 4.14

Here is the equivalent code in Ruby. Notice there is no common ancestor of the AST classes.

```
1   class BinaryNode
2      attr_reader :left, :right
3
4      def initialize(left,right)
5         @left = left
6         @right = right
7      end
8   end
9
10  class UnaryNode
11     attr_reader :subTree
12
13     def initialize(subTree)
14        @subTree = subTree
15     end
16  end
17
```

```
18  class AddNode < BinaryNode
19      def initialize(left, right)
20          super(left,right)
21      end
22
23      def evaluate()
24          return @left.evaluate() + @right.evaluate()
25      end
26  end
```

Static vs Dynamic Type Checking

- Ruby does not check the types of expressions in the program before executing the code.
- This isn't necessary in Ruby because all methods are looked up via a hash table at run-time as described in this chapter.
- Since no type checking is done prior to executing the code, when evaluate is called to compute the value represented by an AST, the right *evaluate* methods are looked up at run-time and the correct code gets called.

Static vs Dynamic Type Checking

- The difference in how polymorphism is implemented in C++ and Ruby has some pretty big consequences.
- The Ruby code is substantially shorter than the C++ code and it is certainly more convenient to write the Ruby program since all the extra classes and syntax aren't required.
- However, errors in type won't show up in a Ruby program until the program executes the code with the error in it.
- Static typing insures that most type errors are found when the program is compiled.
- Dynamic typing requires less coding but means that errors may not be found until run-time.

Exercises

1. What's the value of (R+7)/4S if the memory contained 4 prior to evaluating this expression?
2. What is the value of the memory location after evaluating the previous expression?
3. What does the abstract syntax tree look like for the expression (R+7)/4S?
4. How could the calculator language be modified to allow more than one memory location like modern calculators? Discuss what changes would be required to implement this enhanced calculator language.

5. Complete the calculator interpreter by downloading the code given in this chapter and finishing the implementation of the parser and the AST in the calc file. The rest of the project is provided.

   When you download the code you will want to unzip the package with some sort of unzip program. On Linux you can issue the command,

```
1  unzip rubcalc.zip
```

   Then you can make the program and run it. Here is an example of making and running you can use to get started.

```
1  $ unzip cppcalc.zip
2  $ cd rubycalc
3  $ calc
4  Please enter a calculator expression: 5 + 4
5  The result is 9
6  $
```

Commands that you enter are preceded by a dollar sign. The program will add two integers together as provided. Your job is to extend the project to the full calculator language. This requires changes to the parser and ast modules. The parser changes are highlighted in section 11. You can complete the parser by completing the functions that are incomplete in the parser class. These functions can be patterned after the code presented in the chapter.

The parser code will require that you build AST nodes for storing values and for recalling values from the calculator's memory. You will also need multiply and divide nodes in the abstract syntax tree. These new node types can be added to the AST code using the existing code as a pattern.

The store and recall nodes in the AST will need to access the memory location of the calculator. The global variable

called **$calc** can be used to access the calculator's memory. The line

```
1  $calc.memory = 6
```

will store 6 in the calculator's memory. Similarly, the expression $calc.memory will retrieve the value stored in the memory of the calculator.

6. Once you have completed the project described above extend the calculator language to allow more than one memory location to hold a value.
7. Modify the project to be a compiler instead of an interpreter. Instead of evaluating the expression, generate EWE code for it instead.

   In addition, to make this interesting, add a new keyword to the language, called *I*, that when executed waits for user input before proceeding. The value returned by the call to *I* is the value entered at the keyboard.

This project can be implemented with a few modifications. First, the eval function of the abstract syntax tree will print code to a file called *a.ewe* instead of directly evaluating the expression. The web page for the text contains a link to code to start this modified project. Remember, you are now printing code in this project and not evaluating. The EWE interpreter is evaluating the code.

For this project to work well you should decide on a model of computation for the generated EWE code to follow. A stack makes a nice model. When you generate EWE code for an expression, the resulting value should always be left on the top of a stack that you simulate using the EWE interpreter. That way, you can always find a value when you need it. Consult the code provided on the web site to see how this stack is simulated. The code provided has enough of the compiler implemented to add two integers together.

Solution to Practice Problem 4.1

1. 81
2. 6
3. Depends on the initial value of memory. It could be an error. Assuming the calculator starts with 0 in memory the answer would be 0. If the calculator is written to evaluate more than one expression in a session then the memory might contain the last value stored.
4. 10

Go Back to Practice Problem

Solution to Practice Problem 4.2

1. There are number, keyword, add, keyword tokens in this one.
2. The tokens are: lparen, number, add, number, rparen, keyword, times, keyword.

Go Back to Practice Problem

Solution to Practice Problem 4.3

The integer object that `x` refers to (i.e. the 5) is not mutated. The reference `x` is changed to point to a new object, the result of adding 5 and 1.

▸ Go Back to Practice Problem

Solution to Practice Problem 4.4

This is the NumNode implementation. Notice it does not inherit from anything because it is a leaf node and not a UnaryNode or BinaryNode. Inheritance is only used for code reuse in Ruby. It is not needed for polymorphic type checking.

```
1  class NumNode
2      def initialize(num)
3          @num = num
4      end
5
6      def evaluate()
7          return @num
8      end
9  end
```

▸ Go Back to Practice Problem

Solution to Practice Problem 4.5

This is how Ruby objects are organized in memory. This is still only a sampling of the memory organization. There are details omitted because there is too much to display the complete organization of even these two objects. It should give you a good idea of the organization with Ruby though.

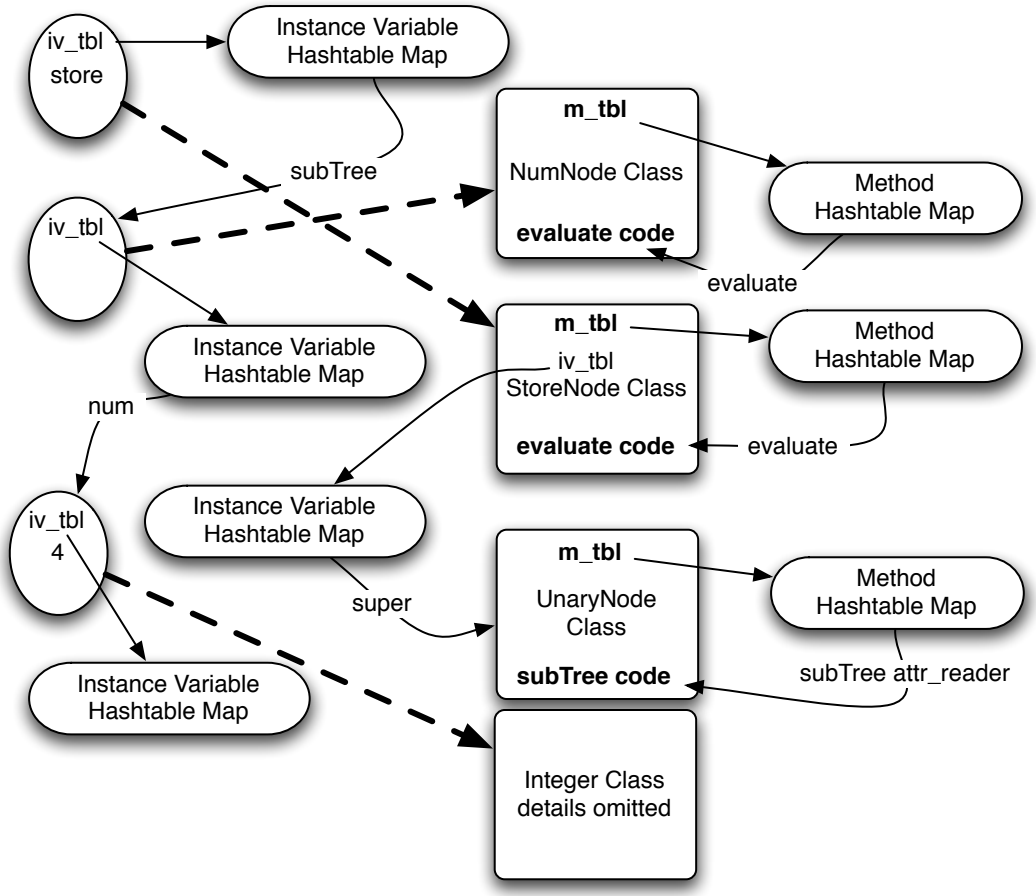


Figure: Ruby object organization

▸ Go Back to Practice Problem

Solution to Practice Problem 4.6

This is the RestExpr implementation.

```
1     def RestExpr(e)
2        t = @scan.getToken()
3
4        if t.type == :add then
5           return RestExpr(AddNode.new(e,Term()))
6        end
7
8        if t.type == :sub then
9           return RestExpr(SubNode.new(e,Term()))
10       end
11
12       @scan.putBackToken()
13
14       return e
15    end
```

▸ Go Back to Practice Problem