# Chapter 2

# Getting Started

*The secret of getting ahead is getting started. — Mark Twain*

Our goal in this chapter is to become familiar with some basic ideas in a programming language, e.g. grammar, lexical analysis, parsing, abstract syntax tree, etc. This chapter also serves to familiarize the reader with the design ideas e.g. data abstraction, design for change, the visitor design pattern that permeate throughout the programming language implementations discussed in this book.

## 2.1 Arithlang: An Arithmetic Language

We will get started by designing and implementing a simple programming language, *Arithlang*. Goal of Arithlang is to allow its programmer to write programs that can perform arithmetics. For example, a user of this programming language might write the following programs.

1. 1
2. (+ 1 1)
3. (- (+ 2 2) 1)
4. (* (+ 1 1) (- 3 1))
5. (/ (* (- 6 1) 2) (+ 1 1))

These users can then expect such programs to produce values 1, 2, 3, 4, and 5 respectively. Notice that the programs in Arithlang are written in a *prefix* notation, as opposed to the *infix* notation that languages like Java use. In a prefix notation, operators such as +, -, *, and / appear before their operands. To be unambiguous, we delineate start and end of an arithmetic expression with '(' and ')'. The symbols +, -, *, and / represent addition, subtraction, multiplication and division respectively.

There are some distinct advantages to writing in a prefix notation, e.g. one can easily express operations that apply over multiple operands. For example, arithmetic expression 2 + 4 + 6 + 5 can be written as (+ 2 4 6 5), which uses only one operator. Similarly, we can write 10 * 2 * 7 * 3 as (* 10 2 7 3), 5 - 4 - 1 as (- 5 4 1), and 6 / 3 / 2 as (/ 6 3 2).

### Exercise

2.1.1. Convert following arithmetic expressions written in an infix notation to the prefix notation of Arithlang.

1. 2 + 3
2. 2 + 3 * 4
3. 5 * ( 6 + 4)
4. (5 - 1) / ( 2 * 2)
5. (2 + 3) * (6 - 1) / (7 - 2)

2.1.2. Write five different arithmetic expressions using the prefix notation of Arithlang such that each expression produces the value 342. Each expression must use all four arithmetic operators +, -, *, and /.

The objective of a valid implementation of this programming language is: *given a program compute the value of the arithmetic expression represented by that program.* Your simple calculator, if it supports prefix notation, is a valid implementation of such programming language.

## 2.2 Legal Programs

Not every text can be considered a valid program in this language. We need some laws to distinguish valid programs from invalid ones. Moreover these

laws can also serve as guidance for creating new programs in this programming language. Such laws are encoded as the grammar of a programming language.

There are several possible notations for describing grammars depending on the tools used in constructing the implementation of the programming language. This book follows the extended backus-naur form (EBNF) notation. The grammar itself follow the format used by ANTLR (`www.antlr.org/`), a tool for constructing language implementations.

In this format, a grammar can begin with the following declaration:

```
1 grammar ArithLang;
```

which says that we are defining a grammar and giving it the name `ArithLang`. There are other kinds of grammar that ANTLR supports, but that does not concerns us at the moment.

The simplest kind of arithmetic expressions are numbers. We can allow such numbers to be valid programs in our programming language.

```
1 grammar ArithLang;

3 program : Number ;
4 Number :
5     DIGIT
6     | (DIGIT_NOT_ZERO DIGIT+);

8 DIGIT: ('0' .. '9');
9 DIGIT_NOT_ZERO: ('1'..'9');
```

Line 3 of this grammar says that a program (`program`) can be a number (`Number`). Here onwards in this book we will read `x : y` in grammars as "x can be a y" or sometimes "x can be expanded to y". This is an example rule in this grammar. Each rule terminates with a semi-colon (`;`).

Next rule defines legal numbers. It says that a number can be a digit (`DIGIT`) or (`|`) a non-zero digit followed by one or more digits (represented as `DIGIT+`). This is an example of specifying alternatives in grammar rule. The grammar also defines a digit to be a character between 0 and 9 in the rule `DIGIT`, and similarly non-zero digits.

According to this grammar "0" is a valid program, and so is "1", and so on until "9". This is because "0" - "9" are valid digits, and since number

can be a digit “0” - “9” are valid numbers, and since a program can be a number “0” - “9” are valid programs.

According to this grammar “10” is a valid program. This is because “1” is a non-zero digit, and since a number can be a non-zero digit followed by one or more digits (according to the rule on line 6), and since “0” is another digit “10” is a valid number. Since a program can be a number, and “10” is a valid number, therefore “10” is a valid program.

Unlike previous valid programs, “01” is not a valid program. This is because there is no such rule in this grammar where ‘0’ can be followed by any other digit. Similarly, “001” is also not a valid program and we can determine that by following the same reasoning process.

Numbers are nice programs, but in order to fulfill our objective we must be able to support more complex programs in Arithlang. We can extend our grammar as shown in figure 2.1 to allow such programs.

We have seen the rules for number and digits previously. The rule for a number expression (numexp) says that a numexp can be a number. As a result of this rule, we can still give an argument about why “0” - “9” are valid programs, and “10” is a valid program, but “01” is not a valid program. The argument, however, gets just a bit lengthy. To argue that “0” is a valid program we will first observe that a program can be an expression (exp), an expression can be a number expression (num-exp), a number expression can be a Number, and a Number can be Digit, and since “0” contains digit ‘0’ only we do not need to apply any other rule, therefore “0” is a valid program.

The rule for program is now extended to allow several kind of arithmetic expressions. The rule now says that a program can be a expression (abbreviated as exp). An expression can be either a number expression (numexp), an addition (addexp), a subtraction (subexp), a multiplication (multexp), or a division (divexp) expression.

Later rules define what are legal addition, subtraction, multiplication and division expressions.

Each of these rules have a similar structure, so let us focus on the rule for addition expression (addexp). This rule says that an addition expression starts with an open parenthesis character ‘(’ followed by a ‘+’ character, and it ends with a close parenthesis character ‘)’. The middle section of this rule is important in allowing an addition expression to be formed out of subexpressions. It says that an addition expression can contain an expression followed by *one or more* expressions. The notation (X)+ represents

```
1  grammar ArithLang;

3  program : exp ;

5  exp :   numexp
6          | addexp
7          | subexp
8          | multexp
9          | divexp  ;

11 numexp : Number ;

13 addexp : '(' '+' exp (exp)+ ')' ;

15 subexp : '(' '−' exp (exp)+ ')' ;

17 multexp : '(' '∗' exp (exp)+ ')' ;

19 divexp  : '(' '/' exp (exp)+ ')' ;

21 Number :
22     DIGIT
23     | (DIGIT_NOT_ZERO DIGIT+);

25 DIGIT: ('0' .. '9');
26 DIGIT_NOT_ZERO: ('1'..'9');
```

Figure 2.1: Grammar for the Arithlang Language

one or more of X. This definition allows us to write programs like (+ 0 0 0). From our previous discussion we know that "0" is a valid expression of kind num-exp. Since a program can be an addition expression, and an addition expression can be '(' '+' exp (exp)+ ')', if we substitute first exp with a valid expression we will continue to get a program, i.e. '(' '+' 0 (exp)+ ')' continues to be a valid program. Think of occurences of exp in the addition expression above as a placeholder, which can be replaced by any other valid exp. Since (exp)+ represents one or more of exp, we can replace it by two valid expressions and still continue to have a valid program, i.e. '(' '+' 0 exp exp ')' continues to be a valid program. By following our previous argument, we can substitute two later expressions with "0" and continue to have a valid program '(' '+' '0' '0' '0' ')' in Arithlang. We can write down this argument more systematically as shown in figure 2.2.

**Derivation** This process of creating an argument as to why a string is a valid program in a particular programming language is called a *derivation*. In the derivation above, at each step the emphasized symbol is replaced by its expansion. Some symbols are never expanded, e.g. '0', or '+'. Also notice that at each step the left-most symbol is expanded. This kind of derivation is known as a leftmost derivation.

**Terminals and Non-terminals** In grammars those symbols that can be expanded are called non-terminals, whereas those that are leaf nodes and cannot be expanded are called terminals. In our grammar for Arithlang, program, exp, numexp, addexp, subexp, multexp, divexp are non-terminals. Special characters like '(', ')', as well as digits are terminals.

Other rules for subtraction, multiplication and division arithmetic expression can be similarly understood.

## Exercise

2.2.1. Write down the leftmost derivations for the following programs in Arithlang.

1. 3
2. (+ 3 4)
3. ( + (+ 3 4) 2)

1. *program* →
2. *exp* →
3. *addexp* →
4. '(' '+' *exp* (exp)+ ')' →
5. '(' '+' *numexp* (exp)+ ')' →
6. '(' '+' *Number* (exp)+ ')' →
7. '(' '+' *DIGIT* (exp)+ ')' →
8. '(' '+' '0' *(exp)+* ')' →
9. '(' '+' '0' *exp* (exp)+ ')' →
10. '(' '+' '0' *numexp* (exp)+ ')' →
11. '(' '+' '0' *Number* (exp)+ ')' →
12. '(' '+' '0' *DIGIT* (exp)+ ')' →
13. '(' '+' '0' '0' *(exp)+* ')' →
14. '(' '+' '0' '0' *exp* ')' →
15. '(' '+' '0' '0' *numexp* ')' →
16. '(' '+' '0' '0' *Number* ')' →
17. '(' '+' '0' '0' *DIGIT* ')' →
18. '(' '+' '0' '0' '0' ')'

Figure 2.2: A Leftmost Syntax Derivation

4. ( + (* 3 4) (- 2 0))
5. ( / 2 (- 4 3) )

2.2.2. A rightmost derivation is where the right-most available non-terminal is expanded first. Write down the rightmost derivations for the programs in the previous exercise. For example, the rightmost derivation for (+ 1 2) is:

1. program →
2. exp →
3. addexp →
4. '(' '+' exp (exp)+ ')' →
5. '(' '+' exp exp ')' →
6. '(' '+' exp numexp ')' →
7. '(' '+' exp Number ')' →
8. '(' '+' exp DIGIT ')' →
9. '(' '+' exp 2 ')' →
10. '(' '+' numexp 2 ')' →
11. '(' '+' Number 2 ')' →
12. '(' '+' DIGIT 2 ')' →
13. '(' '+' 1 2 ')'

2.2.3. Arithlang grammar supports positive integers. Extend this grammar to support negative integers as well as double values.

## Arithlang grammar, more formally

In previous section, we saw one realization of the Arithlang grammar — using the ANTLR syntax. Going forward, we will present our grammar rules independent of the concrete ANTLR syntax. The ANTLR grammar from figure 2.1 is restated in figure 2.3.

This form, especially the documentation to the right, makes it easier to identify rules. Like the concrete ANTLR form, in this form `Program`, `Expression`, `Number`, `Digit`, and `DigitNotZero` are nonterminals and `(`, `)`, `+`, `-`, `*`, `/`, `0`, ..., `9` are terminals.

| | | | |
|---|---|---|---|
| `Program` | ::= | `Exp` | *Program* |
| `Exp` | ::= | | *Expressions* |
| | | `Number` | *Number (NumExp)* |
| | \| | `(+ Exp Exp`$^+$`)` | *Addition (AddExp)* |
| | \| | `(- Exp Exp`$^+$`)` | *Subtraction (SubExp)* |
| | \| | `(* Exp Exp`$^+$`)` | *Multiplication (MultExp)* |
| | \| | `(/ Exp Exp`$^+$`)` | *Division (DivExp)* |
| `Number` | ::= | `Digit` | *Number* |
| | \| | `DigitNotZero Digit`$^+$ | |
| `Digit` | ::= | `0` \| `DigitNotZero` | *Digits* |
| `DigitNotZero` | ::= | `1` \| `2` \| `3` \| `4` \| `5` | *Non-zero Digits* |
| | | \| `6` \| `7` \| `8` \| `9` | |

Figure 2.3: Grammar for the Arithlang Language (restated)

## 2.3 Two Possibilities for Implementation

Now that we have a specification of legal programs in Arithlang we can create an implementation, one that can take an Arithlang program and compute its value.

We can realize this programming language in one of two ways. Either we can build a *compiler*. A compiler is a program that reads the *source program* in our new programming language that we are designing (*source language*) and translates into another program in a different language, lets call it the *target language*, such that when the resulting program in target language runs it will perform the computation that the source program is intended for. Alternatively we can build an *interpreter*, a program that reads the source program and immediately performs the computation that the source program is intended for.

The programming language used to implement the compiler or the interpreter is called the *defining language*. The programming language that is being implemented is called the *defined language*.

There are pros and cons to both kinds of implementation techniques. For example, interpreter-based techniques tend to be simpler so they are easier to implement. On the other hand, since they often repeat certain steps every time a program is run, e.g. reading a program and converting it into abstract syntax trees, so they can be inefficient at times.

In this book since our primary purpose is to focus on understanding and modeling programming language concepts, we will favor simplicity and realize our programming languages as interpreters.

An interpreter is a program that runs a loop that reads new programs from the user (Read), evaluates the value of those programs (Eval), and prints those values (Print). So the main component of an interpreter is also sometimes referred to as the REPL (Read-Eval-Print-Loop).

## 2.4 Reading Programs

A program is provided to an interpreter as a string. Main objective of the read phase is to convert this string into a more organized form known as the *abstract syntax tree* so that the later Eval phase can compute its value. This step usually consists of:

- dividing the program string into symbols, called *tokens*,
- organizing the tokens according to the rules of the programming language grammar into a tree structure called the *parse tree*, and
- converting the parse tree to an *abstract syntax tree (AST)* by ignoring the tokens irrelevant for evaluating the program.

First step is called lexical analysis. Legal tokens in a program are also prescribed by its grammar. For example, only legal token in the program "3" is the digit 3, whereas legal tokens in the program "(+ 3 4)" are '(', '+', the digit 3, the digit 4, ')' in that order.

Second step is called syntax analysis or more commonly parsing. Parsing is the process of automatically creating a syntax derivation. Output of parsing is a parse tree. figure 2.4 shows an example parse tree. Blocks in this tree are terminals and nonterminals and lines between blocks can be read as "expands-to" or "made-up-of". For instance, a program expands to an exp, which expands to an addexp, which expands to the terminal '(' followed by terminal '+' followed by a numexp, followed by another numexp, followed by a terminal ')'.

In the third step, irrelevant concrete tokens are eliminated to create a more abstract representation of the program. An abstract syntax tree for the example program is shown in figure 2.5. Notice that tokens that

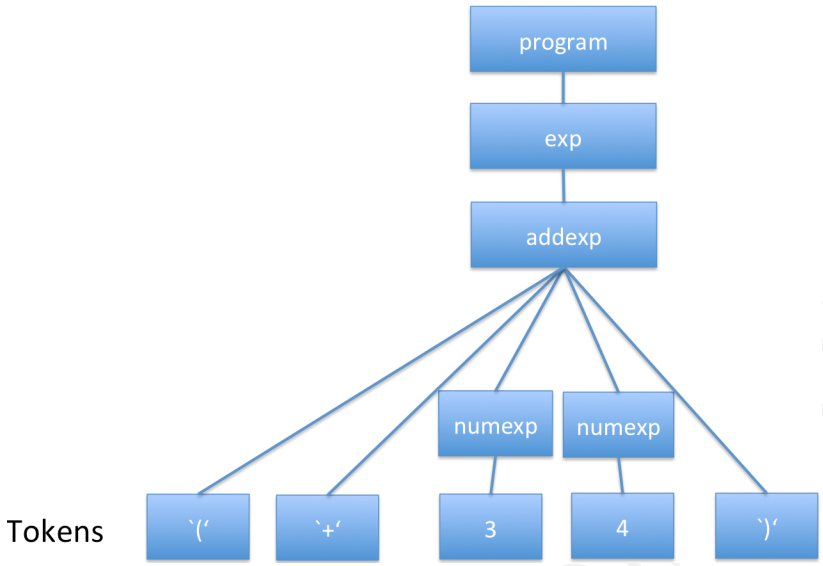


Figure 2.4: A parse tree for the Arithlang program “( + 3 4)”

belong to the concrete syntax such as ‘(’, ‘+’, ‘)’, and whitespace are not represented in this abstract syntax tree[1].

## 2.5 Storing Programs

The output of the read phase of an interpreter is typically an abstract syntax tree (AST).

In this section, we consider one possible AST representation for Arithlang. We derive this object-oriented representation from the grammar of Arithlang. In an abstract form, i.e. after ignoring concrete tokens like ‘(’, ‘+’, ‘-’, ‘*’, ‘/’, ‘)’, etc..., the grammar of Arithlang in figure 2.3 can be thought of as *entities* and their relationships shown in figure 2.6.

A program consists of an expression, represented as the relation “has” between `Program` and `Exp` entity nodes. Expressions can be either numeric

[1] The determination that a token can be eliminated is entirely dependent on the intended usage of the abstract syntax tree. For example, if the abstract syntax tree is to be used for code transformations that must preserve certain syntactic elements, we would need to design it differently to preserve whitespace, line breaks, and tabs.

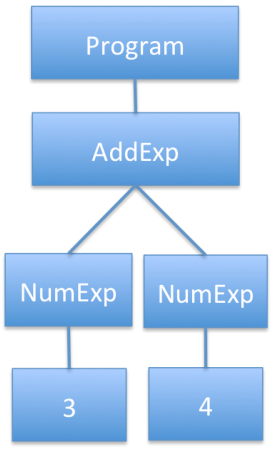


Figure 2.5: An abstract syntax tree for program ( + 3 4)

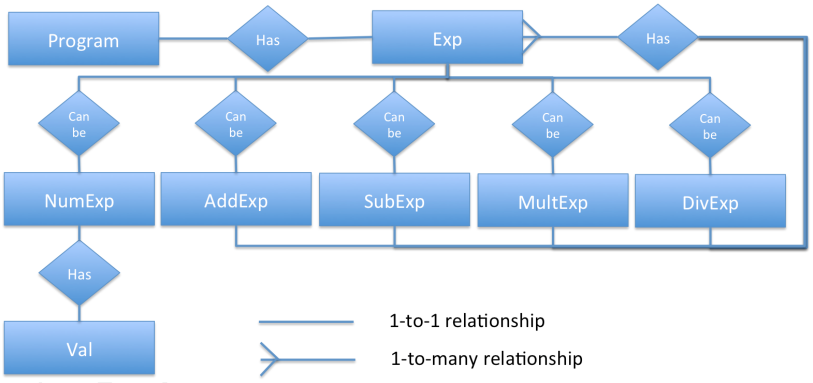


Figure 2.6: An Entity Relationship (ER) Diagram for the Arithlang Grammar in figure 2.1

expression, addition, subtraction, multiplication and division. This is represented as the relation "can be" between `Exp` and `NumExp`, `AddExp`, `SubExp`, `MultExp`, and `DivExp` entities. Furthermore, since each of the last four kind of expressions (`AddExp`, `SubExp`, `MultExp`, and `DivExp`) can have one or more expressions as its subexpression, we have a 1-to-many "has" relation between each kind of expression and the `Exp`. The `NumExp` expression has a numeric value.

In the object-oriented representation, it would be convenient to deal

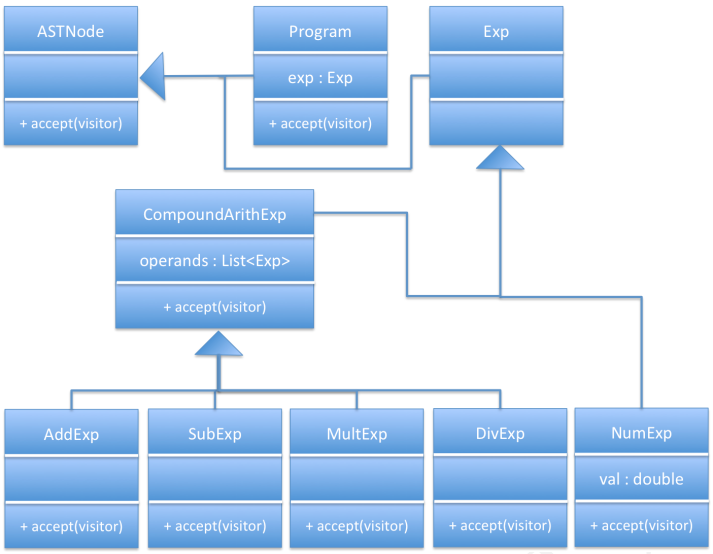


Figure 2.7: A UML diagram for the abstract syntax tree (AST) hierarchy designed to store Arithlang programs as defined by the grammar in figure 2.1. Concrete code is in figure 2.8, and figure 2.9

with each entity uniformly, i.e. `Program`, `Exp`. So in that representation, shown in figure 2.7, we introduce a top-level class `ASTNode`.

Both programs and expressions are AST nodes, so classes `Program` and `Exp` are designed to be subclasses of the class `ASTNode`. This class provides only one abstract method `accept` for the Arithlang implementation. This method is part of the implementation of the visitor pattern that we will frequently use to examine abstract syntax trees. We will discuss this pattern in greater detail in section 2.6. Concrete implementation of classes `ASTNode`, `Program`, and `Exp` is shown in figure 2.8.

Since there are five kinds of expression in Arithlang as defined by its grammar, we can create a subclass of Exp for each kind so that the subclass can implement the behavior specific to that kind of expression. Each of these kind needs to store information about subparts of that expression. For example, the `NumExp` expression in figure 2.8 ought to store actual value of number for later use in program evaluation, printing, etc. Similarly, an addition expression "(+ 3 4 2)" would need to store subexpressions "3", "4", "2", and a multiplication expression "(* 3 (+ 4 2))" would need to store subexpressions "3" and "(+ 4 2)" for later use.

```
1  class ASTNode {
2    public abstract Object accept( Visitor  visitor );
3  }
4  class Program extends ASTNode {
5    Exp _e;
6    public Program(Exp e) { _e = e; }
7    public Exp e() { return _e; }

9    public Object accept( Visitor  visitor ) {
10     return visitor . visit (this);
11   }
12 }
13 abstract class Exp extends ASTNode { }
14 class NumExp extends Exp {
15   double _val;
16   public NumExp(double v) { _val = v; }
17   public double v() { return _val ; }

19   public Object accept( Visitor  visitor ) {
20     return visitor . visit (this);
21   }
22 }
```

Figure 2.8: Concrete implementation of ASTNode, Program, Exp, and NumExp

One straightforward design of these classes representing expression nodes in the abstract syntax tree would have the `AddExp` class maintain a list of argument expressions and operations to manipulate that list. However, notice from the grammar that addexp, subexp, multexp, and divexp all have a similar structure in terms of subexpressions. Each expression can have 1 or more subexpressions as operands. By realizing this commonality we can improve our initial object-oriented design. We can allow code reuse if we can create a subclass of Exp that implements commonalities, and then create four subclasses that implement expression-specific variabilities. The object-oriented design shown in figure 2.7 utilizes this strategy. An

implementation of this design strategy is shown in figure 2.9.

```
23  abstract class CompoundArithExp extends Exp {
24    List<Exp> _rep;
25    public CompoundArithExp(List<Exp> args) {
26      _rep = new ArrayList<Exp>();
27      _rep.addAll(args);
28    }
29    public List<Exp> all() {
30      return _rep;
31    }
32  }
33  class AddExp extends CompoundArithExp {
34    public AddExp(List<Exp> args) {
35      super(args);
36    }
37    public Object accept( Visitor visitor ) {
38      return visitor . visit (this);
39    }
40  }
```

Figure 2.9: Concrete implementation of CompoundArithExp and AddExp

Implementation of other expressions `SubExp`, `DivExp`, and `MulExp` are similar to `AddExp`.

In later chapters, we will not be showing the object-oriented design such as that in figure 2.7, but leaving it as an exercise for the reader. The reader is also encouraged to consider pro and cons of merging the functionality of `CompoundArithExp` with `Exp` in terms of code reuse and storage.

## Exercise

2.5.1. Write an alternate implementation for abstract syntax tree for Arithlang, one in which `CompoundArithExp` stores a field of enum type `OperatorKind` modeling the concrete operator with four values `Add`, `Sub`, `Mult`, and `Div`.

2.5.2. Extend the implementation of abstract syntax tree for Arithlang to include an AST node for negation expression "- Exp".

## 2.6 Analyzing Programs

Recall from section 2.3 that an interpreter runs a three-step loop (Read-Eval-Print Loop or REPL). First step read consumes a program and produces an abstract syntax tree representing the program. In previous section, we learned about one possible realization of abstract syntax tree. Next step eval consumes an abstract syntax tree and produces its value. This process is much like tree traversal that you may have come across in courses on data structures, where the value of an expression depends on its subexpression. For example, the value of a multiplication expression "(* 3 (+ 4 2))" would depend on value of subexpressions "3" and "(+ 4 2)".

Program evaluation is not the only task that requires abstract syntax tree traversal. If you have ever used the code formatter of your integrated development environment (IDE) you have come across another consumer of AST that produces formatted string.

There are two design strategies for realizing functionalities that consume an abstract syntax tree, and require traversing it. In the first strategy, we can extend the implementation of each class representing AST nodes to implement the functionality. For example, classes corresponding to constant, addition, subtraction, multiplication, and division are all extended to implement code formatting, evaluation, etc. In the second strategy, we can extend the implementation of each class representing AST nodes to implement a generic traversal functionality. Concrete traversal strategies can extend the generic behavior.

In the first strategy, implementation of the evaluation behavior (evaluation concern) is spread across implementation of abstract syntax tree related classes. Similarly, implementation of the formatting behavior (formatting concern) is spread across implementation of abstract syntax tree related classes. In other words, evaluation and formatting concerns are scattered across AST related classes and could be considered tangled with each other. That makes it harder to modify them, if needed. In the second strategy, this behavior is localized in one class which makes it easier to maintain it.

## Visitor Design Pattern

In this book, we will use the second strategy also known as the *visitor design pattern* in object-oriented program design. We will illustrate this pattern by implementing a code formatter.

Recall from the AST representation discussed in section 2.5 that each concrete class implemented a method `accept` that takes an object of type `Visitor` as a parameter and invokes method `visit` on that object. See figure 2.9 for an example. Concrete definition of the type `Visitor` is given in figure 2.10. This interface provides a method `visit` for each concrete AST node. Concrete AST traversal functionalities can be implemented by extending the `Visitor` interface. An example appears in figure 2.11.

```
1  interface Visitor <T> {
2    T visit (NumExp e);
3    T visit (AddExp e);
4    T visit (MultExp e);
5    T visit (SubExp e);
6    T visit (DivExp e);
7    T visit (Program p);
8  }
```

Figure 2.10: The Visitor interface

The purpose of code formatter is to print an abstract syntax tree. Since code formatter is a concrete class and it implements the `Visitor` interface, it must provide an implementation for each method declared in that interface. The listing in figure 2.11 shows some of those methods, others are elided because they are very similar to the visit method for `AddExp`.

The code formatter is an excellent example of how values for an AST node are built using values for its component AST nodes. Take `visit` method that takes a `Program` as a parameter as an example. Since in Arithlang language a program consists of an expression, the formatted string of a program is the same as the formatted string of its expression. Formatted string for the expression (`p.e()`) is computed by invoking the `accept` method with current visitor as the parameter. Take `visit` method that takes `AddExp` as a parameter as another example. Since in Arithlang an addition expression consists of one or more subexpressions, the formatted

```
1  class Formatter implements AST.Visitor<String> {
2    String visit(Program p) {
3      return (String) p.e().accept(this);
4    }
5    String visit(NumExp e) {
6      return "" + e.v();
7    }
8    String visit(AddExp e) {
9      String result = "(+";
10     for(AST.Exp exp : e.all())
11       result += (" " + exp.accept(this));
12     return result + ")";
13   }
14  ...
15 }
```

Figure 2.11: A code formatter: an example visitor

value of an addition expression is "(+" followed by formatted value of each subexpression, followed by ")".

## Control Flow of Visitor Pattern

The flow of a program using the visitor pattern can be a little difficult to understand at first due to extra indirections, but it helps to remember that (a) the logic for handling each kind of AST node is in corresponding `visit` method, and (b) calling the `accept` method is a way to jump back to the suitable `visit` method in the current class.

To illustrate, let `prog` be an AST object of type `Program` corresponding to the program "(+ 3 4 2)". We can manually construct this AST object as follows.

```
1  Exp exp3 = new NumExp(3);
2  Exp exp4 = new NumExp(4);
3  Exp exp2 = new NumExp(2);
4  List<Exp> expList = new ArrayList<Exp>();
5  expList.add(exp3);
```

```
6     expList.add(exp4);
7     expList.add(exp2);
8     AddExp addExp = new AddExp(expList);
9     Program prog = new Program(addExp);
```

We can then try to find the formatted form of this program as follows.

```
10    Formatter f = new Formatter();
11    prog.accept(f);
```

The call to method `accept` causes method `visit (Program p)` in class `Formatter` on line 2 in figure 2.11 to run. This method then invokes method `accept` on the object `addExp`, which in turn causes method `visit (AddExp e)` in class `Formatter` on line 8 to run. This method iterates over the component expressions `exp3`, `exp4`, and `exp2` and invokes method `accept` on each object. That in turn causes method `visit (NumExp e)` in class `Formatter` on line 5 to run three times returning result strings "3", "4", and "2" respectively. Consequently, return value of the method `visit (AddExp e)` is the string "(+ 3 4 2)". Therefore, the return value of the method `visit(Program p)` is also the string "(+ 3 4 2)", and as a result value of the expression `prog.accept(f)` is also the string "(+ 3 4 2)".

## Exercise

2.6.1. *[AST node counter]* Use the visitor design pattern to create another example visitor class, say `ASTCounter`, which counts the number of abstract syntax tree nodes. The methods in `ASTCounter` should not use any global or static variables. The `ASTCounter` class should not have any fields.

2.6.2. *[Number collector]* Use the visitor design pattern to create another example visitor class, say `NumberCollector` which collects all numbers that appear in the abstract syntax tree in a list. For example, three numbers, "3", "4", and "2" appear in the AST of the program "(+ 3 4 2)". The methods in `NumberCollector` also should not use any global or static variables. The `NumberCollector` class should not have any fields.

## 2.7 Legal Values

Output of a program in a programming language is defined and constrained by the legal values that it can produce. For Arithlang only legal values are numerical values. This is because all expressions in Arithlang (as defined by its grammar in figure 2.3) can only produce numeric values.

| | | | |
|---|---|---|---|
| `Value` | ::= | | *Values* |
| | | `NumVal` | *Numeric Values* |
| `NumVal` | ::= | `(NumVal n)`, where `n` $\in$ the set of doubles | *NumVal* |

Figure 2.12: The Set of Legal Values for the Arithlang Language

We can define this set of legal values as shown in figure 2.13, which says that only legal values are elements of `NumVal`. The `NumVal` is defined in terms of the double data type in the defining language. In other words, definition of Arithlang is not independent of the defining language, but for the moment we will tolerate this dependence in favor of simplicity.

**Notations** Here onward we will use a prefix declarative form for writing values, as opposed to using their imperative form. So `v = (NumVal 342)` is the same as `NumVal v = new NumVal(342)`. When we use the name of an abstract class such as `Value`, we mean that all of its subclasses can be used at that location. In other words, the relation is valid for all subclasses of that Value. If some name from the left hand side of equality matches that on the right hand side of equality they are the same object.

**Implementation** We can create data structures to realize this definition as shown in figure 2.13. A note for readers not familiar with Java: in the programming language it is possible to define *static* classes as part of an interface as shown on line 3 in figure 2.13. These classes are referred to by appending their containing interface name in front of their name, e.g. `Value.NumVal` to refer to the class `NumVal` in figure 2.13. This mechanism allows keeping related classes in a single compilation unit.

In the design of the value type hierarchy, we have created an interface `Value` as a supertype of all kinds of values. This decision allows design for change. In particular, we would like to allow other kinds of program values in future. So it made sense to create an interface so that we can

```
1  public interface Value {
2    public String toString();
3    static class NumVal implements Value {
4      private double _val;
5        public NumVal(double v) { _val = v; }
6        public double v() { return _val; }
7        public String toString() { return "" + _val; }
8    }
9  }
```

Figure 2.13: Values in Arithlang

protect clients of `Value` that do not need to refer to concrete kinds of values from the effect of adding new kinds of values. This concept, called *data abstraction* plays a significant role in both definition and implementation of programming languages.

## 2.8 Evaluating Programs to Values

To evaluate programs to values, we will follow a case-by-case recursive approach similar to formatter. We will realize this functionality as `Evaluator`, which is a kind of visitor, but first let us learn about this evaluation procedure more abstractly.

A program in Arithlang consists of an expression. So what is the value of a program? It is the value of this expression.

### Notations

Let `Program` be the set of all programs in Arithlang, `Exp` be the set of all expressions in Arithlang, and `Value` be the set of all values that can be produced by Arithlang programs. Also, let `p` be a program, i.e it is in set `Program` and `e` be an expression, i.e. it is in set `Exp` such that `e` is the inner expression of `p`.

Here onward we will use a prefix declarative form for writing AST nodes, as opposed to using their imperative form. So `p = (Program e)` is the same as `Program p = new Program(e)`. When we use the name of an abstract

class such as `Exp`, we mean that all of its subclasses can be used at that location. In other words, the relation is valid for all subclasses of that AST node. If some name from the left hand side of equality matches that on the right hand side of equality they are the same object.

**Value relation** We will write the statement "value of a program" more precisely as the following mathematical relation.

```
value :  Program -> Value
```

Here, think of `value` as a mathematical function that takes the program AST node as an argument, e.g. `p`.

We will write the statement "value of an expression" more precisely as the following mathematical relation.

```
value :  Exp -> Value
```

**Logical rules** In describing the intended semantics of programming languages, we will often need to make statements like "B is true when A is true". We will state this relation as follows.

$$\textsc{Relation BifA}$$
$$\frac{A}{B}$$

Similarly, the relation "C is true when both A and B are true" is stated as follows.

$$\textsc{Relation CifAandB}$$
$$\frac{A \quad B}{C}$$

We take the conjunction of the conditions above the line.

A relation "A is unconditionally true" is stated as follows.

$$\textsc{Relation Aunconditionally}$$
$$A$$

## Value of a Program

With these two relations in place, we can write the statement "value of a program `p` is the value of its inner expression `e`" more precisely as the following mathematical relation.

$$\frac{\texttt{value e = v}}{\texttt{value p = v}}\ \textsc{Value of Program}$$

You should read the relation above as follows: *the result of applying the function* ***`value`*** *on a program* ***`p`*** *with inner expression* ***`e`*** *is* ***`v`*** *if the result of applying the function* ***`value`*** *on the expression* ***`e`*** *is* ***`v`****.*

The implementation of `Evaluator` mimics this mathematical relation.

```
class Evaluator implements Visitor<Value> {
  Value valueOf(Program p) {
    // Value of a program is the value of the expression
    return (Value) p.accept(this);
  }
  ...
}
```

Clients of `Evaluator` will call the `valueOf` method to find values of the argument program `p`.

This method makes use of the visitor functionality to find the value of the program. Recall from before that the accept call will eventually transfer control to the case `Program` of the visitor.

```
Value visit (Program p) {
  return (Value) p.e().accept(this);
}
```

This case says that the value of the program is the value of the expression that forms the program. This accept call will eventually run a case for the expression depending on the type of the result `p.e()`. We will now examine each of these possibilities.

## Value of a Numeric Expression

First, the simplest case when the result turns out to be a `NumExp` expression (literal). A numeric expression is a leaf AST node in any Arithlang program,

it doesn't have any component subexpressions. The value of a numeric expression is an `NumVal` value.

VALUE OF NUMEXP
`value (NumExp n) = (NumVal n)`

Here, `n` is a double. As before, `value` is a mathematical relation from expression to values. The notation `(NumExp n)` is the same as `new NumExp(n)` and `Number` stands for all legal numbers.

Here are some applications of the relation above.

`value (NumExp 3) = (NumVal 3)`

`value (NumExp 4) = (NumVal 4)`

`value (NumExp 2) = (NumVal 2)`

The implementation of this case in `Evaluator` also models this relation.

```
Value visit (NumExp e) {
  return new NumVal(e.v());
}
```

In this case, we extract the numeric value from the expression, and then create a numeric value from that result. This models the semantics that the value of a literal is its numeric value, e.g. value of 10 is (NumVal 10).

## Value of an Addition Expression

Computing the value of an addition expression is different from literals because addition is a compound expression, whereas a numeric expression is a leaf expression.

To compute the value of a compound expression we must first compute the values of its component subexpressions. Recall that an addition expression can have two or more subexpressions, e.g. `(+ 300 40 2)`. Following mathematical relation specifies the value of an addition expression.

VALUE OF ADDEXP

$$\frac{\texttt{value } e_i \texttt{ = (NumVal } n_i\texttt{)}, \text{for i} = 0...\text{k} \qquad n_0 \texttt{ + ... + } n_k \texttt{ = n}}{\texttt{value (AddExp } e_0 \texttt{ ... } e_k\texttt{) = (NumVal n)}}$$

Here, $e_0$ ... $e_n$ are expressions, i.e. they are in set `Exp`. All notations have the same meaning as before. Recall that the condition below the line

holds if all conditions above the line hold. You should read this relation as: *if each of the component subexpressions* $e_i$ *evaluates to a numeric value* `(NumVal` $n_i$`)` *and the sum of the numbers* $n_0$ *to* $n_k$ *is* $n$*, then the value of the addition expression with* $e_0$ *to* $e_k$ *as subexpressions is* $n$*.*

Here is an application of the relation above.

```
value (AddExp (NumExp 4) (NumExp 2)) =
  value (NumExp 4) + value (NumExp 2)
```

Here is another application that recursively uses the same relation.

```
value (AddExp (NumExp 3) (AddExp (NumExp 4) (NumExp 2))) =
  value (NumExp 3) + value (AddExp (NumExp 4)(NumExp 2))
```

The implementation of this case in `Evaluator` also models this relation.

```
Value visit (AddExp e) {
  List<Exp> operands = e.all();
  double result = 0;
  for(Exp exp: operands) {
    NumVal interim = (NumVal) exp.accept(this);
    result += interim.v();
  }
  return new NumVal(result);
}
```

The implementation technique that we use here has semantic implications, i.e. it can change the intended meaning of addition in our programming language. For example, if an addition expression has three component subexpressions, we can compute their value at random, from first to last (which if you recall is equivalent to left to right textual order) or from last to first (equivalent to right to left textual order). Here, in our programming language we chose to evaluate subexpressions left to right, and sum up their values in an intermediate variable `result`. The final value stored in result is the value of this addition expression.

## Exercise

2.8.1. *[Semantic Variations]* Examine following different interpretations of the value of the addition expression. What effect could it have on value of programs?

- An addition expression evaluates its subexpressions at random.
- An addition expression treats any non-integral value (not an NumVal) obtained from evaluating any of its subexpressions as 0.

## Value of a Subtraction Expression

The value relation for a subtraction expression is also formulated in a manner similar to that of addition expression.

VALUE OF SUBEXP

$$\frac{\texttt{value e}_i\ \texttt{= (NumVal n}_i\texttt{)}, \text{for i} = 0...\text{k} \qquad \texttt{n}_0\ \texttt{-}\ \ldots\ \texttt{-}\ \texttt{n}_k\ \texttt{= n}}{\texttt{value (SubExp e}_0\ \ldots\ \texttt{e}_k\texttt{) = (NumVal n)}}$$

All notations have the same meaning as before. Here is an application of the relation above.

```
value (SubExp (NumExp 4) (NumExp 2)) =

  value (NumExp 4) - value (NumExp 2)
```

Here is another application that recursively uses the same relation.

```
value (SubExp (NumExp 3) (SubExp (NumExp 4) (NumExp 2))) =

  value (NumExp 3) + value (SubExp (NumExp 4)(NumExp 2))
```

The implementation of this case in `Evaluator` also models this relation.

```
public Value visit (SubExp e) {
  List<Exp> operands = e.all();
  NumVal lVal = (NumVal) operands.get(0).accept(this);
  double result = lVal.v();
  for(int i=1; i < operands.size(); i++) {
    NumVal rVal = (NumVal) operands.get(i).accept(this);
    result = result - rVal.v();
  }
  return new NumVal(result);
}
```

This implementation is only slightly different from the case for addition in that we first compute value of the first operand and then iteratively subtract other operands from this value.

### Value of Multiplication and Division Expressions

The value relations for multiplication and division expressions are also formulated in a manner similar to that of addition expression, but for completeness we reproduce them below.

VALUE OF MULTEXP

$$\frac{\texttt{value e}_i\ \texttt{= (NumVal n}_i\texttt{)}, \text{for i} = 0...\text{k} \qquad \texttt{n}_0\ \texttt{*}\ \ldots\ \texttt{*}\ \texttt{n}_k\ \texttt{= n}}{\texttt{value (MultExp e}_0\ \ldots\ \texttt{e}_k\texttt{) = (NumVal n)}}$$

VALUE OF DIVEXP

$$\frac{\texttt{value e}_i\ \texttt{= (NumVal n}_i\texttt{)}, \text{for i} = 0...\text{k} \qquad \texttt{n}_0\ \texttt{/}\ \ldots\ \texttt{/}\ \texttt{n}_k\ \texttt{= n}}{\texttt{value (DivExp e}_0\ \ldots\ \texttt{e}_k\texttt{) = (NumVal n)}}$$

As an exercise, reader can implement the cases for multiplication and division expressions in the Evaluator.

## 2.9 Read-Eval-Print Loop

Now since we have all the components of the interpreter, we can put them together in a simple read-eval-print loop and have the first working implementation of our Arithlang programming language.

```
public class Interpreter {
  public static void main(String[] args) {
    Reader reader = new Reader();
    Evaluator eval = new Evaluator();
    Printer printer = new Printer();
    try {
      while (true) { // Read-Eval-Print-Loop (also known as REPL)
        Program p = reader.read();
        Value val = eval.valueOf(p);
        printer.print(val);
      }
    } catch (IOException e) {
      System.out.println("Error reading program.");
    }
  }
}
```

We can play with this interpreter to see how it works. In the *interaction log* below $ is the prompt of the interpreter, text after $ is the program that the user writes and the text on the next line is the value of this program.

```
$ 3
3
$ (* 3 100)
300
$ (- 279 277)
2
$ (/ 84 (- 279 277))
42
$ (+ (* 3 100) (/ 84 (- 279 277)))
342
```

## Exercise

2.9.1. *[Modulus Expression]* Extend the Arithlang programming language to support modulus expression (% a b), also called remainder sometimes. Following interaction log illustrates the intended semantics of this expression.

```
$ (% 8 3)
2
$ (% 8 3 2)
0
```

2.9.2. *[Power Expression]* Extend the Arithlang programming language to support exponential operation (** a b). Following interaction log illustrates the intended semantics of this expression.

```
$ (** 2 4)
16
$ (** 3 2 4)
6561
$(** 8 0)
1
```

2.9.3. *[Greatest, Least Expressions]* Extend the Arithlang programming language to support two new expressions *greatest-of* (>? a b) and

*least-of* (<? a b). Following interaction log illustrates the intended semantics of this expression.

```
$(>? 3)
3
$(<? 3)
3
$ (>? 3 4 2)
4
$ (<? 3 4 2)
2
$ (>? 3 3)
3
$ (<? 3 3)
3
```

2.9.4. *[Divide-by-Zero]* Current semantics of the Arithlang programming language does not account for divide-by-zero errors.

1. Write a program that uses all arithmetic expressions defined in Arithlang and gives a divide-by-zero error.
2. Extend the Arithlang programming language so that the value of dividing any number by zero is a special kind of value of type `DynamicError`. This kind of value should give users some information about which expression caused the error.

2.9.5. *Memory Cell* A typical calculator provides four operations to remember value: add to memory (M+), subtract from memory (M-), recall value stored in memory (MRec), and clear value stored in memory (Mclr). Extend the Arithlang programming language to support four expressions that model these operations.

1. Memory recall expression (MrecExp) with syntax ( Mrec ) should recall the current value stored in the interpreter's memory.
2. Memory clear expression (MrecExp) with syntax ( Mclr ) should clear the current value stored in the interpreter's memory.
3. Memory add expression (MaddExp) with syntax ( M+ exp0 ... expn ) should add the value of `exp0` to `expn` to the current value stored in the interpreter's memory.

4. Memory sub expression (MsubExp) with syntax ( M− exp0 ... expn ) should subtract the value of exp0 to expn from the current value stored in the interpreter's memory.

## 2.10 Further Reading

### Parsing, Syntax Analysis

For more information about grammar, parsing, syntax analysis, and other aspects of compiler construction see: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*", Addison Wesley; 2nd edition (September 10, 2006), ISBN: 978-0321486813.

For more information about the ANTLR syntax and semantics see: Terence Parr, "*The Definitive ANTLR 4 Reference*", The Pragmatic Bookshelf, Release P2.0 (Jan 15, 2015) ISBN: 978-1-93435-699-9.

### Visitor and other Design Patterns

For more information about the visitor and other object-oriented design patterns see: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "*Design patterns: elements of reusable object-oriented software*", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN: 0-201-63361-2.

### Examples of Interpreters

The implementation of the Java programming language employs both compilers and interpreters. The frontend `javac` is a program that takes a program written in the source language (here Java) and produces another program in the bytecode language. This resulting program is stored in one or more `.class` files and can be run later to perform the computation that the source program was intended to do. The backend `java` is a program that takes a program written in the bytecode language and immediately performs the computation that the program is intended for[2].

[2]We will learn later that even that step is usually divided into two parts: a compiler and an interpreter, but for now let us not worry about those details.