What is a (n executable) model of syntax?

1 Concrete Syntax is Mostly Irrelevant

Different programming languages use different kinds of syntax. Many present syntax as sequences of any characters, with words delineated via some white space, and sentences separated via semicolons or special forms of white space. Language creators write down rules that explain the organization of sequences of characters into words, the grouping of words into sentences, and the recognition of sentences as complete programs.

Borrowing terminology from linguistics, these rules are called *grammars*. The word *Parsing* denotes the process of recognizing sequences of characters as words, groups of words as sentences, and certain sentences as complete program. A *parser* implements the parsing process.

Lisp stands out. The members of the Lisp family of languages use parentheses, braces, and brackets (and arbitrary white space) to separate words and to organize words into groups. Indeed, due to nesting, it is the programmer who takes on some of the task of organizing code to help the parser. People refer to this kind of notation as a *concrete syntax tree*, because the nesting of parenthesize sentences suggests what computer scientists call trees. In Lisp languages, notations for instructions as well as data employ parenthetical notation since the late 1950s; the latter is called an *S-expression*.

The designers of contemporary data description notations, such as XML and JSON, have adapted Lisp's S-expression notation, while making a few concessions to people who like conventional syntax. They employ brackets, braces, and named parentheses to organize data. Developers tend to speak of *semi-structured data*.

XML's pairs of parentheses have the shape <letters> ... </letters>.

Due to the large variety of syntax, studying grammars and parsing techniques has essentially become its own research area. On the positive side, grammars for describing grammars date back to the 1960s, and their expressive power and limitations is well understood. Furthermore, by the 1970s, programming language researchers could offer programs that turn grammars into parsers, so-called *parser generators*. On the negative side, using these tools isn't straightforward. Worse, while a generated parser typically is good at determining whether some sequence of characters satisfies the rules of a grammar, it typically fails to provide an accurate diagnosis when the characters fail the rules.

This is also a reason why courses for beginners should avoid text and parsing.

Hence, if a course on programming language pragmatics were to pay close attention to grammars and parsing, students might never get to see any topics related to semantics or pragmatics. To avoid this problem, we compromise. Concretely, this book uses semi-structured data as concrete syntax for its sample programs; to make this precise, it uses S-expressions. As the history of Lisp and data notations validates, programmers can easily use such a notation to write down programs. At the same time, it enables us to explain the concepts of grammar and parsing in a concise manner and to include a discussion of the pragmatics of parsing.

2 Grammars for Describing Grammars

As the preceding chapter mentions, BNF (or one of its modern variants) is the most commonly used tool to describe the concrete syntax of a programming language. Instead of describing BNF formally, we present just enough examples so that readers can easily comprehend the syntax descriptions in the remaining chapters.

The primary element of a BNF grammar is a *production*, which have this shape:

```
Program ::= a
```

The ::= symbol separates the name that is being defined from its definition. Our convention is to use capitalized words for defined entities and lower case words plus other sequences of keyboard characters for pieces of code. People pronounce such a definition as "Program is a".

A typical grammar describes elements of a language's syntax as not just one possible shape but several possible shapes:

```
Program ::= a | b | c
```

The BNF symbol | is pronounced as "or" and, when a production has this shape, the ::= is pronounced as "one of."

For the syntax description of most languages, it is necessary to use several productions. Additionally, grammatical rules often need to specify that developers may repeat some element several times:

```
Program    ::= Statement* Expression
Statement  ::= print(Expression);
Expression ::= 43
```

This collection of three production rules informs a programmer of three definitions:

1. A Program is a possibly empty sequence of Statements followed by one Expression.

2. A Statement consists of a four pieces: the word print, the character `(`, an Expression, and the characters `)` and `;`.

3. An Expression is just the text 43.

Given a bunch of productions, a programmer can follow their implied instructions to write code. For example, the following is a Program according to the above rules:

```
43
```

Our definitions leave it implicit that white space is ignored.

It consists of no Statement and the required Expression. Here is another one:

```
print(43); print(43); 43
```

This program consists of two Statements, followed by the one required Expression.

Finally, a language creator may wish to require that there is at least one particular piece in a sequence, say, at least one Statement in a Program. The BNF convention is to use a plus superscript as follows:

```
Program    ::= Statement+ Expression
```

And that is all there is to describing grammars with BNF as far as this book is concerned.

3 The Parsing Process and the Parser Function

While the first role of a BNF is to inform the future developer about the shape of well-formed code, its second role is to tell a language implementer how to parse any given text according to some production. For example, given the text

```
44
```

it is impossible to parse it according to the production for Expression. Similarly,

```
print(44);
```

does not belong to the syntax specified by Statement, even though a lot of the characters of this text fit the pattern of the production. By contrast,

```
print(43);
```

is a legal Statement.

In general, parsing some text essentially answers a yes-no question relative to some production:

- yes, the given text belongs to all the collection of all texts that the production describes;
- no, the given text does not belong to the collection of texts that the production describes.

The task of a language implementer is to create a program that, given some inputs, answers this question.

At first glance, this suggests that a parser for some production has the following signature:

```
parseExpression : PlainText  -> Boolean
  // does the given text belong the collection of texts
  // described by the Expression production
```

In words, the parse for the Expression production from the preceding section consumes some text and produces true if the given text belongs to the syntax of the production; otherwise, it returns false to indicate that the text somehow does not fit the description.

For this particular production, it is trivial to define this function:

```
parseExpression(text) =
  true if the text consists of ``4'' followed by ``3''
  false otherwise
```

But, a developer would be rather unhappy if, after entering 44 into the IDE, the parser would display "false" in response. Creating the kind of parser that developers appreciate takes a lot more effort, and this brings us to the first question of language pragmatics.

4 The Pragmatics Question

When developers enter text into an IDE's editor, they want informative feedback. They don't want to know *whether* a text fails to belong to the expected production; they want to know *which part* causes the failure. In other words, they want a red squiggle under 44 when they enter

```
print(43); print(44); print(43); 43
```

into the IDE, because they consider this a program according to the BNF production in section 2.

This practical concern suggests that, at a minimum, parseExpression should return a (data representation of a) reason why some text fails to live up to the expectations of a particular productionn. Using Java's Optional type, a language implementer might propose the following alternative to the original signature of parseExpression

```
parseExpression : PlainText  -> Optional<A>
  //  where A is either
  //  - some data that represents a parsing failure, or
  //  - none, if parsing succeeds
```

Implementing this interface would satisfy the developer who creates the erroneous text from above. After this text is entered into the IDE, the environment would consult the parser function, which would return a some data. In turn, the IDE would interpret the data by marking up the unexpected and faulty piece of text that the programmer expects to be a program.

A language implementer, however, should still have doubts about the usefulness of such a parser. If the parsing succeeds, the result is uninformative.Specifically, it does not share with the caller of a parser for, say, the Program production, a number of interesting insights from the parsing process:

- which parts of the text belong to the Expression;
- which parts of the text belong to the Statement; and
- which parts of the text belong to the Program.

After all, the complete implementation of the language must process this text again to, say, determine its meaning. Clearly, knowing the answers to the above questions is useful in this bigger working context, too.

In short, a language implementer wants an informative result from a parser in either case:

```
parseExpression : PlainText  -> B or C
  // where B is informative data in the error case
  // and C is informative data in the success case
```

The question is what kind of data to pick for `B` and `C`, though before we consider this questions, let's reflect on two concerns that come up as part of all pragmatic questions.

4.1 Pragmatics Pairs Workers With Working Situations

When we consider the pragmatics of a language feature—a syntactic element or a tool that processes language elements—we study a pairing: a human being in a specific work situation. The case of a parser function presents two such pairings:

- the user of the programming language entering text into the IDE and wishing to get fast feedback as to whether this text is well-formed according to the intended grammar production;
- the implementer of the programming language perceiving the parser as just one piece of the complete implementation or, in the case of this book, an executable model for syntax and semantics.

Every consideration of pragmatic questions should spell out which pairing is under consideration, because the implication of making a design decision—picking one of several alternatives, such as the three signatures for parser functions—differs from pairing to pairing.

4.2 Costs and benefits

The costs and benefits of a design decision represent its most important implication. Here is a table for the specific case of a parser function:

Result Type	User	Implementer
Boolean	uninformative	straightforward
Optional<A>	informative	middling complexity
B or C	informative	complex; *useful for* the overall implementation

It concisely summarizes how each person in a specific working situation benefits from a particular design alternative and indicates how much work it imposes on each person. For example, a user working with the first kind of parser must manually compare grammar productions and plain text to figure out why parsing fails. Similarly, the implementer has some benefit from implementing the most complex solution.

Summarizing the benefits and costs of the design alternatives in this manner is a helpful tool for making decisions concerning language pragmatics. While the third alternative offers advantages to both people in the case of parsing, such clear-cut cases are rare when it comes to programming language pragmatics. It is therefore critical that decision makers keep in mind the pairings that determine pragmatic concerns: the human being and the many working situations that those find themselves in.

5 Parsing: In General

Let us now turn to the challenge of designing a data representation that is useful to both successful parsing processes and unsuccessful ones.

We start with an example, namely, the basic assignment statement. It exists in almost all conventional programming languages. Over the past few decades, programming language designers have come up with a fair number of textual variants for this seemingly straightforward statement:

Algol, Pascal	Fortran, C	Scheme, Racket	R, OCaml
x := 1	x = 1	(set! x 1)	x <- 1

This very variety of textual notations for assignment statements is just one of many indicators that concrete syntax is a matter of taste, not pragmatics.

Like in clothing fashion, every decade has had its distinct preferences.

The key to identifying a data representation is to recognize that all basic assignment statements share a common structure. Here is an informal, but structured description of this common structure:

```
an assignment statement consist of two parts:
- a left-hand side,
  - which is a plain-text variable name
- a right-hand side,
  - which is a number in the presented examples
    but, in general, is an expression.
```

A programmer might draw this informal description as a graphical sketch on a white board; see figure 1.

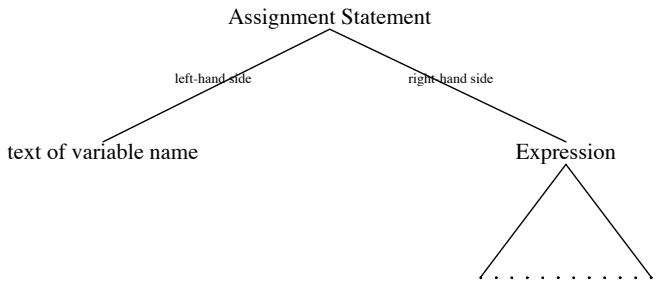

Figure 1: A tree presentation of the data representation for assignments

What this figure suggests is that an assignment statement can be understood as a node in a tree. The node has two branches: one for the left-hand side and one for the right-hand side. While the left branch points to a leaf, namely, the name of the variable, the right one ends in a sub-tree, because the right-hand side of an assignment statement is an expression and because the representation of an expression is likely to be a tree all by itself.

To make this last idea concrete, consider the concrete syntax for an addition expression. In Lisp, it is (+ x 1), and in most other languages it uses the notation familiar from grade school, x + 1. But just like assignment statements, all of these addition expressions share a common structure:

```
an addition expression consist of two parts:
- an operator on the left,
  - which is another expression
- an operator on the right,
  - which is another expression
```

Stop! Draw the tree that corresponds to this informal description. Then draw the tree for the following two assignment statements:

- (set! x (+ x 1)), which is what a Scheme programmer writes to increase x by one;
- x = x + 1, which is what a Fortran, C, or Java programmer uses for the same purpose.

Using this analysis, we can express this idea as a collection of interfaces and classes in Java. Take a look at the two columns of code in Figure 2. The left column displays the data representation of an assignment statement.

While the name of the class signals that an instance represents an assignment statement, its two fields correspond to the two clauses of the informal description or the two branches of the tree in figure 1. The type specification for lhs implicitly informs a reader that variable names are represented as Strings.

```
class AssignmentStatement {
  String lhs;
  Expression rhs;
}
```

```
interface IExpression { }

class VariableReference
 implements IExpression {
  String name;
}

class Addition
 implements IExpression {
  IExpression left;
  IExpression right;
}
```

Figure 2: A data representation for assignment statements and expressions

The right column is the data representation for simple expressions, variable references and additions to be precise. Since there are many different kinds of expressions that the rest of the program treats in a uniform manner, the data representation uses an interface to tie the various variants together. In our case there are two variants: variable references such as the x in x + 1, and addition expressions.

Stop! Reformulate the data representation from figure 2 in your favorite programming language. Then use your data representation to represent (set! x (+ x 1)) and x = x +1.

Choosing such a tree-oriented data representation clearly answers half the question at the end of section 4. If a parser returns a tree-shaped value when parsing succeeds, the nodes in the tree identify the BNF production that the parse recognized. The pieces of each node represent the parts of the plain text that correspond to one of the alternatives on the right-hand side of the BNF production. Because these tree-shaped data structures are generalizations of many concrete-syntax shapes for the same kind of syntactic element, language researchers call them *abstract syntax trees* or just *AST*s for short.

The remaining question is what a parsing function should return when parsing fails. Since the pragmatics question about parsing identified the

IDE and the user of the IDE as the recipient of the relevant information, the data representation must enable the IDE to mark up the pieces of text that do not match the given BNF.

Let's make this concrete with the example from section 3:

```
print(44);
```

This text almost matches the Statement production. Specifically, it matches the prefix characters—`print(`—and the suffix characrers—`);`. It is `44` that causes the parser implementation to fail when it tries to match it with the Expression production.

A bit of reflection suggests that the IDE needs to know

- which part of the text the parser recognizes, and
- which part causes the parser to fail.

Figure 3 displays one reasonably straightforward solution. On the left, it shows the original grammar from section 2. On the right side, the figure lists a mostly direct translation of this grammar into an AST representation. Each production is turned into a class, with fields to represent the pieces. For the statement class, this means a String array field for the potentially empty sequence of Statements and a IExp field for the final Expression. The Statement class contains one field: to represent the Expression that that parser expects to find.

Figure 3 also suggests what happens if the parser does not find an Expression according to the BNF. The interface IExp is implemented twice: once by the Expression class and once by an Error class. So, when the parser for the Statement production is confronted with print(44);, it defers to the parse for the Expression production, which returns an instance of Error instead of the Expression class. In other words, the parser's output remains a tree, but this tree representation contains nodes that indicate that an error happened. When given an abstract syntax tree with an error node or even several such error nodes, it can use the structure of the tree together with the data in the Error node to mark up text that can't be parsed.

Stop! Reformulate the data representation from figure 3 in your favorite programming language. Then use your data representation to represent print(44);.

5.1 A Parser Returns an Abstract Syntax Tree

Let's draw the general lesson from these examples. First, parsing plain text produces an abstract syntax tree. Second, in order to report failures in an

```
Program    ::= Statement* Expression
Statement  ::= print(Expression);
Expression ::= 43
```

```
class Program {
  Statement s[];
  IExp r;
}

class Statement {
  IExp e;
}

interface IExp { }

class Expression
 implements IExp {
  // there is one: 43
}

class Error
 implements IExp {
  String badText;
  // data for the IDE
}
```

Figure 3: A data representation for the grammar from section 2

informative manner, the AST representation is enriched with error nodes.

The signature for every parser function is essentially

```
parse : PlainText  -> AST
```

To distinguish the success case from the failure case, we introduce the convention that AST is a tree that *might* contain an error node and that AST- denotes a tree without any error nodes.

5.2 Parse, Don't Validate

While parsing is a programming-language concept, software developers have recognized its general usefulness. That is, a software system that consumes plain text requiring validation—say from a file, a network connection, or otherwise—should parse the text into a tree format. In the successful case, processing this tree is advantageous and safe compared to processing plain text. And, if the parsing fails, the software system can log the failure with an error message that explains the problem with the text.

6 Parsing: An Example

It is time to put all the pieces together and look at a complete example. As the introduction to this chapter says, this book simplifies the parsing task to avoid getting bogged down in questions of concrete syntax. To start with, the parser presented here analyzes semi-structured data for grammars that identify a subset of semi-structured data. While XML and JSON are modern versions of this idea, the idea itself is decades older and the original version—S-expressions—deserves the honor.

For our purposes, an *S-expression* is a fully-parenthesized term whose leaves are numbers and symbols. The word "leaves" should remind you that a piece of semi-structured data forms a tree. Here is an informal yet rigorous definition:

```
S-expression is one of:
 -- Number
 -- Symbol
 -- (S-expression ... S-expression)

A symbol is a non-empty sequence of keyboard characters
that is not also a number.
```

Stop! Re-formulate this definition with a BNF.

Since most programming languages come with a core of expressions and statements, our first model language—dubbed *Sample*—includes just those elements. To keep things simple, the following grammar also severely restricts the shapes of expressions and statements:

```
Program    ::= (Statement* Expression)
Statement  ::= (Variable = Expression)
Expression ::= 1.0
             | 2.0
             | 3.0
             | Variable
             | (Expression + Expression)

The set of Variables consists of all non-empty sequences of
alphanumeric characters, starting with an alphabetical
letter of maximal length 100.
```

Like the grammar in the preceding sections, the set of well-formed Programs consist of a potentially empty sequence of statements followed by one expression. There is only one kind of statement, namely, assignment statements that resemble those found in C, Java, and similar languages. Finally, the set of expressions comes with an infix-style shape for addition, variable

occurrences, plus the numbers 1.0, 2.0, 3.0. Although it would be possible to define a BNF grammar that specifies variable names, we choose to use English instead for simplicity and to illustrate common practice.

The phrase "proper program design" refers to an understanding of programming as presented in How to Design Programs.

Even simple BNF grammars call for the creations of examples as does proper program design:

```
;; concrete
((x = 1.0)
 (y = 2.0)
 (z = (x + (x + y)))
 (z + 3.0))
```

Clearly, concrete is an S-expression. The question is whether it satisfies the Program grammar. As before, we can verify this fact manually:

- concrete consists of four S-expressions between parentheses. Hence, we must check that the first three belong to the set of Statements and the last one to the set of Expressions.

- (x = 1.0) is a Statement, because x belongs to the set of Variables and 1 is obviously an expression.

- Verifying that (y = 2.0) and (z = (x + (x + y))) belong to the production for Statements proceeds in a similar manner.

  Stop! Don't just accept this sentence. Just do it!

- Finally, (z + 3.0) is an Expression:

  it consists of (followed by a Variable followed by +, followed by 3.0, and wrapped up by).

Of course, the goal is to design a parser function for the grammar.

At this point, we need to choose a programming language in which to articulate an AST data representation and a parser for the BNF of *Sample*. Our favorite programming language for this purpose is Racket, and you may understand why after reading the next couple of pages.

Stop! Settle on your favorite programming language, and choose your favorite semi-structured form of data, e.g., XML or JSON instead of S-expressions. Work through the same exercise using your choices.

Figure 4 consists of two columns, which jointly specify our entire AST data representation. The left-hand column contains four groups of structure-type definitions:

- The first three groups correspond to the three productions of the BNF grammar of *Sample*.

```
(struct prog
  [statements
   end])


(struct ass  [lhs rhs])

(struct expr ())
(struct num expr (n))
(struct ref expr (name))
(struct add expr (left right))

(struct err (msg))
```

```
#; {type Prog =
     (U Err
        (prog
          List<Stmt>
          Expr))}
#; {type Stmt =
     (U Err
        (ass Symbol Expr))}
#; {type Expr =
     (U Err
        (num N)
        (ref Symbol)
        (add Expr Expr))}
#; {type Err  =
     (err String)}
```

Figure 4: A Racket AST data representation for *Sample*

- Since the Expression production describes three kinds of alternatives—numerals, variables, additions—the third group consists of a structure type that plays the role of an interface—expr—and three implementing variants—num, var, and add.

- The fourth and extra group contains a single structure-type: err. It exists to inject error nodes into the AST.

Since Racket does not impose type constraints on code, instances of err can be injected into an AST at any place. Also, note how stmt and expr come with source-location fields, which the IDE needs to highlight problematic code elements.

The notation leans on Typed Racket.

The right-hand column consists of comments that explain the relationship among the structure types as if Racket had a type system. It tells a reader of the code that, for example, an instance of prog always contains an instance of Sta* in the first field and an instance of Expr in the second. Furthermore, the quasi-type Prog is a union that contains Err and all legal instances of prog, meaning a parser function for the Program production may return (err "...").

Stop! Work out an S-expression that is *not* a member of the set that the Program production describes.

Stop again! Explain the remaining type-like comments on the right-hand side of figure 4.

Here is the AST for the concrete example of a Program:

```
define ast
  (prog
   (list
    (ass 'x (num 1.0))
    (ass 'y (num 2.0))
    (ass 'z (add (ref 'x) (add (ref 'x) (ref 'y)))))
   (add (ref 'z) (num 3.0)))
```

It is an instance of prog, which contains a list of ass instances in the first field and an instance of expr—specifically an instance of add, which is a sub-type of expr—as the last one.

Racket greatly facilitates writing a parser that determines whether an S-expression satisfies the grammar rules of the BNF grammar of *Sample*. Figure 5 displays the entire program. The main function composes two computations: read, which reads an S-expression from the standard input or console, and program->ast, which attempts to parse this S-expression as a member of the Program production.

Following proper program design ideas, the remaining functions in figure 5 correspond to the three productions in the BNF of *Sample* in a one-to-one fashion. The three functions are named in a fashion that clarifies their role in the parsing process. Each function consists of $n + 1$ conditional clauses, where n is the number of alternatives on the right-hand side of the production. The extra clause corresponds to a mismatch of the given S-expression and the parsed production.

Let's take a close look at the program->ast function:

- The function uses Racket's algebraic match form to check whether the given S-expression is a list. Concretely, (list s ... e) matches txt if the latter is a list and contains at least one element, e. All other elements are collected in a potentially empty list called s.

- If txt matches, the function instantiates prog; if txt fails to live up to *Sample*'s BNF, it does do in nodes below prog.

- Otherwise the conditional's catch-all clause returns an instance of err to indicate that the S-expression cannot possibly belong to the Program production.

  This last clause kicks in even if the set of S-expressions were extended with additional forms of data.

Stop! Try to explain the remaining two functions before reading on.

```
#; { -> Prog}
;; parses the S-expression given on STDIN
(define (main)
  (program->ast (read)))

#; {S-expression -> Prog}
(define (program->ast txt)
  (match txt
    [(list s ... e)
     (prog (map statement->ast s) (expression->ast e))]
    [_ (err (~a "program expected, given " txt))]))

#; {S-expression -> Stmt}
(define (statement->ast txt)
  (match txt
    [(list lhs '= rhs)
     (ass lhs (expression->ast rhs))]
    [_ (err (~a "statement expected, given " txt))]))

#; {S-expression -> Expr}
(define (expression->ast txt)
  (match txt
    [1.0 (num txt)]
    [2.0 (num txt)]
    [3.0 (num txt)]
    [(? symbol?) txt] ;; needs additional checking
    [(list left '+ right)
     (add (expression->ast left) (expression->ast right))]
    [_ (err (~a "expression expected, given " txt))]))
```

Figure 5: A Racket parser for the BNF grammar of *Sample*

Of the remaining two functions, statement->ast is straightforward; expression->ast deserves an explanation. The match of expression->ast consists of five plus one clauses:

- The first five clauses correspond one-to-one to the right-hand side of the Expression production.
- The first three clauses are literal matches for 1.0, 2.0, and 3.0.
- The fourth clause (mostly) takes care of the parsing of Variables.

  Stop! As is, this clause would record the symbol +-*/ as member of Variable, but it obviously isn't according to *sample*'s BNF. How would you fix the program to enforce the constraints on Variable occurrences

properly? Make sure your own implementation of this *Sample* parser does a better job than our program.

- The last clause is a catch-all clause, which is—as always—used to report errors, that is, cases when the given S-expression should be a member of Expression but isn't.

And this is all there is to creating a parser—as far as this book is concerned.

6.1 Why Parsing Semi-structured Data Matters

The example makes all look too easy and, because of this, perhaps a bit irrelevant. But, appearances are deceiving. Time and again, people have re-discovered forms of semi-structured data for a number of different purposes: storing intermediate results, conveying data from one networked computer to another, and writing programs in little language.

Such little languages often serve as an extension mechanism for large software systems. Among other things, they play a role for configuring software system during start-up or for loading additional functionality while the systems run. To this end, the systems contain a component that can execute programs in these little languages, either directly or via abstract machines.

In short, don't write off this exercise as a trivial little program. It might just come in handy one day.

7 Project Syntax: *Bare Bones*

Figure 6 presents the BNF grammar of the first project language. It extends *Sample* so that the syntax resembles the core of most contemporary programming languages, minus the parentheses. While a Program is still a sequence of Statements followed by an Expression, the set of Statements contains two new variants:

- (if0 Expression Block Block), which looks like the simple conditional, combining a conditional expression with two blocks of Statements; and
- (while0 Expression Block), which is representative of the looping constructs found in ordinary languages.

The symbols =, if0, and while0 are special. They serve as markers that differentiate sentences from each other. In acknowledgment of this role, such symbols are called *keywords*.

```
Program    ::= (Statement* Expression)

Statement  ::= (Variable = Expression)
             | (if0 Expression Block Block)
             | (while0 Expression Block)

Block      ::= Statement
             | (block Statement+)

Expression ::= GoodNumber
             | Variable
             | (Variable + Variable)
             | (Variable / Variable)
             | (Variable == Variable)

The set of Variables consists of all symbols, minus keywords.

The set of GoodNumbers comprises all inexact numbers
(doubles) between -1000.0 and +1000.0, inclusive.
```

Figure 6: The concrete syntax of the *Bare Bones* language

The grammar in figure 6 introduces one innovation: Blocks. Both the if0 alternative and the while0 alternative of the Statement production refer to Block, thus allowing programmers to use either one Statement or a non-empty sequence of Statements.

Finally, the set of Expressions comes with a few additions compared to those of *Sample*: many more numerical literals, a division expression, and a comparison expression. Note, however, that it also restricts the shape of Expressions in comparison to *Sample*. Thus, the *Bare Bones* language no longer contains the concrete example from section 6.

Stop! Determine where a parser for *Bare Bones* would flag the concrete syntax of concrete to inform the programmer of a grammar violation.

A *Bare Bones* programmer can easily work around this restriction with assignments to temporary variables. Here is the result:

```
;; concrete-bare-bones
((x = 1.0)
 (y = 2.0)
 (temporary = (x + y))
 (z = (x + temporary))
 (temporary = 3.0)
 (z + temporary))
```

Here the use of a single additional variable, temporary, eliminates the two

syntactic problems. The nested Expression and the nested literal constant are "lifted" into the surrounding sequence of Statements and given a name: temporary.

Exercise 1. Your task is to design an AST data representation for *Bare Bones* and to implement a parser for this language in your favorite programming language. The parser maps an S-expression to an instance of AST, an abstract syntax tree that may contain error nodes.

7.1 Model

The *Bare Bones* language represents our first, simple model. It looks and feels a bit like the kind of languages you should have encountered so far, except for the many parentheses, which simplify the parsing process. It differs from these languages in two visible ways: (1) it lacks nested expressions, and (2) numbers are its only form of data.

Since the goal of this book is to examine questions of pragmatics, we must judge these differences in those terms. From the perspective of pragmatics, the first difference is superficial. We have already seen a trick that overcomes the lack of nested expressions on the basis of simple local transformations to a program. As a matter of fact, this restriction is merely imposed to simplify your project.

The second difference is highly significant; in this day and age, we know that a language should support different forms of data and indeed an extensible collection of data types. Chapter VI addresses this obviously pragmatic issue in depth.