Why do most languages demand that programmers declare variables?

1 The Pragmatics Question

The *Bare Bones* language of the preceding chapter resembles some existing ones, and it comes with one well-known, serious flaw. For example, contemporary Python still allows programmers to just write down assignment statements to variables and fields without requiring declarations. Here is an example, using *Bare Bones* syntax:

```
((title = 1.0)
 (offset = 0.0)
 ... 1,000 lines of loops and conditions ...
 (titel = title + offset)
 title)
```

Stop! Take a look at the boxed code, which highlights the problem. Has something like this happened to you when you wrote Python code, perhaps with a field assignment?

Unlike the problem of references to variables without value in an expression, which the semantics can deal with, the problem of assignments to misspelled variables does not cause the *BareBones* parser to signal a syntactic error or the *BareBones* semantics to map its meaning to error. Assuming this program does not suffer from bad variable references, it is syntactically well-formed, and the runMachine semantics returns a number for this program.

If it is not a syntax problem and if it is not a semantics problem, the program poses a problem of pragmatics. Since we defined pragmatics as a concern of language combined with work situations, we need to identify these two for this specific problem. So, imagine a quality engineer whose task is to write black-box tests. Unless such tests force the value of offset to

differ from 0.0, these test may never reveal that the program suffers from a misspelled variable name in an assignment statement. Once this software gets deployed, however, a user may run this program in ways that *does* expose the problem in an observable manner.

To sum it all up, typos can pose serious problems to programmers and users in *BareBones* programs. We need to consider solutions and the implications of such solutions.

1.1 Design Choices: Valid Syntax vs Style Checks

Almost all contemporary programming languages make the declaration of variables mandatory. Every variable reference and every assignment to a variable must connect to a variable declaration. Language implementations check this property even before a programmer runs a program; unless the property holds, a program in such contemporary languages can't run. Similarly, modern IDEs mark up uses of undeclared variables so that programmers can fix such problems while they create code.

Generalizing a bit, two fundamental changes to the syntactic part of the programming language answer this first pragmatic question:

1. The language's syntax must include a facility for declaring variables.
2. The language implementation must come with a tool that checks that all references and assignments to variables relate to a corresponding declaration.

The second property points out that checking syntax is more than parsing to check whether a program is well-formed according to some BNF. In addition, a language implementation must check that the well-formed syntax is valid, meaning it satisfies additional properties not expressible via BNF.

Fundamental properties of computation render such tools at best reasonably sound.

Some language communities consider variable declarations a nuisance but admit that the pragmatic question exists. Their leaders have therefore proposed to equip a language's tool chain with a "style check"er (aka, linter) for checking code properties. For the case of references to variables without values in the store, such a software tool can read the program text, analyze it, and make a (reasonably) sound judgment as to whether running the program may signal an error. For the other case, however, when a programmer misspells the variable on the left-hand side of an assignment statement, such software tools can make at best good guesses—and programmers are annoyed when bad guesses pile up. Unsurprisingly, programmers tend to turn off these optional tools and fail to discover these problems too late.

A comparison of these two design choices makes it quite obvious why the vast majority of language designers have resorted to the first solution and have mostly rejected the second one. As a matter of fact, they recognized the pragmatic question in the late 1950s and came up with the solution in 1960, understanding its pros and cons over 80 years ago.

1.2 Costs and Benefits

Every decision concerning pragmatics deserves a cost-benefit analysis, even a decision as clear-cut as the one in favor of variable declarations. In the case of programming language pragmatics, such cost-benefit analysis takes into account the working situations of the programmer that raised the pragmatic question *and* the work that the decision imposes on the language implementer. While shifting work from a lot of programmers to a few language implementers is generally a good idea, it may become unreasonable when the decision is a close one.

As far as programmers are concerned, variable declarations come with some obvious positives and one negative one for the programmer. On the positive side, variable declarations protect programmers from insidious mistakes and unwanted "undefined variable errors" discovered when the program is run. Indeed, the language implementation can collaborate with the IDE and warn programmers about problems as they edit code. That is, variable declarations support the act of entering code into an editor. On the negative side, the requirement to declare variables before use imposes a small amount of extra keyboarding on the programmer. On balance, the positives clearly favor the addition of variable declarations.

As far as the language creator is concerned, variable declarations impose two costs. The first one is intellectual. In addition to a BNF specification, which determines when some text is a well-formed program, the creator is now responsible for making decisions on what it means that a variable is defined before it is used. The decision is expressed as a set of constraints between distant pieces of code. A well-formed program is really considered a program suitable for execution only when it satisfies these constraints too. Language researchers tend to say a program is *well-formed* and *valid* when it is formed according to the BNF and when it also satisfies the declared-use constraints, respectively.

Finally, concerning the language implementer, variable declarations and their accompanying constraints impose extra work. While a parser can check whether a program is well-formed, it isn't suitable to check whether a well-formed program is valid. To implement this validity check, the lan-

guage implementer must process the given program again, *after* it is known to be well-formed. And now it becomes clear, why the parser's creation of an abstract syntax tree (AST) is so important:

> If the validity checker is a function from ASTs to something else, its implementation does not have to account for syntactic mistakes. That is, working with ASTs simplifies subsequent checks of desirable program properties.

In other words, while the introduction of ASTs may have appeared to complicate things at first, it should now be clear that it simplifies language implementations.

2 Valid Syntax: In General

The preceding cost-benefit analysis concludes with the idea that validity checking is a function that consumes an abstract syntax tree. To work out the precise nature of its signature, let's start by recalling the generic signature of a parser:

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

Its domain is, in general, plain text, typically read from an input device; its range is AST, which is the set of abstract syntax trees enriched with error nodes. The IDE uses the latter to inform programmers of syntax mistakes, that is, violations of the language's BNF.

If the parse function discovers a violation, it is unnecessary to check whether the program comes with declarations for all variables. Conversely, the validity checker consumes only the AST- subset of AST. The question is what this function should return. Just like parse, the validity checker may succeed and confirm that all variables mentioned in the program come with a declaration; or it may fail and discover a variable name without a declaration. In the failure case, we would once again like the language implementation to inform the IDE so that, in turn, it can highlight the undeclared variables. In short, we want the following signature:

```
// check that variables in a program come with a declaration
validityCheck : AST-  -> AST
```

Unfortunately, working out the top-level signature is the easy part; the difficult aspect of validity checking is due to the physical separation of variable declarations from variable occurrences, be that on the left-hand side of

an assignment statement or inside of an expression. Following proper program design principles, validityCheck dispatches to auxiliary functions that deal with declarations, statements, and so on. And this implies that these auxiliary functions need to communicate which variables have been declared. We can express this relationship with two sketched signatures:

```
// check that variables in a piece a program are declared
validityCheckOfPieces :
   ASTforPiece- Set<Symbol>  -> ASTforPiece

// determine the set of declared variables
// ensure that declarations are valid
validityCheckOfDecls :
   ASTforDecl ... -> ASTforDecl+ & Set<Symbol>
```

The first signature says that checking pieces of a program—say, statements—requires knowledge about which variables have been declared. We use sets of variables to represent this knowledge. The second one is for a function that processes variable declarations. It clearly needs to return the set of variables it finds. But, as the next section shows, even variable declarations may be invalid, which is one reason why it also returns a potentially annotated abstract syntax tree.

```
(define (main)
  (define s (program->ast (read)))

  ;; STATIC
  (define parsed (program->ast s))
  (unless (plain-ast? ast-well-formed)
    (error "well-formed program expected"))

  (define valid (closed-program parsed))
  (unless (plain-ast? valid)
    (error "valid program expected"))

  ;; DYNAMIC
  (runMachine valid))
```

Figure 12: Language implementations consist of many passes

Now that we have parsers, validity checkers, and a machine, it is time to adapt the main function from figure 5 to cope with them all. Figure 12 explains how this composition works. The function reads an S-expression from the standard input device, followed by two traversals of the program:

parsing and validity checking. Each traversal is followed by a conditional that sends an error signal to the caller unless the result of the traversal is an abstract syntax tree without error nodes. If both of these traversals succeed, main uses runMachine to determine the result of the well-formed and valid program—either a number or an error message.

The figure also indicates a bit of terminology concerning language implementations, meaning executable models, interpreters, or compilers. Each such implementation consists of two parts: a *static* one and a *dynamic* one. The static part always comes in the shape of a series *passes*, each of which checks some properties and potentially computes information about the given program. Furthermore, pass $n + 1$ always consumes the results of pass n and relies on the properties it has checked. The dynamic part consists of just the semantics, that is, a piece of functionality that consumes well-formed and valid programs to determine its value. While our models use abstract machines, a proper implementation typically realizes this functionality via a combination of hardware and software.

Going forward, this book will present additional static passes and significantly richer abstract machines than the CS or CSK machine. In this chapter still, we also explain how static checks eliminate dynamic checks. Put differently, the construction of an abstract machine—or the code sent to hardware—may assume that the success of the static passes and that the corresponding properties hold. Indeed, this is also true for the communication between the language implementation and the IDE; it, too, can use information gathered from static passes to assist the programmer.

3 Valid Syntax: An Example

Let's illustrate the abstract explanation from the preceding section with an extension of the *Sample* language that comes with variable declarations:

```
Program     ::= (Declaration* Statement* Expression)
Declaration ::= (def Variable Expression)
Statement   ::= (Variable = Expression)
Expression  ::= 1 | 2 | 3 | Variable | (Variable + Variable)

The symbol `def` is not a variable.
```

Remember that keywords merely separate S-expressions that play one kind of code from those that play others.

We keep referring to this language as *Sample*. Its programs are still a single S-expressions, though this one consists of up to three pieces: a potentially empty sequence of variable declarations, followed by a potentially empty sequence of (assignment) statements, wrapped by a single expression. The

addition of a production for Declarations is the only new part. It uses one additional keyword: def, similar to the keywords in *Bare Bones*. And like the grammar for *Bare Bones*, the grammar for *Sample* does not permit programmers to use def as a variable name.

Here are the two sample programs from the preceding chapter modified to satisfy the Program production of the *Sample* BNF:

well-formed, valid

```
((def a 1.0)
 (def b 2.0)
 (def temporary (a + b))
 (def c (a + temporary))
 (temporary = 3.0)
 (c + temporary))
```

well-formed, invalid

```
((def a 1.0)
 (def b 2.0)
 (def temporary (a + b))
 (def c (a + temprary))
 (temporary = 3.0)
 (c + temporary))
```

A comparison with the preceding chapter tells us that the first three assignment statements have been changed to variable declarations, which consist of a symbol and an Expression. The box around temprary, the misspelled variable inside the addition expression, indicates where an IDE should flag the code as invalid.

Stop! Formulate a well-formed and valid program whose sequence of variable declarations is empty.

Stop! Did you notice that it is a *variable declaration* that is invalid?

What the right-hand column illustrates is something that the abstract explanation of the preceding section could only allude to, namely, that a variable declaration may be well-formed but invalid. In the case of *Sample*, a variable declaration consists of a name and an expression. The latter may refer to variables—and those variables should already be declared.

The word "already" points back to the word "sequence" (of variable declarations), which implies order. So, if a programmer writes

```
((def oneVariable 1.0)
 (def anotherVariable 2.0)
 (def andAThirdVariable (oneVariable + anotherVariable))
 ...)
```

then the validity checker inspects one such declaration after another. In this particular example, the first declaration checks out because there are no variables on the right-hand expression; the second one is valid for the same reason. Key is that each declarations contributes one variable to the set of declared ones. Hence, right before the validity checker inspects the third declaration the set of declared variables is {oneVariable, anotherVariable}. The checker's next task is to look at the expression part of the third declaration,

which is an addition expression. Such expressions consists of two variable references separated by +. For each of these variables, the validity checker must clarify whether they are already in the set of declared variables. For this example they are, and therefore the checker blesses this sequence of declarations.

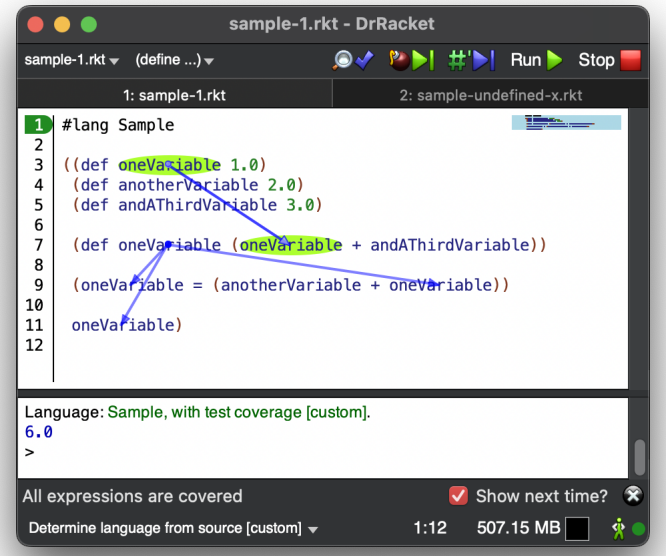

Figure 13: Binding in the *Sample* language

Our explanation immediately raises the question whether it is acceptable that such a sequence contains *two* declarations for the *same* variable. Here is a corresponding modification of the running example:

```
((def oneVariable 1.0)
 (def anotherVariable 2.0)
 (def andAThirdVariable 3.0)

 (def oneVariable (oneVariable + andAThirdVariable))
 ...)
```

And the answer to this question is "it is up to the language creator."

Some creators disallow duplicate variable declarations; others accept them; and in some languages one form of variable declaration is made for duplicate declarations and a different one prohibits them. For illustrative purposes, the *Sample* language accepts duplicate variable declarations, and figure 13 illustrates with a screenshot what this means.

The program in the figure starts with the very sequence of declarations under discussion and adds an assignment statement plus a final expression. The arrows indicate how variable references are resolved. While the first declaration of oneVariable points to the occurrence in the expression of line 7, the second declaration—on line 7—points to three occurrences below: one on the left-hand side of the assignment, one on the right, and one in the final expression. The technical term for this relationship among variable declarations and variable occurrences is *binding*; that is, people say the occurrence inside the expression is bound by the first declaration and the occurrences on lines 9 and 11 are bound by the second declaration.

The screenshot is for an implementation of Sample, with the DrRacket IDE tailored to this implementation via the first line.

```
                                    #; {type Prog =
                                          (U Err
(struct prog                                 (prog
 [declarations                                Listdecl>
  statements                                  Liststmt>
  end])                                       Expr))}
                                    #; {type Decl =
                                          (U Err
(struct decl [var rhs])                      (decl Symbol Expr))}
                                    #; {type Stmt =
                                          (U Err
(struct ass  [lhs rhs])                      (ass Symbol Expr))}
(struct expr [])                    #; {type Expr =
                                          (U Err
(struct num expr (n))                        (num N)
(struct ref expr (name))                     (ref Symbol)
(struct add expr (left right))               (add Symbol Symbol))}
                                    #; {type Err  =
(struct err (msg))                        (err String)}
```

Figure 14: An AST data representation for *Sample* with declarations

3.1 The Implementation

Figure 14 displays the adaptation of our AST data representation to *Sample* enriched with variable declarations. It comes with one new struct type, decl,

and a corresponding type-like comment. The prog struct type has one additional field, declarations, which is going to contain the AST representation of the sequence of declarations. Otherwise, the AST data representation is like the one in figure 4; a quick comparison may help.

```
#; {Prog- -> Prog}
(define (closed-prog prog-)
  (match-define (prog decl-* stmt-* expr-) prog-)
  (define-values [decl* declared] (closed-decl* decl-* '[] (set)))
  (define stmt* (closed-stmt* stmt-* declared))
  (define expr  (closed-expr expr- declared))
  (prog decl* stmt* expr))

#; {List<Decl-> List<Decl> Set<Var.> -> List<Decl>, Set<Var.>}
;; accumulate the declared variables, checked declarations
(define (closed-decl* ast-decl* decl-* declared) ...)

#; {Decl- Set<Var.> -> Decl Set<Var.>}
(define (closed-decl ast-def1 declared) ...)

#; {List<Stmt-> Set<Var.> -> List<Stmt>}
(define (closed-stmt* ast-stmt* declared) ...)

#; {Stmt- Set<Var.> -> Stmt}
(define (closed-stmt ast-stmt declared) ...)

#; {Expr- Set<Var.> -> Expr}
(define (closed-expr ast-expr declared) ...)
```

Figure 15: A validity checker for the *Sample* language

The chosen AST representation drives the design of the validity checking functionality. Since there are five kinds of AST nodes, the validity checking functionality consists of (at least) five functions or methods: one per kind of AST. Furthermore, the function for programs is the entry point and can thus serve as an overview of the complete piece of functionality.

So take a look at the top-most function definition in figure 15. It uses the name closed-prog, because programming-language researchers say a "program is closed" if all variable occurrences refer to a variable declaration. Because this function consumes an error-free AST—ast-prog—it suffices to match it against the only kind of struct instance of Prog-. The result of this match are the three pieces of an AST for a program: the list of declarations, the list of statements, and the final expression. Each of these pieces is processed in turn, yielding ASTs that potentially contain error nodes when

undeclared variables are discovered. The last line of closed-prog assembles these pieces back into a complete prog node.

Of the three processing steps of closed-prog, the one concerning declarations is the only interestng one. A sequence of declarations contributes not one, but two results to the overall check: (1) the declared variables and (2) the results of checking each declaration. The signature indicates this insight with two kinds of data on the right side of the -> marker:

```
#; {List<Decl-> List<Decl> Set<Var.> -> List<Decl>, Set<Var.>}
;; accumulate the declared variables, checked declarations
(define (closed-decl* decl-* decl* declared)
  (match ast-decl*
    ['[] (values decl* declared)]
    [(cons decl1- others)
     (define-values (decl1 declared++)
       (closed-decl decl1- declared))
     (closed-decl* others (snoc decl* decl1) declared++)]))
```

Note the distinction between the first input—List<Decl->, a list of error-free ASTs—and the first result—List<Decl>, a list of ASTs that may contain error nodes. Returning such ASTs is necessary so that the IDE can inform programmers of the kinds of mistakes that the beginning of the section illustrates. The closed-decl* function traverses the list using a match conditional. When the list is empty, it returns the ASTs and the set of declared variables as a tuple of two values. Otherwise, it processes the first declaration and then "loops" over the remaining ones.

The last building block of interest is the function that deals with an individual variable declaration:

```
#; {Decl- Set<Var.> -> Decl, Set<Var.>}
(define (closed-decl decl1 declared)
  (match-define (def x rhs-) decl1)
  (define rhs  (closed-expr rhs- declared))
  (values (def x rhs) (set-add declared x)))
```

It deconstructs the given declaration into the variable and the right-hand side of the declaration. The latter is an expression that may contain one or two variables. The closed-expr function checks that these variables are declared and, if not, constructs an AST with appropriate error nodes. In turn, closed-decl combines whatever AST closed-expr returns with the declared variable into a new AST: (def x rhs). The function's result is a tuple of two values: the potentially revised AST and the set of declared variables enriched with the variable found in the given declaration.

All remaining functions for validity checking in figure 15 are much more straightforward than the two concerning variable declarations. Stop!

Take a look at the figure and try to sketch the last three functions, which are specified via signatures and meaningful names.

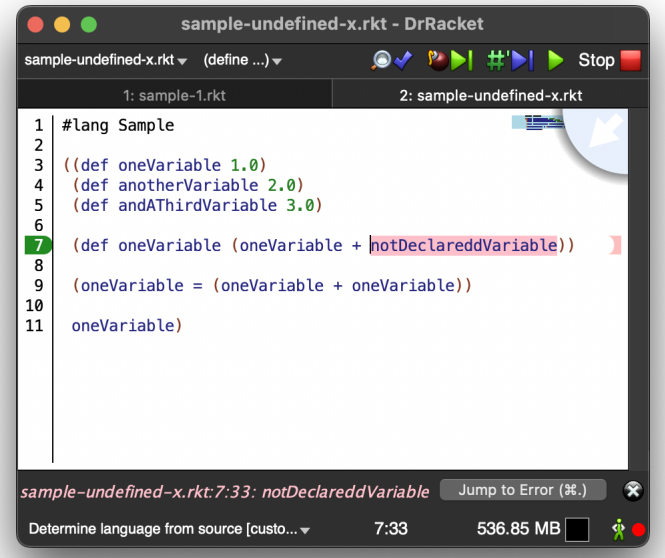

Figure 16: Undefined variable errors in an IDE for the *Sample* language

3.2 The Implementation for an IDE or Compiler

Take a second look at figure 13. Arrows in this figure indicate which variable declaration binds which variable occurrence. Question is how an IDE can know where to anchor these arrows and where to point them to. Now take a first one at figure 16. It shows how an IDE for *Sample* highlights a variable occurrence without a corresponding variable declaration. Again, the question is how an IDE can highlight this part of the program text. Finally figure 17 illustrates yet another case when an IDE clearly needs more than the plain AST data representation of this book: a name refactoring tool for the *Sample* language.

An IDE needs an abstract syntax representation that includes the data representations of the essential pieces of some program text *and* the source

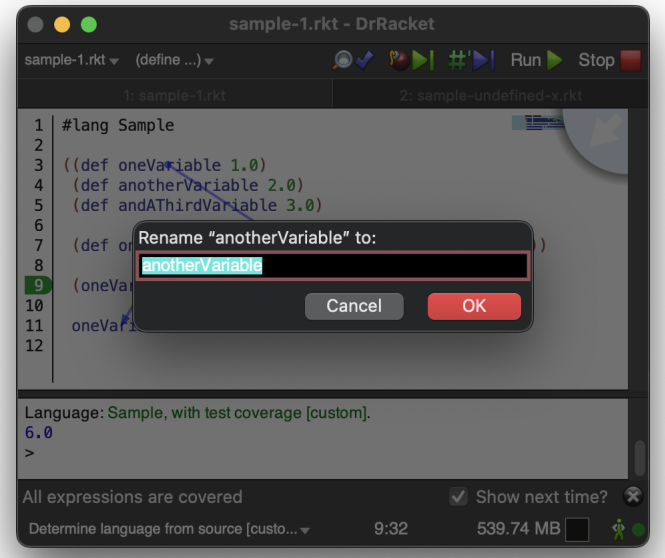

Figure 17: Name refactoring for the *Sample* language

locations of all these pieces. For example, the DrRacket IDE for *Sample* records the line of each piece of plain text in an AST node and the column. In order to perform a decent job with drawing arrows between variables, the IDE also records the span of the names. Even plain compilers that wish to report syntax and validity errors in a productive manner need to include such information in their data representations.

Put differently, we once again expose the difference between a *model* of some aspect of languages and an actual *realization* of the same aspect in the tool chain that software developers actually use. While the executable model suffices to gain an understanding of what, for example, validity checking means, the reader must keep in mind that it is just that: a model.

4 Static Checks Eliminate Dynamic Checks

Checking the validity of a *Sample* program with declarations comes with the additional benefit of simplifying the machine. Recall the CS and CSK machine designs from the preceding chapter. In both cases, the machine has to have transitions for the cases when variables don't have a value in the store yet. Return to figure 10, and inspect the right column. It presents two pairs of transitions for evaluating expressions: each pair consists of a "success" and a "failure" case.

While the presented argument is informal, it is sufficiently rigorous and a theoretician of programming languages could turn it into a mathematical proof.

The addition of variable declarations to *Sample* and checking them *before* the program is handed to the abstract machine eliminates the need for these case distinctions. All variable declarations precede all the program's statements and final expression. The expressions in the sequence of variable declarations may refer only to already declared variables. Finally, each variable declaration immediately associates a value with the declared variable, so that all variable references are guaranteed to have a value in the store register S of an abstract machine.

search for expression, place value

	Control	Store	Kontinuation
search ends with return expression			
before:	†	s	([ ] e)
after:	e	s	([ ] e)
search ends with right-hand side of assignment			
before:	†	s	((x = ex)::stmt* e)
after:	ex	s	((x = ex)::stmt* e)
value for right-hand side of assignment			
before:	n	s	((x = ex)::stmt* e)
after:	†	s[x = n]	(stmt* e)

evaluating expressions

	Control	Store	Kontinuation
evaluate a variable			
before:	y	s	k
after:	n	s	k
where n = s[y]			
evaluate an addition			
before:	(y + z)	s	k
after:	+	s	k

Figure 18: The CSK transitions for well-formed and valid *Sample* programs

Figure 18 presents the CSK transition function for well-formed and valid *Sample* programs. Take a close look. The machine comes *without* transitions for variable declarations. Once the checker confirms an AST's validity, one

can think of variable declarations as just assignment statements in this simple case of *Sample*. In other words, a simple rewrite of all def into ass nodes turns the AST into a program that can run on the simplified CSK machine.

Stop! Design and implement an AST traversal that performs this rewrite for your data representation.

5 Project Language: *Declared*

Let's expand on the *Bare Bones* project from section 5. This section's *Bare Bones* language extends *Sample* with conditionals and loops. Just as with the latter language, *Bare Bones* programs may contain variable references that have no value in the CSK store (of exercise 4).

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

Declaration ::= (def Variable 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 Names, minus keywords.

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

Figure 19: The concrete syntax of the *Declared* language

Figure 19 presents the BNF for the *Declared* language, which extends the grammar for *Bare Bones* with variable declarations. Only the first two productions differ; the remaining ones are adapted as is.

The following exercises guide you through the process of adapting the executable model for *Bare Bones* to this new language.

Exercise 5. Your task is to enrich your AST data representation for *Bare Bones* from exercise 1 and the parser to accommodate the variable declarations of *Declared*.—The parser maps an S-expression to an instance of AST, an abstract syntax tree that may contain error nodes.

Exercise 6. Design and implement a validity checker for *Declared* using the AST representation you chose for exercise 5.

Exercise 7. Design and implement a function that replaces all variable declarations with assignment nodes in a *Declared* program AST.

Exercise 8. Your task is to simplify the CSK machine for *Bare Bones* for running only well-formed and valid *Declared* programs.

Exercise 9. Modify the `main` function from figure 12 so that it can run the entire process of parsing, validating, and determining the meaning of a *Declared* program.