Why should programmers declare variables close to their use?

1 The Pragmatics Question

Imagine yourself editing a *Declared* program of 10,000 lines. Okay, nobody should write such programs but in the past people had to do just that, and therefore language creators had to confront a problem that these programmers encountered time and again. So again, imagine yourself editing this large *Declared* program and realizing that the values of two variables need to swapped. Easy, you say—just use a temporary variable to hold the value of one, and so on. And this raises the question of where this temporary variable comes from.

A programmer can tackle this problem in one of two ways. The first one means to navigate to the top of the program, insert a new variable declaration with an appropriate name, and navigate back to the place where the value swap is to be added. For a long program, this navigation effort might be significant. The second is a mental exercise: the programmer remembers the name of a variable whose current value is no longer needed and uses it. If the programmer is wrong in this second case, the computation is going to go wrong.

What this scenario describes is concern about the language in a particular work situation. The language forces the developer into a needless navigation or a potentially dangerous mental exercise. It sets up a program organization that provides no information to the programmer about variable names and uses. Let's take a look at how to solve this problem.

1.1 Design Choices

Some programmers work in a specific programming language only with one particular IDE. If we were to accept this combination as a common standard, there are two different ways to address the problem:

1. add a linguistic construct to the programming language for declaring variables locally; or

2. equip the IDE with a tool (or tools) that assist developers with the use and re-use of variables.

As you probably know from experience, language creators solved this problem with local variable declarations. The question is why they preferred the linguistic solution over the tools-based one.

An alternative, but equivalent question is which IDE developers prefer. Or, whether a team of developers can agree to use the same IDE for a project. If the team lead is a strong-willed manager, the team's members may just not have a choice. If the team consists of strong-willed individuals, they are unlikely to agree on an IDE. The point is that if an IDE is used for work with a language like *Declared*, the language creators must equip it with a tool for managing variable names. And given the speed of evolution in the field of IDEs makes no sense. Hence, it is the first solution that eventually prevailed.

1.2 Costs and Benefits

Declaring variables locally has clear advantages for programmers. It first of all means a programmer does not have to remember all variables that are visible at a certain place. Just in case a local variable name is the same as one already in use, the first replaces the second. Next, it helps a programmer declare variables when needed and for exactly the narrow region of text where they are needed. And in case, several programmers work on one large product whose code is in a single file, they don't have to worry about each others variable name choices.

For the language creator, every change to a language raises questions about all parts of its implementation:

- Clearly, a change to a language's grammar induces a change to its implementation's parser.

- In the particular case of local variable declarations, such a change also demands a look at the validity checker to make sure it can deal with any new abstract syntax introduced via the changes to the parser.
- Finally, new language features may also require an adaptation of the semantics. While a new semantics was not needed for the addition of program-wide—also called *global*—variable declarations to *Sample*, *local* declarations require a completely new abstract machine.

The remaining sections make these points concrete.

2 Syntax: Enriching *Sample* a Last Time

The design of a language extension starts with syntax. As the preceding chapters, explain "syntax design" has two aspects: (1) the BNF grammar and the parser; and (2) a suitably modified validity check. In addition, the introduction of local variable declarations necessitates the introduction of a bit of terminology: "scope" or "the scope of variable declarations."

2.1 Grammar and Parsing

Here is the grammar for a natural extension of *Sample* with local variable declarations:

```
Program     ::= (Declaration* Statement* Expression)
Declaration ::= (def Variable Expression)
Statement   ::= (Variable = Expression)
              | (block Declaration* Statement+)

Expression  ::= 1 | 2 | 3 | Variable | (Variable + Variable)

The keywords  (`def`, `block`) are not variables.
```

This grammar extends the one from Chapter IV with one alternative for Statements: a block. Such a block starts with a potentially empty sequence of variable declarations followed by a non-empty sequence of statements. Also notice that the symbol "block" is no longer available as a variable, because it is now used as a keyword.

Stop! Some languages allow the use of keywords as variable names. What do you think of this idea? Debate with a partner.

Stop! Note that the grammar allows arbitrarily deep nesting of block statements. Formulate a program according to this BNG that contains block statements nested five levels deep.

A parser for this final revision of *Sample* must recognize one more S-expression as a well-formed Statement. Once the parser recognizes an statement starting with block, it is going to construct an abstract syntax node from the sequence of declarations and statements.

The design of this abstract syntax node should account for the needs of the validity check and even the abstract machine that interprets it. From this perspective, a language creator has two choices:

- The first, obvious one is to develop a data representation for just these new kinds of nodes.

- A second one recognizes the similarity between a Program and a block statement. Both nodes contain a sequence of declarations and statements; a node for Program also includes an AST for the final Expression. One way to unify the two is to parse a block statement into a Program node that contains a distinct marker instead of an Expression AST.

Stop! Think through the pros and cons of each design alternative.

2.2 Scope

A programming language such as *Sample* must come with a description of the scope of variable declarations. Language researchers introduced the word *scope* to refer to the region of text where a declaration binds occurrences of a variable name. Knowing this terminology, and understanding the concept, helps programmers present code in precise terms and assess the problems of programming languages they encounter.

Once a programming language design team has specified the syntax of the language, its next task is to specify scope. While this task sounds straightforward, it turns out that even the global declarations of the preceding chapter come with a hitch. Since two distinct variable declarations may use the same name, the region of the second one cuts a hole into the region of the first one. Take a second, close look at the screenshot of figure 13. It displays a program that declares the same variable—oneVariable—twice. The arrows indicate which declaration binds which occurrence of the same name, and they implicitly define the scope of each declaration:

- The first declaration's scope are lines 4 through the right-hand side of the declaration on line 7.

- By contrast, the second declaration's scope consists of lines 9 and 11.

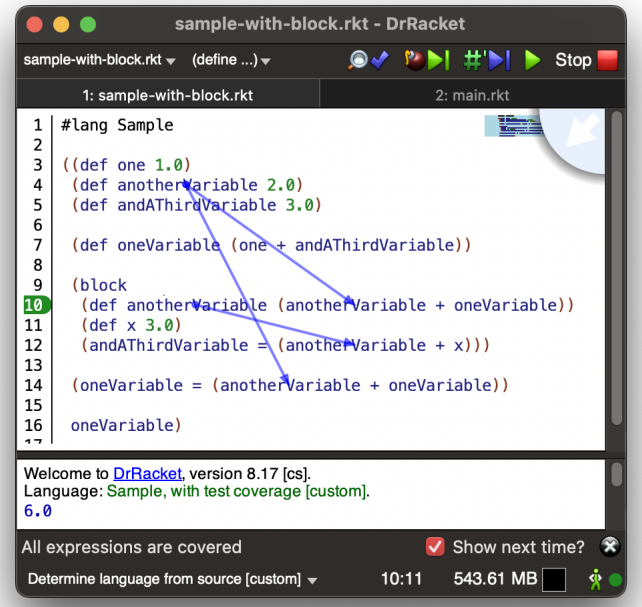

Figure 20: Blocks, scope, and binding in the *Sample* language

The purpose of a block statement is to create such holes in scopes explicitly, as they are needed by programmers without imposing any extra work on them. Consider the following program, which contains one block statement with two variable declarations:

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

 (block
  (def anotherVariable (anotherVariable + oneVariable))
  (def x 3.0)
  (andAThirdVariable = (anotherVariable + x)))

 (oneVariable = (anotherVariable + oneVariable))
 oneVariable)
```

Figure 20 displays the same program in one of *Sample*'s IDEs.

One of the two local declarations introduces the same variable name as the second global declaration. Programming language people say the local declaration shadows the global one. But, the expression on its right-hand side refers to this name, which definitely raises the question of what the scope of each declaration is—and implicitly argues for using well-defined concepts and commonly understood terminology for such linguistic phenomenon.

Figure 20 clarifies the notion of scope for *Sample* in a graphical manner. The scope of a variable declaration in a block statement consists of all declarations and statements that follow, meaning the right-hand side of a declaration is *not* included in the scope.

The two arrows originating from the global declaration of anotherVariable demonstrate a second point about scope. While one arrows points to the right-hand side of the shadowing declaration, the second reaches across the entire block statement. That is, a global declaration's scope may include all pieces of code outside of a nested block statement.

Finally, figure 21 makes another point concerning scope using the same program. The arrow points from the declaration of andAThirdVariable to the assignment statement *inside* of the nested block statement. It binds the left-hand side, meaning the variable whose value is to be changed.

What this discussion shows, is that an abstract machine must be able deal with nested scopes. In other words, a variable may go in and out of scope as the machine evaluates the sub-expressions of a program and changes the association between variables and values. The next section will explain the implications of this statement for the abstract-machine design. Before we deal with the machine, though, we need to briefly address validity checking for *Sample*.

2.3 Validity Checking

The goal of validity checking remains the same: to ensure that all variable occurrences are associate with some variable declaration. Since the design of the validity checker in the preceding section follows fundamental design principles, adapting it to the revised BNF grammar is straightforward. The production for Statement is the only one that got modified. Hence, closed-stmt is the only function that must change.

Here is the adapted function:

```
#; { Stmt- Set<Variable> -> Stmt }
(define (closed-stmt stmt- declared)
```

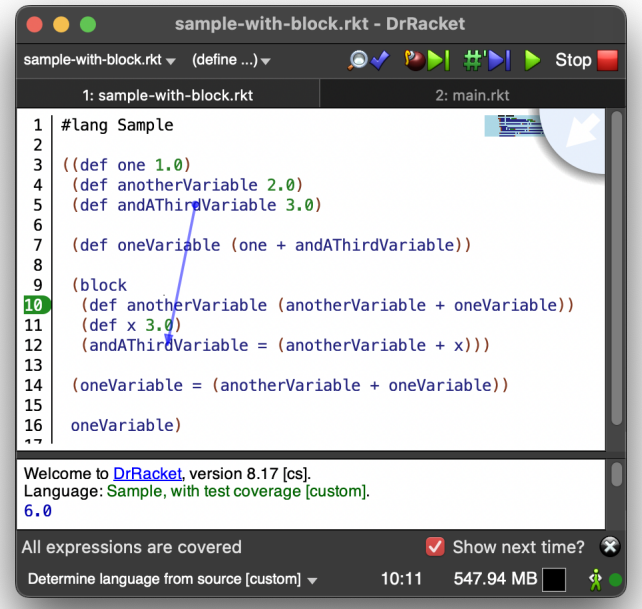

Figure 21: Blocks and assignment statements in the *Sample* language

```
(match stmt-
  [(ass x rhs)
   (define x++    (closed-var x declared))
   (define rhs++ (closed-expr rhs declared))
   (ass x++ rhs++)]
  [(seq decl-* stmt-*)
   (define-values (decl* declared++)
     (closed-decl* decl-* '[] declared))
   (define stmt* (closed-stmt* stmt-* declared++))
   (seq decl* stmt*)])
```

The function consumes an abstract syntax tree free of error nodes. It represents a single statement in *Sample*. While the production for Statement in the preceding chapter has only one alternative on the right-hand side, the one from this chapter has two. Hence the adapted function employs a conditional to distinguish the two cases:

1. The first case is exactly the original code for this function.

2. The second, boxed one corresponds to the newly added alternative form of Statement.

This second case deconstructs the abstract syntax node (seq) into its two pieces: a sequence of error-free declarations (decl-*) and a sequence of error-free statements (stmt*). Since functions for checking the validity of sequences of declarations and sequences of statements exist, checking these pieces is delegated to those. The last line re-assembles the generated ASTs into a new representation of a block statement. If the functions didn't discover any errors, the given node and the returned one are identical; otherwise the latter contains error nodes so that the IDE and the compiler can inform the programmer of infractions. See figure 16 for an example.

3 The CESK Abstract Machine: Local Declarations

A semantics for syntax with local variable declarations must reflect the concept of scope in some ways. Graphically speaking, the semantics of binding must faithfully mirror the arrows of figure 20. So, unlike in the preceding chapter, it is no longer possible to run *Sample* programs with (potentially deeply) nested block statements on the CSK machine by just changing variable declarations into assignment statements. Doing so would conflate two distinct variable declarations.

One way to model the semantics without giving up the CSK machine is to change the names of all variables so that each declaration introduces a distinct name. Since

1. the number of variable declarations (and their scopes) is finite and

2. the set of Variables is infinitely large,

such a renaming transformation is clearly feasible. This transformation would have to respect the scope structure, or, equivalently, the binding structure, of the given program, meaning names bound to the same declaration are replaced by the same new and distinct symbol. Language researchers call this approach a *static semantics* of scope.

The alternative is to transform the CSK machine into a model that assigns a semantics to variable scopes. Assigning meaning to scope as a part of the program execution (in a model) is far more powerful than the static

approach. In particular, this *dynamic* approach can accommodate local variable declarations that appear during execution and which, by definition, could be nested infinitely deep. While the current syntax does not enable such executions, revisions in subsequent chapters do. Hence, we opt for this second, dynamic approach to assigning semantics to scope. The new machine has been known as the CESK machine since the 1980s.

3.1 The States

The CESK machine comes with four registers, just as its name suggests: C for control, E for environment, S for store, and K for continuation.

As before, the C contains expressions, including numerical literals that result from the evaluation of expressions, plus the special † token, which indicates that the machine is searching for the next expression to evaluate.

The new E register contains an association of variables and locations. More precisely, it maps just those variables to locations that are in scope for the evaluation of the expression in C. Assume that the set of locations is infinite, symbolic, and distinct from the set of Variables. The content of this register denotes the meaning of scope.

In contrast to the CS and CSK machines, the S register of a CESK machine associates locations with numbers. The number associated with a location is the current value of the variable that is mapped to this location.

While environments come and ago, reflecting the evaluation of nested block statements, locations never disappear. But, it may no longer be possible to reach a location by starting from the variable names mentioned in the contents of the other registers. The period from when a location comes into existence until it becomes unreachable is the *dynamic extent* of the lifetime of a particular variable declaration in a program.

Note At this stage, the indirection via locations is a convenience, not a necessity, because the scope of a variable coincides with its dynamic extent. It does reflect that the same variable name may exist at the same time and stand for distinct values, even if only one of them is visible at any one place. For the language extensions in the next chapter, the existence of a location becomes a necessity.

Finally, the K register contains not just a program, but stacks of program-like shapes combined with environments:

```
closure:  (def_n* stmt_n* $)  in  e_n
...
closure:  (def_1* stmt_1* $)  in  e_1
closure:  (def_0* stmt_0* expr)  in  [ ]
```

These combinations are known as *closures*; their exact data representation of closures doesn't matter. We refer to each layer in this stack as a *frame*. The stack reflects the nesting of block statements in the program, and the frames correspond to scope extensions via local variable declarations.

Let's take a close look at the shape of these frames, starting with

```
closure:  (def*_0 stmt*_0 expr)  in  [ ]
```

It represents the part of the original program whose expressions haven't been evaluated yet; the machine needs to continue evaluating those when the frames atop of this one are evaluated. Each frame on top of the original program frame contains the $ marker in the expression position instead of an expression:

```
closure:  (def*_i stmt*_i $)  in  e_i
```

This marker signals that these frames represent the nest of block statements whose evaluation has been started and yet contain expressions (inside of the declarations and statements) that remain to be evaluated.

The search for expression remains a process driven by the contents of K. More precisely, the machine searches in the top-most frame until the sequences of declarations and statements are exhausted. Writing down the complete stack for these search is therefore unnecessary. It suffices to focus on the top-most frame, and it is useful to introduce a notation to formulate the cases of the transition function in a concise manner. So,

```
<<  (def*_n stmt*_n $)  >>
```

abbreviates the entire stack from the 0th layer to the nth one:

```
closure:  (def*_n stmt*_n $)  in  e_n
...
closure:  (def*_1 stmt*_1 $)  in  e_1
closure:  (def*_0 stmt*_0 expr)  in  [ ]
```

And,

```
<<  (def*_0 stmt*_0 expr)  >>
```

is short for

```
closure:  (def*_0 stmt*_0 expr)  in  [ ]
```

Note how each frame contains an environment distinct from the environment in any other frame.

Partitioning this set of states into its three pieces follows the patterns from the preceding chapters. An *initial* CESK state has this shape, given some Program p:

```
†, [ ], [ ], (closure:  p  in  [ ] )
```

that is, it combines the search-for-an-expression marker with two empty association tables, and the full-program closure as the first and only frame in the continuation stack. The set of proper *final* CESK states has a number in the C register and an empty stack in K:

```
n, e, s,  [ ]
```

For these final states, it doesn't matter what E and S contain. Additionally, the set of final states includes error states:

```
error, e, s, [ ]
```

Recall that when an abstract machine transitions to an error state, runMachine stops and issues an informative message.

Stop! Choose a data representation for the states of the CESK machine in your favorite programming language.

Stop! Define load and unload functions for the CESK machine.

3.2 The Transition Function

One way to develop the transition function for the CESK machine is to adapt the CSK transition function to deal with the separation of the store into an environment and a store proper. Let's start with the transition for executing assignment statements:

	Control	Environment	Store	Kontinuation
search ends with right-hand side of assignment				
before:	†	e	s	⟪ [ ], (x = rhs)::stmt*, r ⟫
after:	rhs	e	s	⟪ [ ], (x = rhs)::stmt*, r ⟫
value for right-hand side of assignment				
before:	n	e	s	⟪ [ ], (x = rhs)::stmt*, r ⟫
after:	†	e	s[xl = n]	⟪ [ ], stmt*, r ⟫
where xl = e[x]				

The *before:* condition of the first case describes a search scenario when the next expression—rhs—has been found. Considering that the corresponding case in the CSK transition function does not involve the store, it is unsurprising that the *after:* specification says that C contains rhs and otherwise the state remains the same.

By contrast, the second case above involves the store in the corresponding CSK case, so the adaptation needs work. The *before:* condition tells us that the value of the right-hand side of the assignment statement has been found; it is the number n. Intuitively, we understand that from now on going forward, the variable on the left-hand side of the assignment statement—symbolized with x—should be associated with this number. But, environments associate variables with locations, and stores map locations to values. Hence, the transition function first determines the location of the variable via the environment:

```
xl = e[x]
```

and then creates a store that associates this location with n (and retains all other associations):

```
s[xl = n]
```

Stop! Why is (x = rhs) missing from the *after:* state description?

The adaptation of the rules for the evaluation of expressions proceeds in a similar manner:

	Control	Environment	Store	Kontinuation
evaluate a variable				
before:	y	e	s	k
after:	n	e	s	k
where	yl = e[y]			
and	n = s[yl]			
evaluate an addition				
before:	(y + z)	e	s	k
after:	+	e	s	k
where	yl = e[y]			
and	zl = e[z]			
and	yn = s[yl]			
and	zn = s[zl]			

When the CESK machine encounters a variable in C, determining its value proceeds in two steps: (1) find its location in the current environment (e) and (2) extract the value of this location from the current store (s). Similarly, an addition expression causes the abstract machine to retrieve the locations of the two variables and then to look into the store to obtain the actual value. Once these values are available, the transition function uses

an abstract addition +; an implementation would use the addition function for inexact numbers (doubles).

	Control	Environment	Store	Kontinuation
search ends with def rhs				
before:	†	e	s	⟪ (def x rhs)::def*, stmt*, r ⟫
after:	rhs	e	s	⟪ (def x rhs)::def*, stmt*, r ⟫
value for right-hand side of declaration				
before:	n	e	s	⟪ (def x rhs)::def*, stmt*, r ⟫
after:	†	e[x = xl]	s[xl = n]	⟪ def*, stmt*, r ⟫
where xl = a new (relative to s) location				
search encounters nested block				
before:	†	e	s	⟪ [ ], (block d* s*)::stmt*, r ⟫
after:	†	e	s	push[cl, k]
where k = ⟪ [ ], stmt*, r ⟫				
and cl = *closure*: (d* s* $) *in* e				
search exhausts nested block				
before:	†	e	s	⟪ [ ], [ ], $ ⟫
after:	†	e1	s	pop[k]
where e1 = the environment in the top-most closure				
and k = ⟪ [ ], [ ], $ ⟫				
search ends with the return expression				
before:	†	e	s	⟪ [ ], [ ], r ⟫
after:	r	e	s	⟪ [ ], [ ], r ⟫
search ends with evaluated return expression				
before:	v	e	s	⟪ [ ], [ ], r ⟫
after:	v	e1	s	pop[k]
where e1 = the environment in the top-most closure				
and k = ⟪ [ ], [ ], r ⟫				

Figure 22: The CESK transition function for variable declarations

At this point, you should wonder how locations come into existence, and the answer can be found in figure 22. This figure displays all the cases

of the CESK transition function related to variable declarations, both global and local. The first two cases explain how the transition function deals with variable declarations in the top-most stack frame:

1. When the machine searches for the next expression to be evaluated and the sequence of declarations in the top-most closure is not empty, the right-hand side of the declaration becomes the content of the C register. Note the similarity to the cases for assignment statements.

2. Once the value is found, the transition function allocates a new location, that is, a location not used in the current store s. Using this new location the *after:* state specification tells us that

   - the newly created environment associates the left-hand side variable with the new location, and
   - the newly created store associates the location with the value (n) in the C register.

   Note also that the machine switches into search mode again.

Stop! How would you implement "find a new location" in your favorite programming language? What does this implementation depend on?

The second pair of cases in figure 22 concerns local variable declarations, specifically nested block statements:

1. A machine in search mode that encounters a top-level stack frame whose sequence of declarations is empty and its sequence of statements starts with block has to continue the search inside this block statement. Hence the *after:* state specification pushes a new frame on the stack. This frame combines the current environment with a quasi-program formed from the block statement: note the $ marker in the final-expression position. From here the search continues.

2. The CESK machine recognizes that a nested block's instructions have been executed when the top-most stack frame contains an empty sequence of declarations, an empty sequence of statements, and the marker $. If a machine states satisfies this *before:* description, the transition function creates a state that pops the top-most frame. Concretely, this means

   - the environment of the top-most stack frame becomes the current environment in register E, and

- then the top-most frame disappears.

Note that the machine remains in search mode.

In sum, the execution of a variable declaration causes a machine to allocate locations. Otherwise the use of variables just requires an indirection of looking into the environment first and then into the stack.

Stop! The case for the final return expression is missing. Formulate the missing case for the CESK transition function. Let the corresponding case for the CSK machine guide your design.

Stop! Design and implement the transition function in your favorite programming language using your data representations from the preceding subsection.

name(s) and term(s)	standing in for
c	a generic control
e, e1	generic environments
s, s1	generic stores
k	a generic continuation
cl	a closure
n, xn, yn, zn	numbers
x, y, z	program variables
xl, yl, zl	locations
r, rhs, tst	expressions
def*	a potentially empty sequence of definitions
stmt, thn, els, body	statements
stmt*	a potentially empty sequence of statements
e[x]	the location at variable x in environment e
s[xl]	the value at location l in store s
e[x = xl]	an environment like e, but x stands for xl
s[xl = n]	a store like s, but xl stands for n

Figure 23: Conventions for the CESK transition function for *Core*

Stop! Take a look at figure 23, which lists the notational conventions used to describe the transition function for the CESK machine. As you

tackle this chapter's project next and the projects in the following chapters, keep this table in mind and consult it often.

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

Declaration ::= (def Variable Expression)

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

Block       ::= Statement
              | (block Declaration* 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 24: The grammar of the *Core* language

4 Project Language: *Core*

Figure 24 presents *Core*, the model language that represents the primitive but universally shared syntax of most widely used programming languages. It adds local variable declarations to the *Declared* language, and it adds a condition and a looping construct to the latest version of *Sample*. Additionally, its Expression sub-language comes with two new forms:

- a division expression, which divides two numbers, and
- a comparison expression, which compares two numbers, yielding 0.0 if they are the same and 1.0 otherwise.

It really is wrong to compare inexact numbers (doubles) for equality. This topic is beyond the scope of the book, however; it belongs into a first course on programming.

A straightforward adaptation of the CESK machine for the *Sample* language yields a semantics for *Core*. In a sense, the set of states just contain

additional forms of syntax; nothing else changes. The transition function needs cases for the evaluation of the two new forms of expression, and it needs cases for searching the next expression in conditionals and loops. That's it, but also see the next subsection.

	Control	Environment	Store	Kontinuation
search for expression in if				
before:	†	e	s	⟪ [ ], (if0 tst thn els)::stmt*, r ⟫
after:	tst	e	s	⟪ [ ], (if0 tst thn els)::stmt*, r ⟫
pick then branch from if0				
before:	n	e	s	⟪ [ ], (if0 tst thn els)::stmt*, r ⟫
subject to:	n is 0.0			
after:	†	e	s	⟪ [ ], thn::stmt*, r ⟫
pick else branch from if0				
before:	n	e	s	⟪ [ ], (if0 tst thn els)::stmt*, r ⟫
subject to:	n is not 0.0			
after:	†	e	s	⟪ [ ], els::stmt*, r ⟫

Figure 25: The CESK transition function for *Core*: conditionals

Figure 25 gathers the cases for the transition function that describe how it deals with conditionals. The first case explains what happens when the machine is in search mode and encounters a conditional in the top-most frame of K as the next instruction. As the *after* specification shows, the conditional's test expression becomes the content of C.

Once the machine has evaluated this expression, it picks one of the two branches and discards the remainder of the conditional. The chosen branch becomes the first of the sequence of statements in the top-most frame of K, while the C register is set to †, meaning the machine is back in search mode.

Stop! Take a close look at the figure. How does the machine differentiate the two cases?

The cases in figure 26 explain how the transition function treats while0 loops. In a way, the three cases mirror the cases for conditionals, though the generated states differ. Again, if the machine is searching for the next expression and encounters a looping statement as the first one in K, the loop's test expression is placed into C.

	Control	Environment	Store	Kontinuation
search for expression in while				
before:	†	e	s	⟪ [ ], (while0 tst body)::stmt*, r ⟫
after:	tst	e	s	⟪ [ ], (while0 tst body)::stmt*, r ⟫
decide whether to run while loop (positive)				
before:	n	e	s	⟪ [ ], (while0 tst body)::stmt*, r ⟫
subject to:	n is 0.0			
after:	†	e	s	⟪ [ ], body::o, r ⟫
where	o = (while0 tst body)::stmt*			
decide whether to run while loop (negative)				
before:	n	e	s	⟪ [ ], (while0 tst body)::stmt*, r ⟫
subject to:	n is not 0.0			
after:	†	e	s	⟪ [ ], stmt*, r ⟫

Figure 26: The CESK transition function for *Core*: loops

What happens when the value of the test expression is found, is the interesting part. The first case—when the value is 0—tells us that the *after:* state contains a newly synthesized statement in the top-most frame of K. In a conventional syntax, we might write this:

```
body; while0 (tst) body
```

That is, the body part of the while0 statement becomes the first instruction of the sequence of instructions. Furthermore, the original while0 statement is now the second one; after all, once body is executed the loop may have to run again. And, because body is a Block and there is no obvious expression to evaluate next, the machine is instructed to search for the next expression. Stop! Take a close look at the second case in figure 26 to make sure you understand how it works.

Finally, the last case in figure 26 informs an implementer of what happens the value of the test expression in a while0 statement is not 0. The *after:* state specification tells us that the while0 statement is discarded from the top-most stack frame, leaving just the remainder of the sequence of statements (stmt*) in place.

4.1 Why Core? Why CESK?

Core can express all classic computations, meaning the partial-recursive functions of the Church-Turing hypothesis. Also, its syntax and semantics is one that all programmers immediately recognize. In this spirit, *Core* can serve as the foundation from which we investigate truly interesting questions of pragmatics.

Stop! These claims assume that we understand what each *Core* statement and expression computes. But we haven't articulated the cases for comparison and division yet. Do so.

This book focuses on pragmatics, not expressive power. Readers interested in the second topic may wish to check the literature.

While programmers may acknowledge that, in theory, *Core* is everything needed, none of them would want to develop a piece of software in this impoverished syntax or variants thereof. Software development requires much more than Church-Turing computability, both in terms of expressive syntax and pragmatics.

Also, the CESK semantics is easily adapted to any future extension with expressions and statements. Changing the set of states require a "re-direct" to the revised BNF productions. The actual work consists in adding cases to the CESK transition function:

- Every new kind of expression can show up in the C register, and therefore we need an evaluation rule for those occasions. Like the case for addition, these evaluation rules essentially appeal to a function in mathematics or the implementation language chosen to realize the machine.

- For every new kind of statement, the transition function must be equipped with a rule for finding the next nested expressions to be evaluated and for how to use the value to change the K register once the value of the expression is found.

  If a new kind of statement contains more than one sub-expression, it might be necessary to add several such sets of rules to the transition function.

Stop! How could the CESK machine deal with a for loop or a multi-pronged conditional such as a numeric switch statement?

In sum, *Core* and *CESK* jointly set us up for a proper and reasonably smooth investigation of the pragmatics behind programming language concepts that actually affect software developers.

4.2 The Meaning of Numbers, Errors

Stop! Can your transition function deal with the cases for comparison?

One way to formulate the case for division is to copy the cases for addition from the preceding section and to replace + with /. In the *before:* condition, the symbol refers to what the *Core* programmer writes down, while the / in the *after:* specification refers to the division operation in mathematics or the chosen implementation language.

This standard is known as IEEE 754.

If / denotes mathematical division, every reader knows exactly what it means. But, as you know, computer hardware does not implement real numbers; it represents them as inexact numbers, generally known as IEEE floats or doubles. Almost all programming languages therefore use this standard to represent numbers. Sadly, even addition of such numbers—an operation everyone considers straightforward—isn't the addition we know from grade school.

Stop! Write a short program in your favorite language that adds two very large doubles and prints the result.

Control	Environment	Store	Kontinuation
evaluate a division (success)			
before: (y / z)	e	s	k
subject to: z is not 0.0			
after: double ieee division of yn by zn	e	s	k
where yl = e[y]			
and zl = e[z]			
and yn = s[yl]			
and zn = s[zl]			
evaluate a division (failure)			
before: (y / z)	e	s	k
subject to: z is 0.0			
after: error	e	s	[ ]

Figure 27: The CESK transition function for *Core*: division

Assuming you choose a truly large number the result is +inf.0. Computer scientists refer to this phenomenon as *numeric overflow*, meaning the result is such a large number that computer hardware can't represent it. Similarly, the result of a division operation can be so small that it can't be represented; this is called *numeric underflow*.

Division comes with another problem, too. In grade school we learn that division doesn't work when the divisor is 0. A mathematician says that division is a partially defined operation. Hardware people trick people into thinking that (/ 1.0 0.0) is +inf.0, too.

For pedagogic reasons, this book deviates from the IEEE standard for just this one case. Instead of allowing division by 0.0 to yield "infinity," the CESK machine for *Core* "jumps" signals an error. The specification of this distinction consists of two cases in the CESK transition function; see figure 27. Each case comes with a side-condition labeled "subject to." In the first one the side-condition adds the constraint that the divisor is *not* 0.0, while the second case's side-condition expresses the opposite. The *after:* part of the second case specifies a state whose C register contains an error element, which makes it a final state and causes the machine to stop.

4.3 Project Tasks

Use your favorite programming language to solve the following exercises.

Exercise 10. Design an AST data representation for *Core*. Implement a parser for *Core* that maps an S-expression to an instance of AST.

Exercise 11. Design and implement a validity checker for the *Core* language. Remember that a validity checker consumes an AST *without* error nodes. Its result is the same AST if all variables are properly declared; otherwise it is an AST that contains error node that mark references to undeclared variables.

Exercise 12. Formulating a transition function in your favorite language requires the addition of a case for comparison expressions. Unlike in the case of addition and division expressions, there is no obvious function for comparing two numbers. Clearly, the comparison of two numbers must yield a number, because numbers are the only values in *Core*. Since the point of a comparison is to make decisions—which branch of a conditional to evaluate or whether to continue the execution of a while loop—returning 0 for one of the two cases is also natural. We choose to say the comparison function yields

- 0.0 when the two given numbers are the same;
- 1.0 when the two given numbers differ.

Explain the rationale for specifying the outcome in both cases.

Exercise 13. Design a data representation for the states of the CESK machine for *Core*. Implement the load, transition, and unload functions. By

combining these functions with the runMachine function (see section 2 of Chapter III), you obtain a complete semantics for well-formed and valid *Core* programs.

Exercise 14. Adapt the main function from figure 12 so that it can run the entire process of parsing, validating, and determining the meaning of a *Core* program.

5 Scope in the Real World

This chapter presents the most common meaning of scope. It is often called "lexical scope," because each variable declaration connects to an easily identified region of program text. While the text may contain holes due to a re-use of a variable name, the direct association enables programmers to think about basic relationships among variables and the flow of values from one to another.

A few languages stick to lexical scope for their core syntax. C# and Java are examples. Many don't. JavaScript and Python come to mind. The former comes with deprecated linguistic constructs that make it extremely difficult for a reader to understand which name refers to which variable, not to speak of the flow of values from one variable to another. The latter does not know variable declarations. When a program assigns a value to a name, it is also not immediately clear where this name is visible. Thus, although both languages support mostly conventional syntax, determining scope and reasoning about value flow by just reading program text can be surprisingly hard.