What does a machine for executing code look like?

1 Meaning Matters

The BNF of the *Bare Bones* language tricks you into thinking that you know the behavior of programs, statements, blocks, and expressions. You might think that (x = 1) sets the variable x to 1 and that (x = (x + y)) adds x to y and makes x stand for this sum. But, thus far, all we know is how to create Programs, Statements, and so on. We follow the rules of the BNF, and a parser for this BNF will confirm that these sentences are well-formed. The point is that syntax may seem to imply semantics, though without an accompanying specification, we can't know for sure.

As mentioned in the first chapter, semanticists study how to assign meaning to programs. In general, their concern is to specify the semantics of any sentence that is well-formed according to some BNF production. So, for example, a semanticist might say that

- a literal constant is a value;
- an expression yields a value; and
- a statement updates a relationship between variables and values.

These English sentences suggest three insights about describing the semantics of a programming language: (1) we must choose a set of values; (2) we must state what operations on these values yield; and (3) we must have a mechanism for tracking which variables stand for which values.

Let's relate this abstract explanation to the real world of computers, starting from this"mechanism" just mentioned. A computer engineer calls this "mechanism" the machine's *memory, also known as store*. One way to imagine store is as a collection of "boxes," each of which has a label—the

name of a variable—and contains a number. A variable reference in an expression is a request to retrieve the number from the correspondingly labeled box. If an expression consists of an operation, say addition, and two variables, the two corresponding numbers are sent to a co-processor that corresponds to the named operation. Now consider that this expression is on the right-hand side of an assignment statement. Once this co-processor delivers the result of this expression, the computer places this number into the box labeled with the name of the left-hand side of the assignment.

While this concrete description isn't wrong, it is overly simplistic and, yet, it is also inspirational. It explains why semanticists consider abstract machines a useful tool for specifying the semantics of a language. People can easily imagine the execution of primitive statements and expressions in the context of this concrete mechanism. It is easy to illustrate and visualize a program execution. However, some of the English is imprecise—how does a co-processor add two inexact numbers—and it isn't clear how it scales to common features of existing languages. Abstract machines—a mathematical idea that directly corresponds to executable code—solves some of these problems. This chapter introduces the basic idea of abstract machine; the next one expands on it so that the rest of the book can explain many of the features of contemporary languages.

2 Abstract Machines: In General

A semantic specification based on abstract machines starts from a grammar that defines the set of programs and a set of values, which are the meanings of programs run on the machine. For the description of the set of values, it might suffice to pick a set that people in computing accept, say the set of inexact numbers within certain limits, or it might require a small, separate BNF grammar.

Given the grammar and the chosen values, an abstract-machine definition consists of two parts: (1) a set of machine states and (2) a function from states to states, often called a *transition function*. The specification of machine states identifies (1a) initial states, (1b) intermediate states, and (1c) final states. Every state always includes the instructions that the program represents, usually as ASTs, and additional "bookkeeping" data.

Running a program on an abstract machine thus requires a function from programs to initial states. This function is typically called load. Once an initial state exists, the machine repeatedly applies the transition function to interpret the program's instructions, one instruction at a time. When the

transition function yields a final state, the machine stops, and an unload function maps this final state to a value.

Figure 7 presents a diagram that expresses this explanation graphically. The program is loaded, yielding an initial state. Then the transformation function is applied to the initial state, yielding an intermediate state. It is again applied to this state and so on, until the result belongs to the set of final states. By applying the unload function to this final state, we determine the value of the program—that is, its meaning.

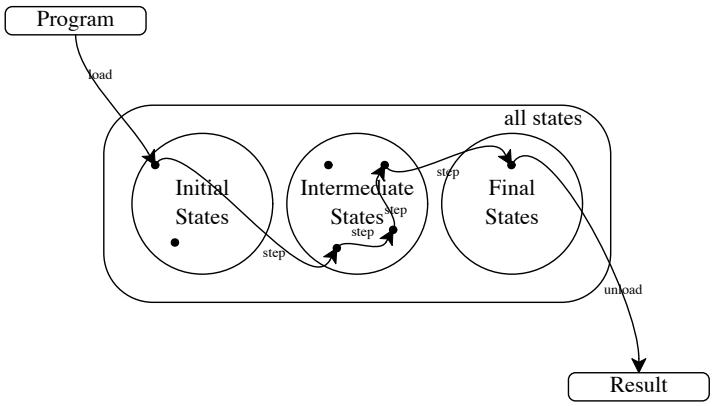

Figure 7: Executing programs on an abstract machine

Although of these concepts—sets, functions—are mathematics, the key is that it is *executable* mathematics, meaning it directly corresponds to code. Here is a sketch:

```
// determines the meaning of `p`
Result runMachine(Program p) {
  State currentState = load(p);
  while (! (isFinal(currentState))) do {
    currentState = transition(currentState);
  }
  Value result = unload(currentState);
  return result;
}

State load(Program p) { ... }
State transition(State s) { ... }
Result unload(State s) { ... }
```

This pseudo-code employs a 1970s style procedural style, easily mimicked in every contemporary mainstream language. The runMachine function is the entry point. It composes three tasks via sequencing and a while loop: load the program; run a loop until the content of currentState belongs to the subset of final states; then unload the result from this final state.

Stop! Articulate this pseudo-code in your favorite programming language, using its preferred style.

Let's ignore exception handlers for now.

What this "story" and the figure fail to account for is the possibility of a misbehaving program. Think of an expression in a program that divides an integer by zero. A few language descriptions, for example C and C++, do not specify the semantics of this situation, leaving it up to the implementation to pick a behavior. Many others, say Java and Python, describe this situation as an error situation and demand that implementations discontinue execution and signal an exception.

With abstract machines, it is straightforward to express this semantics. First, let's enrich the set of final states with designated and recognizable "error states" that bring the machine to a halt. Second, the unload function translates such error states into results that are distinct from any value. And that's why the result type of runMachine is Result, not Value. The Result type represents the disjoint sum of Values and Exceptions:

```
type Result = Value | Exception
```

In short, the author of the abstract machine must not only pick a set of values but also a set of exceptions to assign meanings to programs.

3 The CS Abstract Machine: The *Sample* Language

In a sense, section 1 of this chapters precisely explains how programs in the sample BNF of Chapter II are executed. The goal of this section is to articulate this English description as an abstract machine, using a rigorous formalism that directly corresponds to code.

And set-based mathematics for theoretical purposes.

To keep things simple, let's assume that the addition expressions always have the shape (x + y) for some variable names x and y:

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

The set of Variables consists of all non-empty sequence of
alphanumeric characters, starting with an alphabetical letter.
```

Let's call this the *Sample* language from now on. What we need next are a description of the set of values and states; an examination of whether there are exceptional cases; and a transition function.

3.1 The States

The *Sample* BNF chapter allows programmers to write down exactly three literal constants: 1.0, 2.0, and 3.0. Furthermore, the programs may perform exactly one operation on these values: addition. All other statements merely move such values from one variable to another. Hence, it makes sense to say that the set of Values consists of just the inexact numbers found in computer hardware and that + denotes their addition operation.

See IEEE 754 for this standard. This text will not address the numerous complications that inexact numbers impose on programmers.

Following the introductory section of this chapter, a state obviously consists of two parts: the program's remaining instructions and the store. Semanticists call these parts *registers*, even if this usage isn't quite in conformance to how computer engineers apply it. Specifically, we say that the C register contains the remaining instructions and thus controls the execution of the machine; the S register represents the current settings of the store. The name of the machine reflects this combination: CS machine.

A store, as informally described, is a collection of "boxes" or "cells" with labels on them. If we were to use mathematics here, we could say it is a function from variable names to the set of Values, with a finite domain. By contrast, a programmer would represent such a collect as some form of table, which many programming languages offer as primitive objects with relevant operations.

One task remains: identifying the three partitions of the set of states and defining the load and unload functions. Clearly, an *initial CS state* consists of just a program p combined with an empty store, representing that no instruction has been executed. A mathematically inclined person may write this as a Cartesian pair:

```
p, [ ]
```

A *final CS state* is reached when all instructions of a program have been evaluated, meaning all assignment statements and all expressions:

```
n, s
```

where n is the number that results from evaluating the final expression in a *Sample* program. Intermediate states are those that combine programs with instructions and any store. Here are the function definitions:

```
load : Program --> State          unload : State --> Result
load(p) = p, [ ]                  unload(n,s) = n
```

If you'd rather think of setting variables and referring to them, you may want to skip ahead to the last subsection.

3.2 Exceptional Cases: Another Case of Pragmatics

Although the *sample* BNF supports only one operation on values, a programmer can still write faulty programs. Take a close look at these two programs, which are almost identical:

produces a number dubbed "good"	produces what? dubbed "bad"

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

Stop! Explain the difference.

The boxed variable reference in bad gives it away. A typo—note the missing letter—makes it impossible to evaluate the addition expression, because the store does not have a value for this variable.

Language designers have faced this problem for decades and have come up with various solutions. One of them would be to pick a random number for temprary and to continue the execution of the program. Another one would be to let the machine seg fault. Over time, though, it has become clear that a language should let the programmer know that the program refers to an unknown variable. In the context of an abstract-machine creation, this means shutting down the machine gracefully when it encounters such a situation, via the transition to an appropriate final state. We therefore extend the set of final states with error, s where error contains informative texts. Let's modify unload appropriately:

```
Result =                  unload : State --> Result
 InexactNumber
 U                        unload(n,s)   = n
 String                   unload(error,s) = .. some string ..
```

3.3 The Transition Function

Now we're in a position to define the transition function. Such a function always performs two tasks: (1) determining the nature of the next instruc-

tion in the given state and (2) creating a new state in response. This new state is similar to the old one, but with the interpreted instruction pruned.

Due to this characteristics, transition functions are typically expressed as tables that specify a number of "before" and "after" state cases. Computer engineers use similar tables, though of course with states that specify bits not code.

executing a sequence of statements

Control	Store
execute x = n	
before: ((x = n)::stmt* e)	s
after: (stmt* e)	s[x = n]
execute x = y (success)	
before: ((x = y)::stmt* e)	s
subject to: y is defined in s	
after: (stmt* e)	s[x = n]
where n = s[y]	
execute x = y (failure)	
before: ((x = y)::stmt* e)	s
subject to: y is not defined	
after: error	s
execute x = y + z (success)	
before: ((x = (y + z))::stmt* e)	s
subject to: y and z are defined in s	
after: (stmt* e)	s[x = k]
where n = s[y]	
and m = s[z]	
and k = +	
execute x = y + z (failure)	
before: ((x = (y + z))::stmt* e)	s
subject to: y or z is not defined in s	
after: error	s

evaluating return expressions

Control	Store
evaluate y as return expression	
before: ([ ] y)	s
subject to: y is defined in s	
after: ([ ] n)	s
where n = s[y]	
evaluate y as return expression	
before: ([ ] y)	s
subject to: y is not defined	
after: error	s
evaluate y + z as return expression	
before: ([ ] (y + z))	s
subject to: y and z are defined in s	
after: ([ ] k)	s
where n = s[y]	
and m = s[z]	
and k = +	
evaluate y + z as return expression	
before: ([ ] (y + z))	s
subject to: y or z is not defined in s	
after: error	s

Figure 8: The CS transition function for *Sample*

Figure 8 presents the transition function for the *sample* BNF with its intended semantics; see the first section. Its definition distinguishes nine cases: five in the left column and four in the right one. Each case states a *before* condition on the given state and an *after* state specification:

- The *before* condition always specifies a specific program shape and a general store (s).

  Some cases also subject this state pattern to additional conditions; for example, if the program shape contains a specific variable, the additional constraint may require this variable to have a value in the store.

- The *after* state specification explains how the next state is formed, if the given state satisfies the *before* condition. Concretely, it specifies the content of the registers:

  - The C register typically contains some program code or an error, in which case the state is final and the machine stops.
  - The S register always contains the given store or a revised store. A revision has the shape s[x=n], denoting a store that is like s, but x associated with n.

  Some cases also come with *where* clauses. Each such clause defines the value of a variable. For example, s[x] retrieves the value of x from the table named s, and (+ m n) denotes an actual addition of two inexact numbers according to the chosen standard.

The definition function uses one additional notation: a::stmt* is a list of statements where a is the first one and stmt* denotes the remainder.

Note that the abstract machine uses concrete syntax. An implementation will use ASTs instead so that it doesn't have to account for ill-formed programs; see the next section.

While the left column of the figure contains the cases for executing assignment instructions, the right column concerns the evaluation of the final expression in a program. Let's take a close look at the first transition on the left, using good from the preceding subsection. The initial state is a state where the C register contains all of good and the S register the empty table. By matching this initial state with the *before* line of the transition, we get the following:

abstract state description		concrete pieces of state
`(x = n)`	`====`	`(a = 1.0)`

`stmt*`	`====`	`((b = 2.0)`
		`(temporary = (a + b))`
		`(c = (a + temporary))`
		`(temporary = 3.0))`

`e`	`====`	`(c + temporary)`
`s`	`====`	`[ ]`

Since the matching succeeds, the transition function returns the *after* state of this same part of the table. Now C contains

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

And S is set to the table [a = 1.0].

An application of the transition function to this new state proceeds in analogous manner. Interpreting (b = 2.0) results in a shorter program in C and one new entry into the table that is in S.

The interpretation of (temporary = (a + b)) differs from the first two steps. The first (case of the) transition function does not apply. Instead the fourth case matches as follows:

abstract state description		concrete pieces of state
`(x = (y + z))`	`====`	`(temporary = (a + b))`
`stmt*`	`====`	`((z = (a + temporary))` `(temporary = 3.0))`
`e`	`====`	`(c + temporary)`
`s`	`====`	`[a = 1.0, b = 2.0]`

The guidance of the *after* part of the case tells a reader to look up a and b in s, to add the two numbers, and to store the result as the value of temporary in the table. Accordingly, the resulting state consists of

```
((temporary = 3.0)
 (c + temporary))
```

in register C and [a = 1.0, b = 2.0, temporary = 3.0] in S.

Stop! Which case of the transition function is needed to interpret the state we just constructed? And the state that results from this step?

Once the C register contains a program that consists of just an expression —an addition expression or a variable reference to be precise—an interpretation of such a state must use the right-hand side of the table in

figure 8. If you solved the above riddle, you realize that the result is just such a state and here is how it matches the third case's *before* pattern:

abstract state description		concrete pieces of state
`e`	`====`	`(c + temporary)`
`s`	`====`	`[a = 1.0, b = 2.0, c = 4.0, temporary = 3.0]`

According to *after*, the resulting state is a final state, with 7.0 in the expression position of the control register. Hence, unload of this state yields 7.0 as the meaning of this program.

Stop! Run good in your favorite programming language and confirm the result—if you don't trust the transition function or our use of it.

Let's next consider how the transition function would deal with bad. For the first three assignment statements, the application of transition function yields the same sequence of states as for good. The fourth intermediate state matches the last clause on the left, because the concrete variable corresponding to z is not defined in the store:

abstract state description		concrete pieces of state
`(x = (y + z))`	`====`	`(c = (a + temprary))`
`z`	`====`	`temprary`
`stmt*`	`====`	`((temporary = 3.0))`
`e`	`====`	`(c + temporary)`
`s`	`====`	`[a = 1.0, b = 2.0, temporary = 3.0]`

Hence, the result of the transition function is <error,s>, a final state. Unloading this state informs the programmer that something went wrong during the interpretation of bad.

In general, the numbered cases in a transition function must be considered in order. They come with "subject to" conditions in the *before* part, and if any of these conditions doesn't hold, the next case is considered. Other cases could be considered independently.—This convention won't surprise anyone who has written a conditional in a programming language, because in all mainstream languages, conditional cases are considered in order, too.

3.4 An Implementation of the CS Machine for *Sample*

An implementation of an abstract machine requires picking a data representation of the sets of values and states; a data representation for the store;

and function definitions for load, transition, and unload. To run the machine, it is also necessary to port the code of runMachine from the preceding section to the chosen programming language.

This section presents the Racket implementation of the CS machine. Racket structures serve as simple data representations of states:

```
(struct state [control store])
#; {type State  = (U Initial InterM Final)}
#; {type Iniial = [state Program MT]}
#; {type InterM = [state Program Store]}
#; {type Final  = [state (U Value Error) Store]}
#; {type Value  = InexactReal}
```

The first line declares the struct type; the remaining ones are type-like comments that explain how the structures are instantiated to represent initial, intermediate, and final states. In this example, the struct type and the comments directly encode the informal explanations into Racket.

MT in the store field of the structures denotes the empty table. Like most languages, Racket supports a number of table-like data types. Since the details do not matter, a comment and two signatures suffice:

```
#; {type Store  = [TableOf Symbol Value]}

#; { Store Symbol Value -> Store }
(define (extend table x v) ...)

#; { Store Symbol -> Value }
(define (lookup table x) ...)
```

Lastly, we add a structure type for reporting errors concerning references to undefined variables:

```
(struct err [message code])
#; {type Error  = .. some informative data ..}
```

These data representations basically dictate how to define the load and unload functions:

```
(define (load p) (state p mt-table))

(define (unload s)
  (match s
    [(state (prog '() n) s) n]
    [(state (err msg xtra) s) msg]))
```

The definition of transition is more interesting than these. Due to the already-mentioned algebraic match construct, Racket still makes it easy to translate the cases from figure 8 case by case.

```
#; { State -> State }
(define (transition s)
  (match s
    [(state (prog (cons (ass x (num n)) stmt*) e) s)
     (state (prog stmt* e) (extend s x n))]
    [(and (state (prog (cons (ass x (ref y)) stmt*) e) s)
          (? is-y-defined?))
     (define n (lookup s y))
     (state (prog stmt* e) (extend s x n))]
    [(state (prog (cons (ass x (ref y)) stmt*) e) s)
     (state (err "undefined variable" y) s)]
    [;; BEFORE
     (and (state (prog (cons (ass x (add y z)) stmt*) e) s)
          (? are-y-and-z-defined?))
     ;; AFTER
     (define n (lookup s y))
     (define k (lookup s z))
     (state (prog stmt* e) (extend s x (+ n k)))]
    [(state (prog (cons (ass x (add y z)) stmt*) e) s)
     (state (err "one of the variables is undefined" s) s)]

    ;; evaluate the return expression:

    [(and (state (prog '() (ref y)) s)
          (? is-y-defined?))
     (define n (lookup s y))
     (state (prog '() n) s)]
    [(state (prog '() (ref y)) s)
     (state (err "undefined variable" y) s)]
    [(and (state (prog '() (add (ref y) (ref z))) s)
          (? are-y-and-z-defined?))
     (define k (+ (lookup s y) (lookup s z)))
     (state (prog '() k) s)]
    [(state (prog '() (add y z)) s)
     (state (err "one of the variables is undefined" s) s)]
    ...))
```

Figure 9: A Racket implementation of the CS machine

Figure 9 shows the complete definition of transition. Following the abstract description, the function consumes a state and produces one. It uses match to determine which *before* condition of which case of figure 8 applies. It is the responsibility of the specifier of this table to ensure uniqueness of such conditions or to clarify in which order the *before* conditions are to be checked. Racket's match conditional tries the cases in order, and they are arranged in the same order as the original table.

Let's take a look at one case, specifically the boxed case in figure 9, which implements the case in figure 8 labeled "execute x = y + z (success)".

The *before* condition of this case is narrower than the one labeled "failure," so it must be checked first.

The match pattern, labeled BEFORE, consists of two parts, combined with *and*:

- The first one says that the given state structure must have a shape that looks pretty much like the one in the mathematical specification.
- The second one, (? are-y-and-z-defined?), applies the aptly named predicate to the given state, ensuring that y and z are defined in the store.

The code labeled AFTER constructs the result state in three steps. The first two retrieve the values of y and z, respectively, from the store s in the given state. These retrievals succeed, because the BEFORE condition checked that the variables have values in s. Finally, the function instantiates state with (1) a program that no longer contains the interpreted assignment statement and (2) a new store that associates the sum of the two retrieved values with the left-hand side of the interpreted assignment statement. In sum, the code for this case is a direct translation of the mathematical into Racket code.

Stop! How would you code up the transition function in your favorite programming language?

Every implementation effort potentially suffers from bugs. In the particular case of a transition function, a programmer can easily make a mistake with the translation of the *before* conditions so that some instance of state does not match any of them. Worse, the specification effort of writing down a transition function in the table notation of figure 8 may suffer from inconsistencies in these conditions, rendering a precise translation into code buggy.

To catch such problems, an implementation must come with a catch-all clause, which is what the dots (...) at the end of figure 9 indicate:

```
[_ (eprintf "stuck state (should never happen)")
   (state (err "stuck state (should never happen)" 's) s)]
```

The _ patterns informs readers of Racket code that all given states are matched. The clause then prints an error message and returns an error state so that runMachine no longer calls transition.

A programming language researcher refers to states that match this pattern as *stuck states*. When running a machine ends up in a stuck state, the message must inform the user that the machine crashed—or, in the terminology of old computer scientists—that it seg-faulted. One major research thrust in the field is the search for proof methods that eliminate the possibility of stuck states.

4 The CSK Abstract Machines: A Second Example

Several different abstract machines can express the same semantics for one and the same syntax. By choosing one over the other, a language creator emphasizes a particular perspective. For example, the CS machine mirrors our hardware interpretation of *Sample*. A language creator who presents this machine may hope to allude to the programmers' sense of machine execution. Or, the language creator may expect that proving a universal property–i.e., one that holds for all programs—is best done with the chosen abstract machine.

On occasion, programming language researchers study a particular language and find that the chosen abstract machine isn't a good choice for their objectives. In this case, they may formulate an alternative abstract machine (or even an interpreter or a denotational-mathematical one). If they do, they are obliged to show that the two semantics define identical functions from Programs to Values; otherwise they don't know whether the results of their investigations hold for the original combination of syntax and abstract machine.

This section presents the CSK machine, an alternative for the *Sample* language. While the CS machine elegantly interprets one assignment statement after another, its definition intimately relies on the simple nature of programs. This tight relationship between simple program syntax and machine instructions makes it difficult—not impossible—to adapt this machine to even the syntax of the *Bare Bones* language.

4.1 The States

K is short for "continuation," the mathematical term for "ASTs still to be interpreted." But, since C is taken, semanticists use the letter "K" instead.

As its name suggests, the CSK machine comes with three registers. The C register still contains ASTs that controls the machine's execution, and the S register continues to represent the association of variables and values. But, the C register of this new machine contains the expression to be evaluated, not the sequence of (mostly) un-interpreted assignment statements. Those are instead placed in the additional K register.

Let's capture these ideas as a struct type definition for the states of the CSK machine, combined with comments on how to use its instances:

```
(struct state [control store kontinuation])
#; {type State  = (U Initial InterM Final)}
#; {type Iniial = [state † MT Program]}
#; {type InterM = [state (U Expression †) Store Program]}
#; {type Final  = [state (U Value Error) Store (Expression)]}
```

search for expression, place value

	Ctrl.	Sto.	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

	Ctrl.	Sto.	Kontinuation
evaluate a variable (success)			
before:	y	s	k
subject to:	y is defined in s		
after:	n	s	k
where	n = s[y]		
evaluate a variable (failure)			
before:	y	s	k
subject to:	y is not defined		
after:	error	[ ]	[ ]
evaluate an addition (success)			
before:	(y + z)	s	k
subject to:	y and z are defined in s		
after:	(+ s[y] s[z])	s	k
evaluate an addition (failure)			
before:	(y + z)	s	k
subject to:	y or z is not defined		
after:	error	[ ]	[ ]

Figure 10: The CSK transition function for *Sample*

The set of values remains the set of inexact numbers; Error is also the same as for the CS machine.

The novelty in these definitions is the † symbol, and we definitely need an interpretation for this symbol to make sense of these states. Since C contains the next expression to be evaluated, and since it isn't always obvious which expression it is, the CSK machine instead knows two modes:

- When † is in C, the CSK machine searches for the next expression that it must evaluate in the AST of the K register.
- When C contains an expression, it evaluates the expression, and when the value is found, it uses the program AST in K to place the value into the store.

Stop! Define the load and unload function for these data definitions.

4.2 The Transition Function

Figure 10 displays the definition of the CSK transition in the same format as the one for the CS machine. The cases in the left column explain how the CSK machine searches for the next expression to evaluate. The right column defines the four cases of how the machine determines the value of an expressions—if possible—and how it uses this value.

Consider the second case in the left column. Its *before* constraint on the C register signals "search." The K register is supposed to contain a program that consists of a non-empty sequence of assignment statements followed by an expression e. Since the assignment statement's right-hand side, ex, is an expression whose value is needed, the next expression is found. Hence the case's *after* specification places ex into the C register and leaves the other two registers alone.

Once the C register contains an expression, the evaluation starts. Consider the third case in the right-hand column. Its *before* condition demands that the C register contains an addition expression; it imposes no constraints on the content of S and K. The side condition, however, ties together the expression with the store. It requires that the store associates values with both variables in the addition expression. Assuming that the two conditions hold, the *after* specification constructs a state with the sum of the two numbers in C, with the other two registers unchanged.

To understand how an assignment statement is executed, we need to inspect the third case in the left column. Its *before* line says that C contains a number (n) and a program in K with a non-empty sequence of assignments. The number is the result of evaluating the right-hand side of this assignment, and that immediately explains the *after* specification. The number n becomes the new value of the left-hand side of the assignment statement, x, and the assignment itself disappears.

Stop! Explain the remaining cases in a similar manner.

Exercise 2. Add cases for if0 and while0 (see figure 6) to the transition function of figure 10. The implied semantics are as follows:

- An if0 statement determines the value of its sub-expression. If this value is 0.0, the "then" branch is executed next; otherwise the machine picks the "else" branch.
- A while0 proceeds in a similar manner. As long as the evaluation of its sub-expression yields 0.0, the machine executes the sub-statement. If

the sub-expression evaluates to some other number, the while0 statement is removed from the K register.

Don't peek ahead. The solution is presented in the next section.

Exercise 3. Your task is to implement the CSK abstract machine in your favorite programming language.

Define csk0, a program that consumes an S-expression from standard input, parses it according to the BNF of section 2, determines its meaning via the CSK machine, and prints the meaning to standard output. Keep in place the restriction we have imposed here, namely, that addition expressions must use only two variables as operators.

The output should be a number only when the parser succeeds and the machine runs to completion without discovering variables without an association in the store. If parsing the given S-expression fails, the error message should say "parser error." If running the machine fails, the program should signal a "run-time error."

Challenge Can the csk0 run forever?

Hint You may wish to re-read section 4.

4.3 But Machines Don't Search For the Next Instruction!

At this point, you might think that these CSK "machines" are anything but machines. You know that *real* computer hardware does *not* search for the next instruction to execute. Real hardware comes with an program counter (aka instruction counter aka instruction pointer aka many other names) that points to the next instruction to be executed. As one instruction gets done, the pointer is advanced appropriately.

While the CS machine acts in a way that resembles hardware, recognizing the relationship between the CSK machine and a computer is much more difficult. Here is an attempt. First, its K register does contain a sequence of instructions. Second, S still points to a store that is like a hardware memory. Finally, the CSK machine's C register mostly drives arithmetic instructions like an ALU on a CPU, meaning they compute sums; in other cases, it demands the retrieval and storing of numbers in memory.

A CSK machine differs from hardware in that it carefully explains how the program counter is moved forward. One way to imagine is to think of this as a "slow motion" execution of instructions.

We have two reasons to formulate abstract machines. First, it is straightforward to implement these machines in almost any programming language. While differentiating the cases of the transition function may take

more code in one language than another, it remains an achievable task. Second, the transition function doesn't make big leaps (like, say, an interpreter) but performs small steps, one small instruction at a time. Sure, they aren't quite hardware steps, but you can simulate each of them in your head. And as you will see, these abstract machines thus provide an adaptable formalism for basically any kind of language feature.

4.4 A Little Bit of Theory

The addition of the CSK machine to our repertoire raises the kind of question that theoreticians in the field of programming languages often have to consider:

do the CS machine and the CSK machine define the same semantics?

And you may wonder what it even means for two semantics to be the same.

So suppose you write a program P in the restricted BNF. You use your parser to make sure it is well-formed. If so, you get the AST of P and you proceed to apply runMachine_{CS} function to this tree. When the final is unloaded, the result is either a number or an error. What if we proceeded in this manner but used the runMachine_{CSK} function instead?

Stop! Did you notice the subscripts? Do you know what they mean?

What you should expect is that the second process should produce the same outcome as the first one. Indeed, you should get the same outcome no matter which P you start from.

Since we can understand the runMachine functions as *mathematical* functions, we can turn this informal claim into a mathematical theorem about them as follows:

$$\text{For all programs } P \text{ in } \mathit{Sample}, \mathit{runMachine}_{CS}(P) = \mathit{runMachine}_{CSK}(P)$$

A language theoretician would use mathematical methods (relations, induction) to prove this statement formally.

In this book, we are not concerned with such theoretical arguments. Our focus is to explain the meaning of language features via abstract machines and their pragmatics through rigorous analyses.

5 Project Machine: *Bare Bones*

Let's equip the *Bare Bones* BNF of section 7 with a semantics. Programs in this BNF consist of sequences of Statements, followed by an Expression. This

production is exactly the same as for *Sample*'s BNF—if we ignore what Statements and Expressions are.

According to the second production of the *Bare Bones* BNF, a Statement is either an assignment statement, a conditional, or a loop. In other words, the *Bare Bones* syntax adds two kinds of Statements to *Sample*.

Finally, a *Bare Bones* expression is either a literal numeric constant, a reference to a variable, or an addition expression that contains two variable references. While the *Sample* BNF comes with just three literal numeric constants (1, 2, and 3), the good news is that we know what these constants mean—the numbers that they denote.

In sum, *Bare Bones* syntax extends the syntax of the *Sample* BNF in one essential place: the production for Statement. Hence the abstract machine for this extended syntax should merely extend the set of transitions of the CSK machine for *Sample*. Concretely, the extended machine should need transitions for using the value of expressions in if0 and while0 plus search transitions for finding expressions in such statements.

search for expression, place value

	Ctrl.	Sto.	Kontinuation
pick then branch from if0			
before:	n	s	((if0 tst thn els)::stmt* e)
subject to:	n is 0		
after:	†	s	(thn::stmt* e)
pick else branch from if0			
before:	n	s	((if0 tst thn els)::stmt* e)
subject to:	n is not 0		
after:	†	s	(els::stmt* e)
search for expression in if			
before:	†	s	((if0 tst thn els)::stmt* e)
after:	tst	s	((if0 tst thn els)::stmt* e)

evaluating expressions

	Ctrl.	Sto.	Kontinuation
decide whether to run while loop (positive)			
before:	n	s	((while0 tst body)::stmt* e)
subject to:	n is 0		
after:	†	s	(body::o e)
where	o =	(while0 tst body)::stmt*	
decide whether to run while loop (negative)			
before:	n	s	((while0 tst body)::stmt* e)
subject to:	n is not 0		
after:	†	s	(stmt* e)
search for expression in while			
before:	†	s	((while0 tst body)::stmt* e)
after:	tst	s	((while0 tst body)::stmt* e)

Figure 11: The CSK transition function for *Bare Bones*

Figure 11 adds six cases to the transition function of the CSK machine from figure 10. All other functions of the CSK machine can be re-used as

is—as long as the transition function is revised.

The left column of the figure presents the cases that deal with if0 statements. First, when the expression evaluates to 0.0, the machine picks the thn sub-statement to execute next. Since it doesn't know which expression is to be evaluated next, it initiates a search. Second, when the expression evaluates to a number other than 0.0, the machine picks the els sub-statement and searches for an expression in it. Note how the remainder of the instructions (stmt*) remain the same.

The right column of figure 11 is about the cases that process while0 statements. First, when the expression evaluates to 0.0, the machine executes the body sub-statement of the loop followed by the loop. Second, otherwise the machine discards the loop statement. The machine initiates a search for next expression to be evaluated in both cases.

A second look at the last cases in both columns confirms how searching for an expression terminates when the machine encounters a if0 or while0 statement as the first one in K. In both cases, the machine picks the sub-expression of the statements, because its value is needed to make a decision about what to execute next.

Exercise 4. Your task is to implement the complete CSK abstract machine for *Bare Bones* in your favorite programming language.

If you completed exercise 3, this task represents just a request to add six cases to the already-existing transition function. Also, implement a csk1 programming; use a suitably modified specification like the one for csk0 in exercise 3.