Chapter 8
TRADITIONAL OPERATIONAL SEMANTICS

In contrast to a semantics that describes only what a program does, the purpose of operational semantics is to describe how a computation is performed. An introduction to computers and programming languages is usually presented in terms of operational concepts. For example, an assignment statement "V := E" might be described by the steps that it performs: Evaluate the expression E and then change the value bound to the variable V to be this result. In another example, pointer assignments might be made clearer by drawing boxes and arrows of the configurations before and after the assignments.

Informal operational semantics illustrates the basic components of the operational approach to meaning. A state or configuration of the machine is described by means of some representation of symbols, such as labeled boxes, values in the boxes, and arrows between them. One configuration assumes the role of the initial state, and a function, determined by the program whose meaning is being explained, maps one configuration into another. When the program (or programmer) is exhausted or the transition function is undefined for some reason, the process halts producing a "final" configuration that we take to be the result of the program.

In this chapter we first discuss how earlier chapters have already presented the operational semantics of languages. We then briefly describe a well-known but seldom-used method of specifying a programming language by means of formal operational semantics—namely, the Vienna Definition Language.

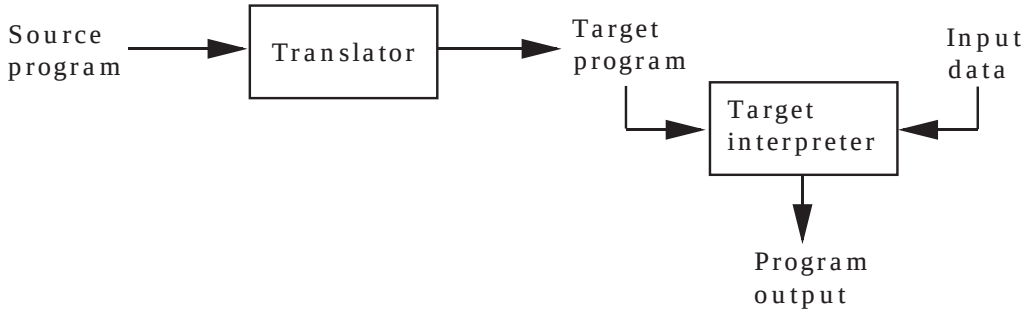
The main part of this chapter looks at the SECD abstract machine defined by Peter Landin, an early method of describing expression evaluation in the context of the lambda calculus. The chapter concludes with an introduction to a formal specification method known as structural operational semantics. This method describes semantics by means of a logical system of deductive rules that model the operational behavior of language constructs in an abstract manner.

The laboratory exercises for this chapter include implementing the SECD machine for evaluating expressions in the lambda calculus following the pattern used in Chapter 5 and implementing a prototype interpreter of Wren based on its structural operational semantics using the scanner and parser developed in Chapter 2.

8.1 CONCEPTS AND EXAMPLES

We have already considered several versions of operational semantics in this text. For the lambda calculus in Chapter 5, the β-reduction and δ-reduction rules provide a definition of a computation step for reducing a lambda expression to its normal form, if possible. A configuration consists of the current lambda expression still to be reduced, and the transition function simply carries out reductions according to a predetermined strategy. The occurrence of a lambda expression in normal form signals the termination of the computation. As with most operational semantics, the computation may continue forever.

One view of operational semantics is to take the meaning of a programming language as the collection of results obtained by executing programs on a particular interpreter or as the assembly language program produced by a compiler for the language. This approach to meaning is called **concrete operational semantics** . The translational semantics of a compiler, such as the one discussed in Chapter 7 using an attribute grammar, can be understood as a definition of a programming language. The diagram below shows the structure of a translational system for a programming language.

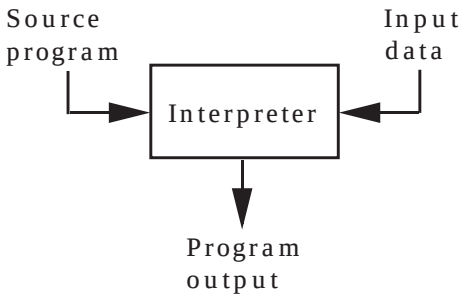


The translational approach entails two major disadvantages as a formal specification tool:

1. The source language is defined only as well as the target language of the translator. The correctness and completeness of the specification relies on a complete understanding of the target language.

2. A translator can carefully describe the relation between a source program and its translation in the target language, but it may at the same time provide little insight into the essential nature of the source language.

More commonly, concrete operational semantics refers to an interpreter approach in which a source language program is simulated directly. An interpretive definition of a programming language is generally less complex than following a translational method. The diagram below shows the basic structure of an interpretation system.



Defining the meaning of a programming language in terms of a real interpreter also has shortcomings as a specification mechanism:

1. For complex languages, correct interpreters (as well as compilers) are difficult to write. Moreover, these definitions are too machine dependent to serve as formal specifications for a programming language.

2. Interpreters are written to provide practical program development tools, and they do not provide the mathematical precision needed in a formal definition.

The metacircular interpreters in Chapter 6 describe the operation of a Lisp machine and a simplified Prolog machine using the languages themselves to express the descriptions. These descriptions represent the configurations directly as structures in the language being defined. John McCarthy's definition of Lisp in Lisp [McCarthy65b] was an early landmark in providing the semantics of a programming language. Although such meta-interpreters can give insight into a programming language to someone who is already familiar with the language, they are not suitable as formal definitions because of the circularity.

Formal semantics demands a better approach, using a precisely defined hypothetical abstract machine described in a formal mathematical or logical language with no limitations on memory, word size, precision of arithmetic, and other such implementation-dependent aspects of a language, and rigor-

ously defined rules that reveal the way the state of the machine is altered when a program is executed.

VDL

The most ambitious attempt at defining an abstract machine for operational semantics was the Vienna Definition Language (VDL) developed at the Vienna IBM laboratory in 1969. All the nonprimitive objects in VDL are modeled as trees. This includes the program being interpreted (an abstract syntax tree), memory, input and output lists, environments, and even the control mechanism that performs the interpretation. Figure 8.1 shows a typical VDL configuration that is represented as a collection of subtrees. A set of instruction definitions, in effect a "microprogram", interprets an abstract representation of a program on the abstract machine defined by this tree structure.

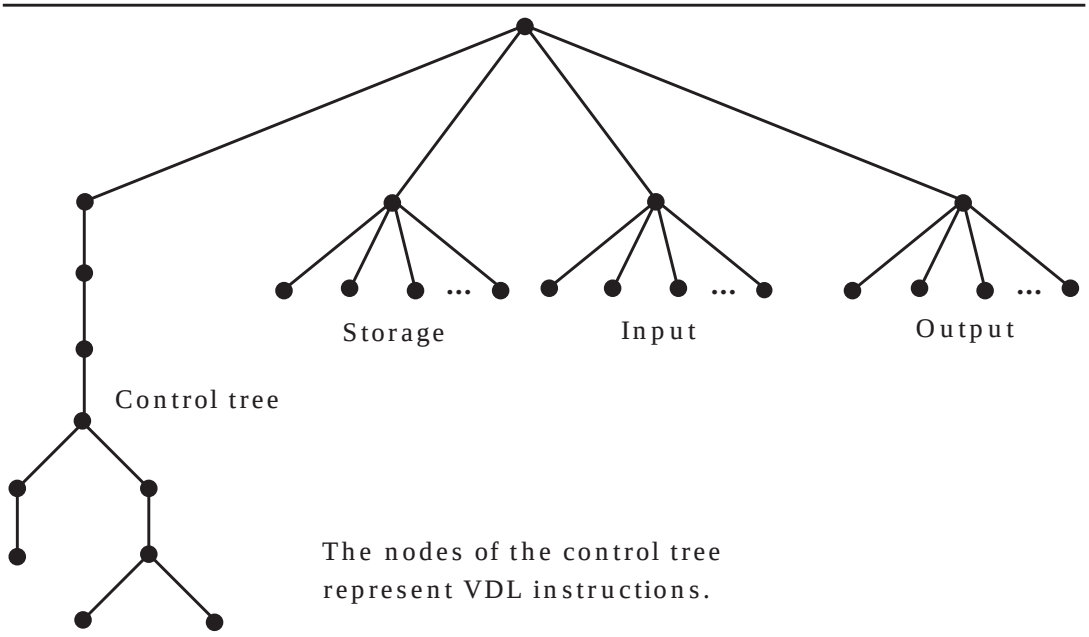


Figure 8.1: A VDL Configuration

Starting with an initial configuration or state that has all the components of storage properly initialized (probably to "undefined"), input defined as a tree representing the list of input values, output set as an empty tree, and the control tree defined as a single instruction to execute the entire program, the transition function given by the instructions of the VDL interpreter performs the steps of a computation. One step consists of selecting a leaf node of the control tree and evaluating it according to the microprogram, producing a

new state with a modified control tree. As the leaf nodes of the control tree are evaluated, a sequence of configurations results:

$$\text{configuration}_0 \rightarrow \text{configuration}_1 \rightarrow \text{configuration}_2 \rightarrow \text{configuration}_3 \rightarrow \ldots.$$

An interpretation of a program terminates normally when the control tree becomes empty, signaling that the program has completed. The VDL interpreter also recognizes certain error conditions that may occur during an execution, and the computation may execute forever.

The major accomplishment of the VDL effort was a specification of PL/I. Unfortunately, the complexity of definitions in VDL hampers its usefulness. The tree structures do not relate to any actual implementation, and the details of the representation can overwhelm the users of a VDL specification to the point of raising questions about its correctness. Any hope of practical application of formal semantics depends on providing a certain amount of clarity and conciseness in language definitions.

Our venture into traditional operational semantics considers two examples that are more accessible. The first technique illustrates the use of an abstract machine, called the SECD machine, to interpret the lambda calculus. Developed in the mid 1960s, it provides an elegant example of traditional operation semantics that has become the basis for some implementations of functional programming languages. The second method of language specification, called structural operational semantics, finds its roots in logic deduction systems. It is a more abstract approach to operational specifications, recently supporting applications in type theory.

Exercises

1. List several ways that programming languages are described to beginners using informal operational semantics.

2. How well does the translational semantics of Chapter 7 provide a formal definition of the programming language Wren? Can it be used to prove the equivalence of language constructs?

3. Enumerate some of the advantages and disadvantages of using an actual interpreter or compiler as the definition of Pascal.

8.2 SECD: AN ABSTRACT MACHINE

In 1964 Peter Landin proposed an abstract machine, called the **SECD machine**, for the mechanical evaluation of lambda expressions. This machine has become a classic example of operational semantics, involving computational techniques that have been adopted in practical implementations of functional programming languages. With the SECD machine, evaluation of a function application entails maintaining an environment that records the bindings of formal parameters to arguments in a way similar to the method of implementing function application in some real implementations. The SECD machine surpasses the efficiency of a lambda calculus evaluator based on β-reductions, but it lends itself primarily to an applicative order evaluation strategy. In fact, the SECD machine as described by Landin follows pass by value semantics in the sense that combinations in the body of a lambda expression are not reduced unless the lambda abstraction is applied to an argument; so the evaluator stops short of normal form in some instances.

The states in the abstract machine consist of four components, all exhibiting stack behavior. The names of these four stacks, S, E, C, and D, provide the title of the machine.

S for **Stack**: A structure for storing partial results awaiting subsequent use.

E for **Environment**: A collection of bindings of values (actual parameters) to variables (formal parameters).

C for **Control**: A stack of lambda expressions yet to be evaluated plus a special symbol "@" meaning that an application can be performed; the top expression on the stack is the next one to be evaluated.

D for **Dump**: A stack of complete states corresponding to evaluations in progress but suspended while other expressions (inner redexes) are evaluated.

In describing the SECD interpreter we represent a state, also called a configuration, as a structured object with four components:

cfg(S, E, C, D).

Borrowing notation from Prolog and mixing it with a few functional operations, we depict the S and C stacks as lists of the form [a,b,c,d] with the top at the left and define *head* and *tail* so that

head([a,b,c,d]) = a, and

tail([a,b,c,d]) = [b,c,d].

To push an item X onto the stack S, we simply write [X | S] for the new stack. The empty list [ ] acts as an empty stack.

Environments, which provide bindings for variables, are portrayed as lists of pairs, say [x|→3, y|→8], with the intention that bindings have precedence from left to right. An empty environment is denoted by the atom "nil". In describing the SECD machine, we let E(x) denote the value bound to x in E, and let [y|→val]E be the environment E_1 that extends E with the property

$$E_1(x) = E(x) \text{ if } x \neq y, \text{ and}$$

$$E_1(y) = val.$$

So if E = [y|→5][x|→3, y|→8] = [y|→5, x|→3], E(y) = 5. If an identifier x has not been bound in E, the application E(x) returns the variable x itself.

The D stack is represented as a structure. Since a dump is a stack of configurations (states), we display it using notation with the pattern

$$cfg(S_1,E_1,C_1,cfg(S_2,E_2,C_2,cfg(S_3,E_3,C_3,nil)))$$

for a dump that stacks three states. An empty dump is also given by "nil".

When a lambda abstraction (λV . B) appears on the top of the control stack, it is moved to the partial result stack while its argument is evaluated. The object that is placed on the stack is a package containing the bound variable V and body B of the abstraction, together with the current environment Env, so that the meaning of the free variables can be resolved when the abstraction is applied. This bundle of three items is known as a **closure** since the term represented is a closed expression in the sense that it carries along the meanings of its free variables. We represent such a closure by the structure "closure(V,B,Env)", which we abbreviate as "cl(V,B,Env)" when space is short.

To evaluate a lambda expression expr, the SECD machine starts with the initial configuration cfg([ ],nil,[expr],nil) that has empty stacks for S, E, and D. The one item on the control stack is the expression to be evaluated. The SECD machine is defined by a transition function,

$$transform : State \rightarrow State,$$

that maps the current configuration to the next configuration until a final state results, if it ever does. A final state is recognized by its having an empty control stack and an empty dump, indicating that no further computation is possible. Figure 8.2 gives a definition of the *transform* function as a conditional expression returning a new configuration when given the current configuration.

transform cfg(S, E, C, D) =

(1) if *head*(C) is a constant
then cfg([*head*(C) | S], E, *tail*(C), D)

(2) else if *head*(C) is a variable
then cfg([E(*head*(C)) | S], E, *tail*(C), D)

(3) else if *head*(C) is an application (Rator Rand)
then cfg(S, E, [Rator,Rand,@ | *tail*(C)], D)

(4) else if *head*(C) is a lambda abstraction λV . B
then cfg([closure(V,B,E) | S], E, *tail*(C), D)

(5) else if *head*(C) = @ and *head*(*tail*(S)) is a predefined function f
then cfg([f(*head*(S)) | *tail*(*tail*(S))], E, *tail*(C), D)

(6) else if *head*(C) = @ and *head*(*tail*(S)) = closure(V,B,E_1)
then cfg([ ], [V|→*head*(S)]E_1, [B], cfg(*tail*(*tail*(S)),E,*tail*(C),D))

(7) else if C = [ ]
then cfg([*head*(S) | S_1], E_1, C_1, D_1) where D = cfg(S_1,E_1,C_1,D_1)

Figure 8.2: Transition Function for the SECD Machine

In order to explain the SECD machine, we will discuss each case in the definition of the transition function. The cases are determined primarily by the top element of the control stack C.

1. If the next expression to be evaluated is a constant, move it as is from the control stack to the partial result stack S.

2. If the next expression is a variable, push its binding in the current environment onto S. If no binding exists, push the variable itself.

3. If the next expression is an application (Rator Rand), decompose it and reenter the parts onto the control stack C with the Rator at the top, the Rand next, and the special application symbol @ following the Rand. (In his original machine, Landin placed the Rand above the Rator to be evaluated first, but that results in rightmost-innermost evaluation instead of leftmost-innermost—redexes in the Rator before the Rand—that we described for applicative order evaluation in Chapter 5.)

4. If the next expression is a lambda abstraction, form a closure incorporating the current environment and add that closure to the partial result stack. The use of a closure ensures that when the lambda abstraction is applied, its free variables are resolved in the environment of its definition, thereby providing static scoping.

5. If the next expression is @ and the function in the second place on the S stack is a predefined function, apply that function to the evaluated argument at the top of the S stack and replace the two of them with the result.

6. If the next expression is @ and the function in the second place of the S stack is a closure, after popping @ and the top two elements of S, push the current configuration onto the dump. Then initiate a new computation to evaluate the body of the closure in the closure's environment augmented with the binding of the bound variable in the closure, the formal parameter, to the argument at the top of the partial result stack.

7. If the control stack is empty, that means the current evaluation is completed and its result is on the top of the partial result stack. Pop the configuration on the top of the dump, making it the new current state with the result of the previous computation appended to the top of its partial result stack.

If the control stack and the dump are both empty, the transition function is undefined, and the SECD machine halts in a final state. The value at the top of the partial result stack is the outcome of the original evaluation.

Example

Shown below are the state transitions as the lambda expression

((λx . (mul x ((λy . sqr y) 5))) 3)

is evaluated by the SECD machine. The numbers at the far right identify which alternative of the definition by cases is employed at each step. Closures are represented as cl(V,B,E), and g stands for (λy.sqr y) to save space in describing the computation.

S	**E**	**C**	**D**	
[ ]	nil	[((λx.(mul x ((λy.sqr y) 5))) 3)]	nil	
[ ]	nil	[(λx.(mul x (g 5))), 3, @]	nil	(3)
[cl(x,(mul x (g 5)),nil)]	nil	[3, @]	nil	(4)
[3, cl(x,(mul x (g 5)),nil)]	nil	[@]	nil	(1)
[ ]	[x↦3]	[(mul x (g 5))]	d_1	(6)
where d_1 = cfg([ ],nil,[ ],nil)				
[ ]	[x↦3]	[(mul x), (g 5), @]	d_1	(3)

[ ]	[x↦3]	[mul, x, @, (g 5), @]	d_1	(3)
[mul]	[x↦3]	[x, @, (g 5), @]	d_1	(1)
[3, mul]	[x↦3]	[@, ((λy.sqr y) 5), @]	d_1	(2)
[mul3]	[x↦3]	[((λy.sqr y) 5), @]	d_1	(5)
where mul3 is the unary function that multiplies its argument by 3				
[mul3]	[x↦3]	[(λy.sqr y), 5, @, @]	d_1	(3)
[cl(y,(sqr y),E_1), mul3]	[x↦3]	[5, @, @]	d_1	(4)
where E_1 = [x↦3]				
[5, cl(y,(sqr y),E_1), mul3]	[x↦3]	[@, @]	d_1	(1)
[ ]	[y↦5, x↦3]	[(sqr y)]	d_2	(6)
where d_2 = cfg([mul3],[x↦3],[@],d_1)				
[ ]	[y↦5, x↦3]	[sqr, y, @]	d_2	(3)
[sqr]	[y↦5, x↦3]	[y, @]	d_2	(1)
[5, sqr]	[y↦5, x↦3]	[@]	d_2	(2)
[25]	[y↦5, x↦3]	[ ]	d_2	(5)
[25, mul3]	[x↦3]	[@]	d_1	(7)
[75]	[x↦3]	[ ]	d_1	(5)
[75]	nil	[ ]	nil	(7)

The transition function has no definition when both the control stack and the dump are empty. The result of the evaluation is the value 75 at the top of the S stack.

Parameter Passing

As mentioned earlier, the SECD machine evaluates redexes following a leftmost-innermost strategy but fails to continue reducing in the body of a lambda expression. The next example illustrates this pass by value approach to the lambda calculus by evaluating the lambda expression ((λf . λx . f x)(λy . y)).

S	E	C	D	
[ ]	nil	[((λf.λx.f x)(λy.y))]	nil	
[ ]	nil	[(λf.λx.f x), (λy.y), @]	nil	(3)
[cl(f,(λx.f x),nil)]	nil	[(λy.y), @]	nil	(4)
[cl(y,y,nil), cl(f,(λx.f x),nil)]	nil	[@]	nil	(4)
[ ]	[f↦cl(y,y,nil)]	[(λx.f x)]	d_1	(6)
where d_1 = cfg([ ],nil,[ ],nil)				
[cl(x,(f x),[f↦cl(y,y,nil)])]	[f↦cl(y,y,nil)]	[ ]	d_1	(4)
[cl(x,(f x),[f↦cl(y,y,nil)])]	nil	[ ]	nil	(7)

This final state produces the closure cl(x,(f x),[f↦cl(y,y,nil)]) as the result of the computation. Unfolding the environment that binds f to cl(y,y,nil) in the closure and extracting the lambda abstraction (λx . f x) from the closure, we get (λx . ((λy . y) x)) as the final result following a pass by value reduction. In true applicative order evaluation, the reduction continues by simplifying the body of the abstraction giving the lambda expression (λx . x), which is in normal form, but pass by value reduction does not reduce the bodies of abstractions that are not applied to arguments.

Static Scoping

The easiest way to see that lambda calculus reduction adheres to static scoping is to review **let**-expressions, which we discussed briefly at the end of section 5.2. For example, consider the following expression:

```
let x=5
    in let f = λy . (add x y)
           in let x = 3
                  in f x
```

Here the variable x is bound to two different values at different points in the expression. When the function f is applied to 3, to which value is 3 added? By translating the **let**-expression into a lambda expression

(λx . (λf . ((λx . f x) 3)) (λy . (add x y))) 5

and reducing, following either a normal order or an applicative order strategy, we get the value 8 as the normal form. To match this behavior in the SECD machine, the function f must carry along the binding of x to 5 that is

in effect when f is bound to (λy . (add x y)). That is precisely the role of closures in the interpretation.

In contrast, a slight change to the definition of the transition function turns the SECD machine into an adherent to dynamic scoping. Simply change case 6 to read:

else if *head*(C) = @ and *head*(*tail*(S)) = closure(V,B,E_1)
then cfg([ ], [V|→*head*(S)]E, [B], cfg(*tail*(*tail*(S)),E,*tail*(C),D))

Now the body of the lambda abstraction is evaluated in the environment in effect at the application of f to x—namely, binding x to 3, so that the computation produces the value 6. With dynamic scoping, closures can be dispensed with altogether, and case 4 need only move the lambda abstraction from the top of the control stack to the top of the partial result stack S.

Exercises

1. Trace the execution of the SECD machine on the following lambda expressions in the applied lambda calculus that we have defined:
   a) (succ 4)
   b) (λx . (add x 2)) 5
   c) (λf . λx . (f (f x))) sqr 2
   d) (λx . ((λy . λz . z y) x)) p (λx . x)

2. In the pure lambda calculus, both the successor function and the numeral 0 are defined as lambda expressions (see section 5.2). The expression (succ 0) takes the form

   (λn . λf . λx . f (n f x)) (λf . λx . x)

   in the pure lambda calculus. Use the SECD machine and β-reduction to evaluate this expression. Explain the discrepancy.

3. Trace the execution of the SECD machine on the lambda expression that corresponds to the **let**-expresssion discussed in the text:

   (λx . (λf . ((λx . f x) 3)) (λy . (add x y))) 5

4. Trace a few steps of the SECD machine when evaluating the lambda expression (λx . x x) (λx . x x).

5. Modify the SECD machine so that it follows a normal order reduction strategy. Use a new data structure, a suspension, to model unevaluated expressions. See [Field88] or [Glaser84] for help on this exercise.

8.3 LABORATORY: IMPLEMENTING THE SECD MACHINE

If we use the scanner and parser from the lambda calculus evaluator in Chapter 5, the implementation of the SECD machine is a simple task. We already use Prolog structures for some of the components of a configuration. All we need to add is a Prolog data structure for environments. We design the SECD interpreter to be used in the same way as the evaluator, prompting for the name of a file that contains one lambda expression. The transcript below shows the SECD machine as it evaluates the lambda expression from the example in section 8.2.

```
>>> SECD: Interpreting an Applied Lambda Calculus <<<
Enter name of source file: text
    ((L x (mul x ((L y (sqr y)) 5))) 3)
Successful Scan
Successful Parse
Result = 75
yes
```

We represent the stack S and control C using Prolog lists, as in the definition of the transition function in the previous section, but now we implement *head* and *tail* by pattern matching.

Environments are implemented as structures of the form env(x, 3, env(y, 8, nil)) for the environment $[x \mapsto 3, y \mapsto 8]$. An empty environment is given by nil. A predicate extendEnv(Env,X,Val,NewEnv) appends a new binding to an existing environment, producing a new environment. A single Prolog clause defines this predicate:

```
extendEnv(Env,Ide,Val,env(Ide,Val,Env)).
```

For example, if Env is bound to the structure env(x, 3, env(y, 8, nil)) and we execute extendEnv(Env,z,13,NewEnv), NewEnv will be bound to the structure env(z, 13, env(x, 3, env(y, 8, nil))).

The predicate applyEnv(Env,Ide,Val) performs the application of an environment to a variable to find its binding. Three clauses define this predicate, the first clause tries to match the binding at the top of the environment:

```
applyEnv(env(Ide,Val,Env),Ide,Val).
```

If the first clause fails, the second continues the search in the "tail" of the environment:

```
applyEnv(env(Ide1,Val1,Env),Ide,Val) :- applyEnv(Env,Ide,Val).
```

Finally, the third clause applies to an empty environment signaling a failed search for the variable. In this case the value returned is the variable itself marked with a tag:

```
applyEnv(nil,Ide,var(Ide)).
```

In agreement with the conventions of Chapter 5, variables and constants have tags (var and con) provided by the parser to make the pattern matching easier to understand. Dumps are implemented as Prolog structures following the pattern used in the previous section:

```
cfg(S1,E1,C1,cfg(S2,E2,C2,cfg(S3,E3,C3,nil))).
```

The transition function for the SECD machine is embodied in a Prolog predicate transform(Config, NewConfig) that carries out one step of the interpreter each time it is invoked. Seven clauses implement the seven cases in Figure 8.2.

```
transform(cfg(S,E,[con(C)|T],D), cfg([con(C)|S],E,T,D)).                    % 1

transform(cfg(S,E,[var(X)|T],D), cfg([Val|S],E,T,D)) :- applyEnv(E,X,Val).  % 2

transform(cfg(S,E,[comb(Rator,Rand)|T],D), cfg(S,E,[Rator,Rand,@|T],D)).    % 3

transform(cfg(S,E,[lamb(X,B)|T],D), cfg([closure(X,B,E)|S],E,T,D)).         % 4

transform(cfg([con(Rand),con(Rator)|T],E,[@|T1],D), cfg([Val|T],E,T1,D)) :-  % 5
                                                    compute(Rator,Rand,Val).

transform(cfg([Rand,closure(V,B,E1)|T],E,[@|T1],D),                         % 6
                                  cfg([ ],E2,[B],cfg(T,E,T1,D))) :-
                                                    extendEnv(E1,V,Rand,E2).

transform(cfg([H|S],E,[ ],cfg(S1,E1,C1,D1)), cfg([H|S1],E1,C1,D1)).         % 7
```

Recall the abstract syntax and the associated tags for the lambda calculus. Each lambda expression is a variable (var), a constant (con), a lambda abstraction (lamb), or a combination (comb). The compute predicate used in step 5 is identical to the one in the evaluator in Chapter 5. Notice how well pattern matching performs tests such as the one in step 6

if *head*(C) = @ and *head*(*tail*(S)) = closure(V,B,E_1),

becomes

```
transform(cfg([Rand,closure(V,B,E1)|T],E,[@|T1],D), ... ).
```

The SECD interpreter is driven by a predicate interpret(Config,Result) that watches for a final state to terminate the machine:

```
interpret(cfg([Result|S],Env,[ ],nil), Result).
```

Otherwise it performs one transition step and calls itself with the new configuration:

```
interpret(Config,Result) :- transform(Config,NewConfig),
                            interpret(NewConfig,Result).
```

If the parser produces a structure of the form expr(Exp), the SECD machine can be invoked using the query:

```
interpret(cfg([ ],nil,[Exp],nil), Result), nl, write('Result = '), pp(Result), nl.
```

where cfg([ ],nil,[Exp],nil), Result) serves as the initial configuration and the predicate pp prints the result (see Chapter 5).

Exercises

1. Following the directions above, implement the SECD machine in Prolog and test it on some of the lambda expressions in the exercises for section 8.2.

2. Change the Prolog implementation of the SECD machine so that it follows the semantics of dynamic scoping instead of static scoping. Illustrate the difference between static and dynamic scoping by evaluating the lambda expression that corresponds to the following let expression:

   let a = 7 **in let** g = λx . (mul a x) **in let** a = 2 **in** (g 10)

3. Add a conditional expression (if E_1 E_2 E_3) to the interpreter. Since the SECD machine follows an applicative order evaluation strategy, "if" cannot be handled by compute, which expects its arguments to be evaluated already. A new case can be added to the definition of transform that manipulates the top few items on the control stack. Test the machine on the following lambda expressions:

   ((L x (if (zerop x) 5 (div 100 x))) 0)

   ((L x (if (zerop x) 2 ((L x (x x)) (L x (x x))))) 0)

4. Extend the lambda calculus to include a "label" expression of the form

   <expression> ::= ... | (label <variable> <expression>)

   whose semantics requires that the expression be bound to the variable in the environment before the expression is evaluated. This mechanism allows recursive functions to be defined (the original approach in Lisp). This definition of the factorial function exemplifies the use of a label expression:

   ((label f (L n (if (zerop n) 1 (mul n (f (pred n)))))) 8)

5. Compare the efficiency of the lambda calculus evaluator in Chapter 5 with the SECD machine in this chapter. Use combinations of "Twice" and "Thrice" as test expressions, where

   Twice = λf . λx . f (f x) and
   Thrice = λf . λx . f (f (f x)).

   For example, try Twice (λz . (add z 1))
   Thrice (λz . (add z 1))
   Twice Thrice (λz . (add z 1))
   Thrice Twice (λz . (add z 1))
   Twice Twice Thrice (λz . (add z 1))
   Twice Thrice Twice (λz . (add z 1)) and so on.

8.4 STRUCTURAL OPERATIONAL SEMANTICS: INTRODUCTION

Proving properties of programs and the constructs of programming languages provides one of the main justifications of formal descriptions of languages. Operational semantics specifies programming languages in terms of program execution on abstract machines. **Structural operational semantics** , developed by Gordon Plotkin in 1981, represents computation by means of deductive systems that turn the abstract machine into a system of logical inferences. Since the semantic descriptions are based on deductive logic, proofs of program properties are derived directly from the definitions of language constructs.

With structural operational semantics, definitions are given by **inference rules** consisting of a conclusion that follows from a set of premises, possibly under control of some condition. The general form of an inference rule has the premises listed above a horizontal line, the conclusion below, and the condition, if present, to the right.

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \ldots \quad \text{premise}_n}{\text{conclusion}} \quad \text{condition}$$

If the number of premises is zero, n=0 in the example, the line is omitted, and we refer to the rule as an **axiom** . This method of presenting rules evolved from a form of logic called **natural deduction** . As an example in natural deduction, three inference rules express the logical properties of the conjunction (and) connective:

$$\frac{p \wedge q}{p} \qquad \frac{p \wedge q}{q} \qquad \frac{p \quad q}{p \wedge q}$$

The principle that allows the introduction of a universal quantifier exhibits a rule with a condition:

$$\frac{P(a)}{\forall x P(x)}$$ a does not occur in P(x) or in any assumption on which P(a) depends.

For more on natural deduction, see the further readings at the end of this chapter.

Rather than investigate this method of expressing logical deductions, we concentrate on the use of inference rules of this form in structural operational semantics. But before we consider the semantics of Wren, we see how an inference system can be used to describe the syntax of Wren.

Specifying Syntax

For the present, we ignore the declarations in a Wren program, assuming that any program whose semantics is to be explained has been verified as syntactically correct (including the context-sensitive syntax), and that all integer identifiers in the program are included in a set Id and Boolean identifiers in a set Bid. Furthermore, we concern ourselves only with abstract syntax, since any program submitted for semantic analysis comes in the form of an abstract syntax tree.

The abstract syntax of these Wren programs is formed from the syntactic sets defined in Figure 8.3. The elements of the sets are identified by the designated metavariables, possibly with subscripts, attached to each syntactic category.

When describing the abstract syntax of a programming language, we strive to fit the description to the structure of the semantic formalism that uses it. The precise notational form of abstract syntax is not intrinsic to a language, as is the concrete syntax. We simply need to give the patterns of the structures that capture the essential components of constructs in the language. For instance, the fundamental property of an assignment is that it consists of an identifier and an expression of the same type.

n ∈ Num = Set of numerals
b ∈ { **true**, **false** } = Set of Boolean values
id ∈ Id = Set of integer identifiers
bid ∈ Bid = Set of Boolean identifiers
iop ∈ Iop = { +, –, *, / }
rop ∈ Rop = { <, ≤, =, ≥, >, <> }
bop ∈ Bop = { **and**, **or** }
ie ∈ Iexp = Set of integer expressions
be ∈ Bexp = Set of Boolean expressions
c ∈ Cmd = Set of commands

Figure 8.3: Syntactic Categories

Figure 8.4 gives a version of the abstract syntax of Wren specially adapted to a structural operational semantic description. In particular, the patterns that abstract syntax trees may take are specified by inference rules and axioms, some with conditions. The statements that make up the premises and conclusions have the form of type assertions; for example,

n : iexp with the condition n∈Num

asserts that objects taken from Num may serve as integer expressions. In this context, "n : iexp" states that n is of type iexp, the sort of objects corresponding to the set Iexp defined in Figure 8.3. The types iexp, bexp, and cmd correspond to the sets Iexp, Bexp, and Cmd, respectively.

The biggest difference between this specification of abstract syntax and that in Chapter 1 is the way we handle lists of commands. The inference rule that permits a command to be a sequence of two commands enables the type "cmd" to include arbitrary finite sequences of commands. Since we ignore declarations in this presentation, a Wren program may be thought of simply as a command. Moreover, a combination of symbols c is a command if we can construct a derivation of the assertion "c : cmd". In fact, the derivation parallels an abstract syntax tree for the group of symbols. Later we give semantics to Wren programs by describing the meaning of a command relative to this specification of abstract syntax.

Assuming that the identifier x is a member of Id because of a declaration that has already been elaborated, a deduction showing the abstract structure of the command "x **:=** 5 ; **while not** (x=0) **do** x **:=** x-1 ; **write** x" is displayed in Figure 8.5. Conditions have been omitted to save space, but each condition should be obvious to the reader. Compare this deduction in the inference system with a derivation according to an abstract syntax given by a BNF-type specification.

n : iexp $\quad n \in Num$

b : bexp $\quad b \in \{\textbf{true}, \textbf{false}\}$

id : iexp $\quad id \in Id$

bid : bexp $\quad bid \in Bid$

$$\frac{ie_1 : iexp \quad ie_2 : iexp}{ie_1\ iop\ ie_2 : iexp}\ iop \in Iop$$

$$\frac{ie_1 : iexp \quad ie_2 : iexp}{ie_1\ rop\ ie_2 : bexp}\ rop \in Rop$$

$$\frac{be_1 : bexp \quad be_2 : bexp}{be_1\ bop\ be_2 : bexp}\ bop \in Bop$$

$$\frac{be : bexp}{\textbf{not}(\ be) : bexp}$$

skip : cmd

$$\frac{ie : iexp}{-\ ie : iexp}$$

$$\frac{ie : iexp}{id\ \textbf{:=}\ ie : cmd}\ id \in Id$$

$$\frac{be : bexp}{bid\ \textbf{:=}\ be : cmd}\ bid \in Bid$$

$$\frac{be : bexp \quad c : cmd}{\textbf{if}\ be\ \textbf{then}\ c : cmd}$$

$$\frac{be : bexp \quad c_1 : cmd \quad c_2 : cmd}{\textbf{if}\ be\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : cmd}$$

$$\frac{c_1 : cmd \quad c_2 : cmd}{c_1\ \textbf{;}\ c_2 : cmd}$$

$$\frac{be : bexp \quad c : cmd}{\textbf{while}\ be\ \textbf{do}\ c : cmd}$$

read id : cmd $\quad id \in Id$

$$\frac{ie : iexp}{\textbf{write}\ ie : cmd}$$

Figure 8.4: Abstract Syntax for Wren

As with most definitions of abstract syntax, this approach allows ambiguity. For the derivation in Figure 8.5, the command sequence can be associated to the left instead of to the right. The choice of grouping command sequences is known as an **inessential ambiguity** since it has no effect on the semantics of the language. Essential ambiguities, such as those in expressions (associativity of minus) and **if** commands (dangling else), are handled by the concrete syntax that is used to construct abstract syntax trees from the program text. Since we represent these tree structures with a linear notation, we insert (meta-)parentheses in abstract syntax expressions whose structure is not obvious.

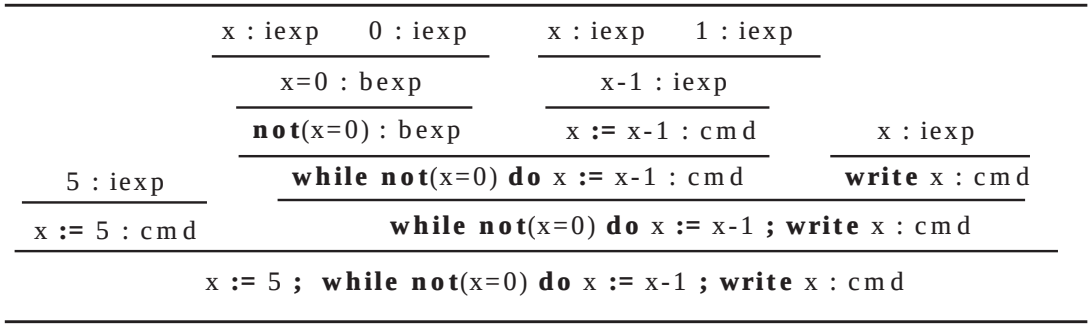


Figure 8.5: Derivation in the Abstract Syntax

Inference Systems and Structural Induction

The abstract syntax for Wren presented in Figure 8.4 can be defined just as well using a BNF specification or even other notational conventions (see [Astesiano91]). The common thread between these presentations of syntax is the inductive nature of the definitions. A set of objects, say Iexp, is specified by describing certain atomic elements—n∈Iexp for each n∈Num and id∈Iexp for each id∈Id—and then describing how more complex objects are constructed from already existing objects,

$$\{ie_1, ie_2 \in Iexp \text{ and } iop \in Iop\} \text{ implies } \{(ie_1 \text{ iop } ie_2) \in Iexp\}.$$

The fundamental structure remains the same whether the set is defined by inference rules or by BNF rules:

iexp ::= n | id | ie_1 iop ie_2 and
iop ::= + | – | * | / where n∈Num, id∈Id, and ie_1,ie_2∈Iexp.

Structured objects described using inductive definitions support a proof method, a version of mathematical induction, known as **structural induction**. This induction technique depends on the property that each object in some collection is either an atomic element with no structure or is created from other objects using well-defined constructor operations.

Principle of Structural Induction : To prove that a property holds for all phrases in some syntactic category, we need to confirm two conditions:

1. **Basis**: The property must be established for each atomic (nondecomposable) syntactic element.
2. **Induction step** : The property must be proved for any composite element given that it holds for each of its immediate subphrases (the induction hypothesis). ∎

For objects defined by a system of inference rules, the axioms create atomic items that are handled by the basis of the induction, and the rules with premises correspond to the induction step. The induction hypothesis assumes that the property being proved holds for all the objects occurring in premises, and we must show that the property holds for the object in the conclusion of the rule.

The syntactic categories defined for the abstract syntax of Wren are so general that few interesting properties can be proven about them. For a simple example, consider the set of all expressions, Exp = Iexp ∪ Bexp. The elementary components of expressions can be divided into two classes:

1. The operands, Rand = Num ∪ {**true**,**false**} ∪ Id ∪ Bid.

2. The operators, Rator = Iop ∪ Rop ∪ Bop ∪ {**not**}.

We can prove a lemma about the number of operands and operators in any Wren expression using structural induction.

Lemma: For any expression e∈(Iexp ∪ Bexp) containing no unary operations (without **not** and unary minus), the number of operands in e is greater than the number of operators in e. Write #rand(e) > #rator(e) to express this relation.

Proof: These expressions are defined by the first seven rules in Figure 8.4.

Basis: Atomic expressions are formed by the four axioms corresponding to numerals, Boolean constants, integer identifiers, and Boolean identifiers. In each case the expression defined has one operand and zero operators, satisfying the property of the lemma.

Induction Step : We consider three cases corresponding to the three inference rules that create structured expressions using a binary operator.

Case 1: e = ie_1 iop ie_2 for some iop∈Iop where ie_1,ie_2 : iexp. By the induction hypothesis, #rator(ie_1) < #rand(ie_1) and #rator(ie_2) < #rand(ie_2). It follows that #rator(ie_2)+1 ≤ #rand(ie_2). But #rator(ie_1 iop ie_2) = #rator(ie_1) + #rator(ie_2) + 1 and #rand(ie_1 iop ie_2) = #rand(ie_1) + #rand(ie_2). Therefore, #rator(ie_1 iop ie_2) = #rator(ie_1) + #rator(ie_2) + 1 < #rand(ie_1) + #rand(ie_2) = #rand(ie_1 iop ie_2).

Case 2: e = ie_1 rop ie_2 for some rop∈Rop where ie_1,ie_2 : iexp. This case is similar to case 1.

Case 3: e = be_1 bop be_2 for some bop∈Bop where be_1,be_2 : bexp. This case is also similar to case 1.

Therefore, by the principle of structural induction, the property #rator(e) < #rand(e) holds for all expressions e∈(Iexp ∪ Bexp) containing no unary operations. ∎

Exercises

1. Construct derivations of these Wren constructs using the abstract syntax inference system in Figure 8.4. Refer to the concrete syntax to resolve ambiguities. Assume that all identifiers have been properly declared.

   a) a*b + c*d

   b) -n-k-5 = n/2*k

   c) n>0 **and not** (switch)

   d) **if** a>=b **then while** a>=c **do write** a ; a := a-1 **else skip**

2. Define (part of) the concrete syntax of Wren using inference rules and axioms in a manner similar to the definition of the abstract syntax of Wren in Figure 8.4.

3. Use the definition of concrete syntax from exercise 2 and structural induction to prove the following properties:

   a) Every expression in Wren has the same number of left and right parentheses.

   b) Each command in Wren has at least as many occurrences of the reserved word **then** as of the reserved word **else**.

4. The following two inference rules define a language comprising lists of integers using "::" as an infix operator denoting the operation of prefixing an element to a list (cons in Lisp):

   $$[\,] : \text{intList} \qquad \frac{\text{tail} : \text{intList}}{\text{m} :: \text{tail} : \text{intList}} \quad \text{m} \in \text{Num}$$

   These are similar to the lists in ML where "::" is a right associative operator and lists can be abbreviated as follows: [1,2,3,4] = 1 :: 2 :: 3 :: 4 :: [ ].

   Functions on these lists of integers can be defined inductively by describing their behavior on the two kinds of lists established by the definitions.

   length([ ]) = 0

   length(m::tail) = 1+length(tail) where m∈Num and tail:intList

   concat([ ],L) = L where L:intList

   concat(m::tail,L) = m :: concat(tail,L) where m∈Num and tail,L:intList

   reverse([ ]) = [ ]

reverse(m::tail) = concat(reverse(tail), m::[ ])
where m∈Num and tail:intList

Use structural induction to prove the following properties concerning the functions just defined on integer lists where the variables L, L_1, L_2, and L_3 range over intList. Some properties depend on earlier ones.

a) concat(L,[ ]) = L

b) length(concat(L_1,L_2)) = length(L_1)+length(L_2)

c) concat(L_1,concat(L_2,L_3)) = concat(concat(L_1,L_2),L_3)

d) length(reverse(L)) = length(L)

e) reverse(concat(L_1,L_2)) = concat(reverse(L_2),reverse(L_1))

f) reverse(reverse(L)) = L

8.5 STRUCTURAL OPERATIONAL SEMANTICS: EXPRESSIONS

We now develop a description of the semantics of Wren using an inference system according to structural operational semantics. The task can be separated into two parts, the first specifying the semantics of expressions in Wren and the second specifying the semantics of commands.

Semantics of Expressions in Wren

Structural operational semantics provides a deductive system, based on the abstract syntax of a programming language, that allows a syntactic transformation of language elements to normal form values that serve as their meaning. Such a definition includes a notion of configurations representing the progress of computations and an inference system that defines transitions between the configurations. We concentrate first on the meaning (evaluation) of expressions in Wren.

Since expressions in Wren permit identifiers, their meaning depends on the values of identifiers recorded in a structure, called the **store**, that models the memory of a computer. Any expression (even any program) contains only a finite set of identifiers, which means that the store structure can be viewed as a finite set of pairs binding values to identifiers, sometimes referred to as a **finite function** . For any store sto, let *dom*(sto) denote the (finite) set of identifiers with bindings in sto.

Thinking of a store as a set of pairs, we informally use a notation of the form {x|→3, y|→5, p|→true} to represent a store with three bindings. For this store, *dom*(sto) = {x,y,p}. Let Store with sto as its metavariable stand for the cat-

egory of stores. The actual implementation of stores is immaterial to the specification of a programming language, so we rely on three abstract operations to describe the manipulation of stores:

1. *emptySto* represents a store with no bindings; that is, all identifiers are undefined.

2. *updateSto*(sto,id,n) and *updateSto*(sto,bid,b) represent the store that agrees with sto but contains one new binding, either id$\mapsto$n or bid$\mapsto$b.

3. *applySto*(sto,id) and *applySto*(sto,bid) return the value associated with id or bid; if no binding exists, the operation fails blocking the deduction.

Observe that *applySto*(sto,id) is defined if and only if id$\in$ *dom*(sto), and the corresponding property holds for Boolean identifiers. Expressions in Wren have no way to modify bindings in a store, so the operation *updateSto* is not used in defining their semantics in Wren.

In a manner similar to the store actions, the binary operations allowed in Wren expressions are abstracted into an all-purpose function *compute*(op,arg_1,arg_2) that performs the actual computations. For example, *compute*(+,3,5) returns the numeral 8, *compute*(<,3,5) returns **true**, and *compute*(**and**,**true**,**false**) returns **false**. Since *compute*(/,n,0) is not defined, the evaluation of any expression in which this computation appears must fail. We say that such an evaluation is **stuck**, since no rule can be successfully applied to an expression of the form "n/0". A stuck computation cannot proceed. This concept is different from a nonterminating computation, which proceeds forever.

For evaluating expressions, a configuration consists of a pair containing an expression to examine and a store that provides a context for the computation. A particular evaluation starts with a configuration, and under control of an inference system, allows a reduction of the configuration to a final or terminating configuration that acts as a normal form value for the expression. In Wren, final configurations for expressions have a first value that is a numeral or a Boolean constant: <n,sto> or <b,sto>.

The inference system for Wren expressions, shown in Figure 8.6, provides rules for each syntactic form that is not in normal form. The symbol $\rightarrow$ serves to represent a transition from one configuration to another. Note that some rules—namely, axioms (7), (12), and (13)—have conditions.

(1) $$\frac{\langle ie_1,sto\rangle \rightarrow \langle ie_1',sto\rangle}{\langle ie_1\ iop\ ie_2,sto\rangle \rightarrow \langle ie_1'\ iop\ ie_2,sto\rangle}$$

(2) $$\frac{\langle ie_1,sto\rangle \rightarrow \langle ie_1',sto\rangle}{\langle ie_1\ rop\ ie_2,sto\rangle \rightarrow \langle ie_1'\ rop\ ie_2,sto\rangle}$$

(3) $$\frac{\langle be_1,sto\rangle \rightarrow \langle be_1',sto\rangle}{\langle be_1\ bop\ be_2,sto\rangle \rightarrow \langle be_1'\ bop\ be_2,sto\rangle}$$

(4) $$\frac{\langle ie_2,sto\rangle \rightarrow \langle ie_2',sto\rangle}{\langle n\ iop\ ie_2,sto\rangle \rightarrow \langle n\ iop\ ie_2',sto\rangle}$$

(5) $$\frac{\langle ie_2,sto\rangle \rightarrow \langle ie_2',sto\rangle}{\langle n\ rop\ ie_2,sto\rangle \rightarrow \langle n\ rop\ ie_2',sto\rangle}$$

(6) $$\frac{\langle be_2,sto\rangle \rightarrow \langle be_2',sto\rangle}{\langle b\ bop\ be_2,sto\rangle \rightarrow \langle b\ bop\ be_2',sto\rangle}$$

(7) $\langle n_1\ iop\ n_2,\ sto\rangle \rightarrow \langle \mathit{compute}(iop,n_1,n_2),\ sto\rangle \quad (iop \neq\ /) \text{ or } (n_2 \neq\ 0)$

(8) $\langle n_1\ rop\ n_2,\ sto\rangle \rightarrow \langle \mathit{compute}(rop,n_1,n_2),\ sto\rangle$

(9) $\langle b_1\ bop\ b_2,\ sto\rangle \rightarrow \langle \mathit{compute}(bop,b_1,b_2),\ sto\rangle$

(10) $$\frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle \textbf{not}(be),sto\rangle \rightarrow \langle \textbf{not}(be'),sto\rangle}$$

(11) $\langle \textbf{not}(\textbf{true}),sto\rangle \rightarrow \langle \textbf{false},sto\rangle \qquad \langle \textbf{not}(\textbf{false}),sto\rangle \rightarrow \langle \textbf{true},sto\rangle$

(12) $\langle id,sto\rangle \rightarrow \langle \mathit{applySto}(sto,id),sto\rangle \qquad id \in \mathit{dom}(sto)$

(13) $\langle bid,sto\rangle \rightarrow \langle \mathit{applySto}(sto,bid),sto\rangle \qquad bid \in \mathit{dom}(sto)$

Figure 8.6: Inference System for Expressions

The inference rules enforce a definite strategy for evaluating expressions. Rules (1) through (3) require that the left argument in a binary expression be simplified first. Only when the left argument has been reduced to a constant (n or b) can rules (4) through (6) proceed by evaluating the right argument. Finally, when both arguments are constants, rules (7) through (9) permit the binary operation to be calculated using *compute*. The only other rules handle the unary operation **not** and the atomic expressions that are identifiers. Atomic expressions that are numerals or Boolean constants have already been reduced to normal form. Unary minus has been left as an exercise at the end of this section.

We can view a computation describing the meaning of an expression as a sequence of configurations where each transition is justified using rules (1) through (13):

$$\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \rightarrow \langle e_3,sto\rangle \rightarrow \ldots \rightarrow \langle e_{n-1},sto\rangle \rightarrow \langle e_n,sto\rangle.$$

Then by adding a rule (14), which makes the $\rightarrow$ relation transitive, we can deduce $\langle e_1,sto\rangle \rightarrow \langle e_n,sto\rangle$ by applying the new rule n-1 times.

$$(14)\quad \frac{\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \qquad \langle e_2,sto\rangle \rightarrow \langle e_3,sto\rangle}{\langle e_1,sto\rangle \rightarrow \langle e_3,sto\rangle} \quad \begin{array}{c} e_1,e_2,e_3 \in Iexp \\ \text{or} \\ e_1,e_2,e_3 \in Bexp \end{array}$$

In addition to having $\rightarrow$ be transitive, it makes sense to assume that $\rightarrow$ is also reflexive, so that $\langle e,sto\rangle \rightarrow \langle e,sto\rangle$ for any expression e and store sto. Then when we write $\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle$, we mean that one configuration $\langle e_1,sto\rangle$ can be reduced to another configuration $\langle e_2,sto\rangle$, possibly the same one, by zero or more applications of inference rules from Figure 8.6.

The rules in these inference systems are really **rule schemes**, representing classes of actual rules in which specific identifiers, numerals, Boolean constants, and specific operators replace the metavariables in the inference rules. For example, "$\langle 5 \geq 12, emptySto\rangle \rightarrow \langle$**false**$, emptySto\rangle$" is an instance of rule (8) since *compute*$(\geq, 5, 12)$ = **false**.

Example

Consider an evaluation of the expression "x+y+6" with sto = $\{x \mapsto 17, y \mapsto 25\}$ given as the store. The sequence of computations depends on the structure of the abstract syntax tree that "x+y+6" represents. We distinguish between the two possibilities by inserting parentheses as structuring devices. We carry out the computation for "x+(y+6)" and leave the alternative grouping as an exercise. Observe that in Wren an abstract syntax tree with this form must have come from a text string that originally had parentheses.

We first display the computation sequence for "x+(y+6)" as a linear derivation:

a) <y,sto>	→ <25,sto> since *applySto*(sto,y)=25	(12)
b) <y+6,sto>	→ <25+6,sto>	(1) and a
c) <25+6,sto>	→ <31,sto> since *compute*(+,25,6)=31	(7)
d) <y+6,sto>	→ <31,sto>	(14), b, and c
e) <x,sto>	→ <17,sto> since *applySto*(sto,x)=17	(12)
f) <x+(y+6),sto>	→ <17+(y+6),sto>	(1) and e
g) <17+(y+6),sto>	→ <17+31,sto>	(4) and d
h) <x+(y+6),sto>	→ <17+31,sto>	(14), f, and g
i) <17+31,sto>	→ <48,sto> since *compute*(+,17,31)=48	(7)
j) <x+(y+6),sto>	→ <48,sto>	(14), h, and i

The last configuration is terminal since <48,sto> is in normal form. Note the use of rule (14) that makes → a transitive relation. Using a proof by mathematical induction, we can establish a derived rule that allows any finite sequence of transitions as premises for the rule:

$$\frac{\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \quad \langle e_2,sto\rangle \rightarrow \langle e_3,sto\rangle \quad \ldots \quad \langle e_{n-1},sto\rangle \rightarrow \langle e_n,sto\rangle}{\langle e_1,sto\rangle \rightarrow \langle e_n,sto\rangle}$$

provided that $n \geq 2$ and every e_i comes from Iexp or every e_i comes from Bexp.

Figure 8.7 depicts the derivation tree corresponding to the inferences that evaluate "x+(y+6)". To save space, the store argument is shortened to "s". The last step of the deduction uses the generalization of rule (14) seen immediately above.

$$\dfrac{\dfrac{\langle x,s\rangle \rightarrow \langle 17,s\rangle}{\langle x+(y+6),s\rangle \rightarrow \langle 17+(y+6),s\rangle} \quad \dfrac{\dfrac{\langle y,s\rangle \rightarrow \langle 25,s\rangle}{\langle y+6,s\rangle \rightarrow \langle 25+6,s\rangle} \quad \langle 25+6,s\rangle \rightarrow \langle 31,s\rangle}{\langle y+6,s\rangle \rightarrow \langle 31,s\rangle} \quad \langle 17+31,s\rangle \rightarrow \langle 48,s\rangle}{\langle x+(y+6),s\rangle \rightarrow \langle 48,s\rangle}$$

Figure 8.7: A Derivation Tree

Notice from the previous example that the step-by-step computation semantics prescribes a left-to-right evaluation strategy for expressions. For an ex-

pression such as "(2*x)+(3*y)", with parentheses to show the structure of the abstract syntax tree, the subexpression "2*x" is evaluated before the subexpression "3*y". Some language designers complain that such ordering amounts to over specification of the semantics of expressions. However, slight changes in the inference rules can make expression evaluation nondeterministic in terms of this order. The problem is left to the reader as an exercise.

Outcomes

We say the computation has terminated or halted if no rule applies to the final configuration $<e_n,sto>$. This happens if the configuration is in normal form or if no rule can be applied because of unsatisfied conditions. Since we solely consider syntactically correct expressions, the only conditions whose failure may cause the computation to become stuck result from the dynamic errors of Wren. These conditions are as follows:

1. (iop≠/) or (n≠0) for rule (7)
2. id∈ *dom*(sto) for rule (12)
3. bid∈ *dom*(sto) for rule (13)

Taking the conditions into account, we can establish a completeness result for the inference system that defines the semantics of Wren expressions.

Definition : For any expression e, let *var*(e) be the set of variable identifiers that occur in e. ∎

Completeness Theorem:

1. For any ie∈(Iexp – Num) and sto∈ Store with *var*(ie)⊆*dom*(sto) and no occurrence of the division operator in ie, there is a numeral n∈Num such that <ie,sto> → <n,sto>.
2. For any be∈(Bexp – {**true**,**false**}) and sto∈ Store with *var*(be)⊆*dom*(sto) and no occurrence of the division operator in be, there is a Boolean constant b∈{**true**,**false**} such that <be,sto> → <b,sto>.

Proof: The proof is by structural induction following the abstract syntax of expressions in Wren.

1. Let ie∈(Iexp – Num) and sto∈ Store with *var*(ie)⊆*dom*(sto), and suppose ie has no occurrence of the division operator. According to the definition of abstract syntax presented in Figure 8.4, ie must be of the form id∈Id or $(ie_1$ iop $ie_2)$ where iop∈Iop–{/} and ie_1,ie_2 : iexp also have no occurrence of /.

   Case 1 : ie = id∈Id. Then id∈dom(sto) and <id,sto> → <n,sto> where n = *applySto*(sto,id) using rule (12).

Case 2: ie = ie_1 iop ie_2 where iop∈Iop–{/} and ie_1,ie_2 : iexp, and for i=1,2, var(ie_i)⊆*dom*(sto) and ie_i contains no occurrence of /.

Subcase a: $ie_1 = n_1 \in Num$ and $ie_2 = n_2 \in Num$. Then <ie,sto> = <n_1 iop n_2,sto> → <n,sto> where n = compute(iop,n_1,n_2) by rule (7), whose condition is satisfied since iop≠/.

Subcase b: $ie_1 = n_1 \in Num$ and $ie_2 \in$ (Iexp – Num). By the induction hypothesis, <ie_2,sto> → <n_2,sto> for some $n_2 \in Num$. We then use rule (4) to get <ie,sto> = <n_1 iop ie_2,sto> → <n_1 iop n_2,sto>, to which we can apply subcase a.

Subcase c: $ie_1 \in$ (Iexp – Num) and $ie_2 \in$ Iexp. By the induction hypothesis, <ie_1,sto> → <n_1,sto> for some $n_1 \in Num$. We then use rule (1) to get <ie,sto> = <ie_1 iop ie_2,sto> → <n_1 iop ie_2,sto>, to which we can apply subcase a or subcase b.

Therefore, the conditions for structural induction on integer expressions are satisfied and the theorem holds for all ie∈(Iexp–Num).

2. An exercise. ∎

A companion theorem, called the consistency theorem, asserts that every computation has a unique result.

Consistency Theorem:

1. For any ie∈(Iexp – Num) and sto∈ Store, if <ie,sto> → <n_1,sto> and <ie,sto> → <n_2,sto> with $n_1,n_2 \in Num$, it follows that $n_1 = n_2$.

2. For any be∈(Bexp – {**true**,**false**}) and sto∈ Store, if <be,sto> → <b_1,sto> and <be,sto> → <b_2,sto> with $b_1,b_2 \in$ {**true**,**false**}, it follows that $b_1 = b_2$.

Proof: Use structural induction again.

1. Let ie∈(Iexp – Num) and sto∈ Store with <ie,sto> → <n_1,sto> and <ie,sto> → <n_2,sto> for $n_1,n_2 \in Num$.

   Case 1: ie = id∈Id. Then both computations must use rule (12), and n_1 = *applySto*(sto,id) = n_2.

   Case 2: ie = ie_1 iop ie_2. The last step in the computations <ie_1 iop ie_2,sto> → <n_1,sto> and <ie_1 iop ie_2,sto> → <n_2,sto> must be obtained by applying rule (7) to expressions of the form k_1 iop k_2 and m_1 iop m_2
   where $k_1,k_2,m_1,m_2 \in Num$,
   <ie_1 ,sto> → <k_1,sto>, <ie_2,sto> → <k_2,sto>,
   <ie_1,sto> → <m_1,sto>, <ie_2,sto> → <m_2,sto>,
   compute(iop,k_1,k_2) = n_1, and *compute*(iop,m_1,m_2) = n_2.
   Then by the induction hypothesis, $k_1 = m_1$ and $k_2 = m_2$.
   Therefore n_1 = *compute*(iop,k_1,k_2) = *compute*(iop,m_1,m_2) = n_2.

Now the result follows by structural induction.

2. An exercise. ∎

Exercises

1. Evaluate the Wren expression "(x+y)+6" using the store, sto = {x↦17, y↦25}. Draw a derivation tree that shows the applications of the inference rules.

2. Evaluate the following Wren expressions using the structural operational specification in Figure 8.6 and the store sto = {a↦6, b↦9, p↦true, q↦false}.
   a) (a<>0) **and not** (p **and** q)
   b) a – (b – (a –1))
   c) (a > 10) **or** (c=0)
   d) b / (a-6)

3. Prove the derived rule for the semantics of Wren expressions:

$$\frac{\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \quad \langle e_2,sto\rangle \rightarrow \langle e_3,sto\rangle \quad \ldots \quad \langle e_{n-1},sto\rangle \rightarrow \langle e_n,sto\rangle}{\langle e_1,sto\rangle \rightarrow \langle e_n,sto\rangle}$$

   with the condition that $n \geq 2$ and every e_i comes from Iexp or every e_i comes from Bexp.

4. Provide additional inference rules in Figure 8.6 so that the system gives meaning to Wren expressions using the unary minus operation. *Hint*: Use *compute* for the arithmetic.

5. Modify the inference system for Wren expressions so that binary expressions can have either the left or the right argument evaluated first.

6. Complete the proof of the completeness theorem for Boolean expressions in Wren.

7. Complete the proof of the consistency theorem for Boolean expressions in Wren.

8. Define rules that specify the meaning of Boolean expressions of the form "b_1 **and** b_2" and "b_1 **or** b_2" directly in the manner of rule (11) for **not**. Then rewrite the specification of Wren expressions so that **and then** and **or else** are interpreted as conditional (short-circuit) operators. A conditional **and**—for example, "b_1 **and then** b_2"—is equivalent to "**if** b_1 **then** b_2 **else false** ".

9. Extend Wren to include conditional integer expressions with the abstract syntax

$$\frac{\text{be} : \text{bexp} \qquad \text{ie}_1 : \text{iexp} \qquad \text{ie}_2 : \text{iexp}}{\textbf{if}\ \text{be}\ \textbf{then}\ \text{ie}_1\ \textbf{else}\ \text{ie}_2 : \text{iexp}}$$

and add inference rule(s) to give them meaning.

8.6 STRUCTURAL OPERATIONAL SEMANTICS: COMMANDS

The structural operational semantics of a command in Wren describes the steps of a computation as the command modifies the state of a machine. We now consider language features—assignment and input—that can change the values bound to identifiers in the store. In addition, the **read** and **write** commands affect the input and output lists associated with the execution of a program. A triple of values represents the state of our abstract machine: the current input list, the current output list, and the current store. Input and output sequences are finite lists of numerals. We use structures of the form st(in,out,sto) to describe the state of a machine at a particular instant, where "in" and "out" are finite lists of numerals, represented using the notation [3,5,8].

A configuration on which the transition system operates contains a command to be executed and a state. Given a command c_0 and an initial state $st(in_0,out_0,sto_0)$, a computation proceeds following a set of inference rules.

$$\langle c_0,st(in_0,out_0,sto_0)\rangle \rightarrow \langle c_1,st(in_1,out_1,sto_1)\rangle \rightarrow \langle c_2,st(in_2,out_2,sto_2)\rangle \rightarrow \ldots.$$

The inference rules for the structural operational semantics of commands in Wren are listed in Figure 8.8. Observe that most commands need not delve into the internal structure of states; in fact, only assignment, **read**, and **write** explicitly modify components of the state. The input and output lists are manipulated by auxiliary functions, *head*, *tail*, and *affix*. The **write** command uses *affix* to append a numeral onto the right end of the output list. For example, *affix*([2,3,5],8) = [2,3,5,8].

Again, the inference rules promote a well-defined strategy for the execution of commands. When the action of a command depends on the value of some expression that serves as a component in the command, we use a rule whose premise describes one step in the reduction of the expression and whose conclusion assimilates that change into the command. See rules (1), (3), and (11) for illustrations of this strategy. When the expression has been reduced to its normal form (a numeral or a Boolean constant), the command carries out its action. See rules (2), (4), (5), and (12) for examples.

(1a) $$\frac{\langle ie,sto\rangle \rightarrow \langle ie',sto\rangle}{\langle id := ie,st(in,out,sto)\rangle \rightarrow \langle id := ie',st(in,out,sto)\rangle}$$

(1b) $$\frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle bid{:=}be,st(in,out,sto)\rangle \rightarrow \langle bid{:=}be',st(in,out,sto)\rangle}$$

(2a) $\langle id := n,st(in,out,sto)\rangle \rightarrow \langle \textbf{skip},st(in,out,\mathit{updateSto}(sto,id,n))\rangle$

(2b) $\langle bid := b,st(in,out,sto)\rangle \rightarrow \langle \textbf{skip},st(in,out,\mathit{updateSto}(sto,bid,b))\rangle$

(3) $$\frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle \textbf{if } be \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle \rightarrow \langle \textbf{if } be' \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle}$$

(4) $\langle \textbf{if true then } c_1 \textbf{ else } c_2,state\rangle \rightarrow \langle c_1,state\rangle$

(5) $\langle \textbf{if false then } c_1 \textbf{ else } c_2,state\rangle \rightarrow \langle c_2,state\rangle$

(6) $\langle \textbf{if } be \textbf{ then } c,state\rangle \rightarrow \langle \textbf{if } be \textbf{ then } c \textbf{ else skip},state\rangle$

(7) $\langle \textbf{while } be \textbf{ do } c,state\rangle \rightarrow \langle \textbf{if } be \textbf{ then } (c \ ; \textbf{while } be \textbf{ do } c) \textbf{ else skip},state\rangle$

(8) $$\frac{\langle c_1,state\rangle \rightarrow \langle c_1',state'\rangle}{\langle c_1 \ ; c_2,state\rangle \rightarrow \langle c_1' \ ; c_2,state'\rangle}$$

(9) $\langle \textbf{skip} \ ; c,state\rangle \rightarrow \langle c,state\rangle$

(10) $\langle \textbf{read } id,st(in,out,sto)\rangle \rightarrow \langle \textbf{skip},st(\mathit{tail}(in),out,\mathit{updateSto}(sto,id,\mathit{head}(in)))\rangle \quad in \neq [\,]$

(11) $$\frac{\langle ie,sto\rangle \rightarrow \langle ie',sto\rangle}{\langle \textbf{write } ie,st(in,out,sto)\rangle \rightarrow \langle \textbf{write } ie',st(in,out,sto)\rangle}$$

(12) $\langle \textbf{write } n,st(in,out,sto)\rangle \rightarrow \langle \textbf{skip},st(in,\mathit{affix}(out,n),sto)\rangle$

Figure 8.8: Semantics for Commands

Note that assignment and the **read** and **write** commands are elementary actions, so they reduce to an "empty command" represented by **skip**. Two commands, **if-then** and **while**, are handled by translation into forms that are treated elsewhere (see rules (6) and (7)). Finally, command sequencing (semicolon) needs two rules—one to bring about the reduction of the first command to **skip**, and the second to discard the first command when it has been simplified to **skip**. Observe that now we have the possibility that a computation may continue forever because of the **while** command in Wren. Rule (7) defines "**while** be **do** c" in terms of itself.

For completeness, we again included an inference rule that makes the transition relation $\rightarrow$ transitive:

$$(13)\quad \frac{\langle c_1,state_1\rangle \rightarrow \langle c_2,state_2\rangle \qquad \langle c_2,state_2\rangle \rightarrow \langle c_3,state_3\rangle}{\langle c_1,state_1\rangle \rightarrow \langle c_3,state_3\rangle}$$

and furthermore assume that $\rightarrow$ is a reflexive relation.

Given a Wren program whose declarations have been elaborated, verifying that it satisfies the context conditions of its syntax, and whose body is the command c, and given a list $[n_1, n_2, n_3, \ldots, n_k]$ of numerals as input, the transition rules defined by the inference system in Figure 8.8 are applied to the initial configuration, $\langle c, st([n_1,n_2,n_3,\ldots,n_k],[\,],\mathit{emptySto})\rangle$, to produce the meaning of the Wren program.

A configuration with the pattern <**skip**,state> serves as a normal form for computations. No rule applies to configurations in this form, and their state embodies the result of the computation—namely, the output list and the final store—when all of the commands have been executed. We have three possible outcomes of a computation that starts with a command and an initial state:

1. After a finite number of transitions, we reach a configuration <**skip**,state> in normal form.
2. After a finite number of transitions, we reach a configuration that is not in normal form but for which no further transition is defined. This can happen when an expression evaluation becomes stuck because of an undefined identifier or division by zero, or upon the failure of the condition on rule (10), which specifies that the input list is nonempty when a **read** command is to be executed next.
3. A computation sequence continues without end when describing the semantics of a **while** command that never terminates. As a simple example, consider the transitions:

   <**while true do skip** ,state>

   $\rightarrow$ <**if true then** (**skip** ; **while true do skip**) **else skip** ,state> (7)

→ <**skip** ; **while true do skip** ,state> (4)
→ <**while true do skip** ,state> (9)
→ <**if true then** (**skip** ; **while true do skip**) **else skip** ,state> (7)
→ <**skip** ; **while true do skip** ,state> (4)
→ <**while true do skip** ,state> (9)
→ ....

We use the notation **<c,state>** → ∞ to denote the property that a computation sequence starting with the configuration <c,state> fails to terminate. It should be obvious from the example above that for any state st, <**while true do skip** ,st> → ∞ .

Using rules (8) and (9) we can derive a new inference rule (9') that will make deductions a bit more concise.

$$(9')\quad \frac{\langle c_1,\text{state}\rangle \rightarrow \langle \textbf{skip},\text{state}'\rangle}{\langle c_1 \textbf{ ; } c_2,\text{state}\rangle \rightarrow \langle c_2,\text{state}'\rangle}$$

Verifying this new rule provides an example of a derivation following the inference system.

$$\dfrac{\dfrac{\langle c_1,\text{state}\rangle \rightarrow \langle \textbf{skip},\text{state}'\rangle}{\langle c_1 \textbf{ ; } c_2,\text{state}\rangle \rightarrow \langle \textbf{skip} \text{ ; } c_2,\text{state}'\rangle}\ (8) \qquad \langle \textbf{skip ; } c_2,\text{state}'\rangle \rightarrow \langle c_2,\text{state}'\rangle\ (9)}{\langle c_1 \textbf{ ; } c_2,\text{state}\rangle \rightarrow \langle c_2,\text{state}'\rangle\ (13)}$$

A Sample Computation

Since the steps in a computation following structural operational semantics are very small, derivation sequences for even simple programs can get quite lengthy. We illustrate the semantics with an example that may well be a test of endurance. The Wren program under consideration consists of the command sequence

mx := 0; **read** z; **while** z≥0 **do** ((**if** z>mx **then** mx:=z);read z); **write** mx

where we assume that all identifiers have been appropriately declared. Meta-parentheses clarify the grouping of this representation of the abstract syntax. The program is given the input list: [5,8,3,-1].

To shorten the description of the derivation, a number of abbreviations will be employed:

c_1 = (mx:=0)

c_2 = **read** z

c_3 = **while** z≥0 **do** ((**if** z>mx **then** mx:=z);read z)

c_4 = **write** mx

c_w = (**if** z>mx **then** mx:=z);read z

{ } = *emptySto*

We start the transition system with the initial state, st([5,8,3,-1],[ ],{ }).

Throughout the derivation, assume that "**if** z>mx **then** mx:=z" is an abbreviation of "**if** z>mx **then** mx:=z **else skip**" to avoid the extra steps using rule (6). For each step in the derivation, the number of the rule being applied will appear at the far right. The details of expression evaluation are suppressed, so we just use "(expr)" to signify a derivation for an expression even though it may consist of several steps. The rule (13) that makes ⟶ transitive is generally ignored, but its result is implied. Here then is the computation according to structural operational semantics.

<mx:=0,st([5,8,3,-1],[ ],{ })> ⟶ <**skip**,st([5,8,3,-1],[ ],{mx|↦0})> (2)

<mx:=0;c_2;c_3;c_4,st([5,8,3,-1],[ ],{ })> ⟶
<**read** z;c_3;c_4,st([5,8,3,-1],[ ],{mx|↦0})> (9')

<**read** z,st([5,8,3,-1],[ ],{mx|↦0})> ⟶ <**skip**,st([8,3,-1],[ ],{mx|↦0,z|↦5})> (10)

<**read** z;c_3;c_4,st([5,8,3,-1],[ ],{mx|↦0})> ⟶ (9')
<(**while** z≥0 **do** c_w);c_4,st([8,3,-1],[ ],{mx|↦0,z|↦5})>

<**while** z≥0 **do** c_w,st([8,3,-1],[ ],{mx|↦0,z|↦5})> ⟶ (7)
<**if** z≥0 **then**(c_w;c_3) **else skip**,st([8,3,-1],[ ],{mx|↦0,z|↦5})>

<z≥0,{mx|↦0,z|↦5}> ⟶ <**true**,{mx|↦0,z|↦5}> (expr)

<**if** z≥0 **then**(c_w;c_3) **else skip**,st([8,3,-1],[ ],{mx|↦0,z|↦5})> ⟶ (3)
<**if true then**(c_w; c_3) **else skip**,st([8,3,-1],[ ],{mx|↦0,z|↦5})>

<**if true then**(c_w;c_3) **else skip**,st([8,3,-1],[ ],{mx|↦0,z|↦5})> ⟶ (4)
<c_w;c_3,st([8,3,-1],[ ],{mx|↦0,z|↦5})> =
<(**if** z>mx **then** mx:=z);**read** z;**while** z≥0 **do** c_w,st([8,3,-1],[ ],{mx|↦0,z|↦5}>

<z>mx,{mx|↦0,z|↦5}> ⟶ <**true**,{mx|↦0,z|↦5}> (expr)

<**if** z>mx **then** mx:=z,st([8,3,-1],[ ],{mx|↦0,z|↦5}> ⟶ (3)
<**if true then** mx:=z,st([8,3,-1],[ ],{mx|↦0,z|↦5}>

<**if true then** mx:=z,st([8,3,-1],[ ],{mx|↦0,z|↦5}> ⟶ (4)
<mx:=z,st([8,3,-1],[ ],{mx|↦0,z|↦5}>

<z,{mx|↦0,z|↦5}> ⟶ <5,{mx|↦0,z|↦5}> (expr)

$\langle mx{:=}z, st([8,3,-1],[\,],\{mx\mapsto 0, z\mapsto 5\})\rangle \rightarrow$ (1)
$\quad \langle mx{:=}5, st([8,3,-1],[\,],\{mx\mapsto 0, z\mapsto 5\})\rangle$

$\langle mx{:=}5, st([8,3,-1],[\,],\{mx\mapsto 0, z\mapsto 5\})\rangle \rightarrow$ (2)
$\quad \langle \textbf{skip}, st([8,3,-1],[\,],\{mx\mapsto 5, z\mapsto 5\})\rangle$

$\langle \textbf{if } z>mx \textbf{ then } mx{:=}z, st([8,3,-1],[\,],\{mx\mapsto 0, z\mapsto 5\})\rangle \rightarrow$ (13)
$\quad \langle \textbf{skip}, st([8,3,-1],[\,],\{mx\mapsto 5, z\mapsto 5\})\rangle$

$\langle (\textbf{if } z>mx \textbf{ then } mx{:=}z); \textbf{read } z; \textbf{while } z\geq 0 \textbf{ do } c_w, st([8,3,-1],[\,],\{mx\mapsto 0, z\mapsto 5\})\rangle \rightarrow$
$\quad \langle \textbf{read } z; \textbf{while } z\geq 0 \textbf{ do } c_w, st([8,3,-1],[\,],\{mx\mapsto 5, z\mapsto 5\})\rangle$ (9')

$\langle \textbf{read } z, st([8,3,-1],[\,],\{mx\mapsto 5, z\mapsto 5\})\rangle \rightarrow \langle \textbf{skip}, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle$ (10)

$\langle \textbf{read } z; \textbf{while } z\geq 0 \textbf{ do } c_w, st([8,3,-1],[\,],\{mx\mapsto 5, z\mapsto 5\})\rangle \rightarrow$ (9')
$\quad \langle \textbf{while } z\geq 0 \textbf{ do } c_w, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\})\rangle$

$\langle \textbf{while } z\geq 0 \textbf{ do } c_w, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$ (7)
$\quad \langle \textbf{if } z\geq 0 \textbf{ then } (c_w; c_3) \textbf{ else skip }, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\})\rangle$

$\langle z\geq 0, \{mx\mapsto 5, z\mapsto 8\}\rangle \rightarrow \langle \textbf{true}, \{m\mapsto 5, \mapsto 8\}\rangle$ (expr)

$\langle \textbf{if } z\geq 0 \textbf{ then } (c_w; c_3) \textbf{ else skip }, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\})\rangle \rightarrow$ (3)
$\quad \langle \textbf{if true then } (c_w; c_3) \textbf{ else skip }, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle$

$\langle \textbf{if true then } (c_w; c_3) \textbf{ else skip }, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$ (4)
$\quad \langle c_w; c_3, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\})\rangle =$
$\langle (\textbf{if } z>mx \textbf{ then } mx{:=}z); \textbf{read } z; \textbf{while } z\geq 0 \textbf{ do } c_w, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\})\rangle$

$\langle z>mx, \{mx\mapsto 5, z\mapsto 8\}\rangle \rightarrow \langle \textbf{true}, \{m\mapsto 5, z\mapsto 8\})\rangle$ (expr)

$\langle \textbf{if } z>mx \textbf{ then } mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$
$\quad \langle \textbf{if true then } mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle$ (3)

$\langle \textbf{if true then } mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$
$\quad \langle mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle$ (4)

$\langle z, \{mx\mapsto 5, z\mapsto 8\}\rangle \rightarrow \langle 8, \{mx\mapsto 5, z\mapsto 8\}\rangle$ (expr)

$\langle mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$
$\quad \langle mx{:=}8, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle$ (1)

$\langle mx{:=}8, st([3,-1],[\,],\{mx\mapsto 5, \mapsto 8\}\rangle \rightarrow$ (2)
$\quad \langle \textbf{skip}, st([3,-1],[\,],\{mx\mapsto 8, z\mapsto 8\}\rangle$

$\langle \textbf{if } z>mx \textbf{ then } mx{:=}z, st([3,-1],[\,],\{mx\mapsto 5, z\mapsto 8\})\rangle \rightarrow$ (13)
$\quad \langle \textbf{skip}, st([3,-1],[\,],\{mx\mapsto 8, z\mapsto 8\})\rangle$

$\langle$(**if** z>mx **then** mx:=z);**read** z;**while** z≥0 **do** c_w,st([3,-1],[ ],{mx$\mapsto$5,z$\mapsto$8})$\rangle$
$\rightarrow$
$\langle$**read** z;**while** z≥0 **do** c_w,st([3,-1],[ ],{mx$\mapsto$8,$\mapsto$8})$\rangle$ (9')

$\langle$**read** z,st([3,-1],[ ],{mx$\mapsto$8,z$\mapsto$8})$\rangle \rightarrow \langle$**skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$ (10)

$\langle$**read** z;**while** z≥0 **do** c_w,st([3,-1],[ ],{mx$\mapsto$8,z$\mapsto$8})$\rangle \rightarrow$ (9')
$\langle$**while** z≥0 **do** c_w,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$**while** z≥0 **do** c_w,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (7)
$\langle$**if** z≥0 **then**(c_w;c_3) **else skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$z≥0,{mx$\mapsto$8,z$\mapsto$3}$\rangle \rightarrow \langle$**true**,{mx$\mapsto$8,z$\mapsto$3}$\rangle$ (expr)

$\langle$**if** z≥0 **then**(c_w;c_3) **else skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (3)
$\langle$**if true then**(c_w;c_3) **else skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$**if true then**(c_w;c_3) **else skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (4)
$\langle c_w;c_3$,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$ =
$\langle$(**if** z>mx **then** mx:=z);**read** z;**while** z≥0 **do** c_w,st([-1],[ ],{m$\mapsto$8,z$\mapsto$3}$\rangle$

$\langle$z>mx,{mx$\mapsto$8,z$\mapsto$3}$\rangle \rightarrow \langle$**false**,{mx$\mapsto$8,z$\mapsto$3})$\rangle$ (expr)

$\langle$**if** z>mx **then** mx:=z,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (3)
$\langle$**if false then** mx:=z,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$**if false then** mx:=z,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (5)
$\langle$**skip**,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$(**if** z>mx **then** mx:=z);**read** z;**while** z≥0 **do** c_w,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$ (9')
$\langle$**read** z;**while** z≥0 **do** c_w,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle$

$\langle$**read** z,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow \langle$**skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$ (10)

$\langle$**read** z;**while** z≥0 **do** c_w,st([-1],[ ],{mx$\mapsto$8,z$\mapsto$3})$\rangle \rightarrow$ (9')
$\langle$**while** z≥0 **do** c_w,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$**while** z≥0 **do** c_w,st([ ],[ ],{m$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (7)
$\langle$**if** z≥0 **then**(c_w;c_3) **else skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$z≥0,{mx$\mapsto$8,z$\mapsto$-1}$\rangle \rightarrow \langle$**false**,{mx$\mapsto$8,z$\mapsto$-1}$\rangle$ (expr)

$\langle$**if** z≥0 **then**(c_w;c_3) **else skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (3)
$\langle$**if false then**(c_w;c_3) **else skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$**if false then**(c_w;c_3) **else skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (5)
$\langle$**skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$**while** $z \geq 0$ **do** c_w,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (13)
$\quad\langle$**skip**,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$**while** $z \geq 0$ **do** c_w;**write** mx,st([8,3,-1],[ ],{mx$\mapsto$0,z$\mapsto$5})$\rangle \rightarrow$ (9')
$\quad\langle$**write** mx,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$mx,{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow \langle 8,\{mx\mapsto$8,z$\mapsto$-1$\}\rangle$ (expr)

$\langle$**write** mx,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (11)
$\quad\langle$**write** 8,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle$**write** 8,st([ ],[ ],{mx$\mapsto$8,z$\mapsto$-1})$\rangle \rightarrow$ (12)
$\quad\langle$**skip** ,st([ ],[8],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$

$\langle c_1;c_2;c_3;c_4$,st([5,8,3,-1],[ ],{ })$\rangle \rightarrow$
$\quad\langle$**skip** ,st([ ],[8],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$ (13)

This linear deduction of the final state represents a derivation tree with axioms at its leaf nodes and the configuration $\langle$**skip**, st([ ],[8],{mx$\mapsto$8,z$\mapsto$-1})$\rangle$ at its root. Clearly, there is no reasonable way we can show the tree for this derivation.

Semantic Equivalence

One justification for formal definitions of programming languages is to provide a method for determining when two commands have the same effect. In the framework of structural operational semantics, we can define semantic equivalence in terms of the computation sequences produced by the two commands.

Definition : Commands c_1 and c_2 are **semantically equivalent,** written $c_1 \equiv c_2$, if both of the following two properties hold:

1. For any two states s and s_f,
   $\langle c_1,s\rangle \rightarrow \langle$**skip**$,s_f\rangle$ if and only if $\langle c_2,s\rangle \rightarrow \langle$**skip**$,s_f\rangle$, and
2. For any state s,
   $\langle c_1,s\rangle \rightarrow \infty$ if and only if $\langle c_2,s\rangle \rightarrow \infty$. ∎

It follows that for semantically equivalent commands, if one gets stuck in a nonfinal configuration, the other must also.

Example : For any c_1,c_2 : cmd and be : bexp,
if be **then** c_1 **else** $c_2 \equiv$ **if not**(be) **then** c_2 **else** c_1.

Proof: Let be : bexp and let c_1 and c_2 be any two commands. Suppose that s = st(in,out,sto) is an arbitrary state.

Case 1: <be,sto> → <**true**,sto> by some computation sequence. Then <**not**(be),sto> → <**not**(**true**),sto> → <**false**,sto> by rules (10) and (11) for expressions in Figure 8.6. Now use rules (3), (4), and (5) for commands to get:

<**if** be **then** c_1 **else** c_2,s> → <**if true then** c_1 **else** c_2,s> → <c_1,s> and

<**if not**(be) **then** c_2 **else** c_1,s> → <**if false then** c_2 **else** c_1,s> → <c_1,s>.

From here on the two computations from <c_1,s> must be identical.

Case 2: <be,sto> → <**false**,sto> by some computation. Proceed as in case 1.

Case 3: <be,sto> → <be',sto> where be' is not a Boolean constant and the computation is stuck. Then both

<**if** be **then** c_1 **else** c_2,s> → <**if** be' **then** c_1 **else** c_2,s> and

<**if not**(be) **then** c_2 **else** c_1,s> → <**if not**(be') **then** c_2 **else** c_1,s>

are stuck computations. ∎

Our definition of semantic equivalence entails a slight anomaly in that any two nonterminating computations are viewed as equivalent. In particular, this means that a program that prints the number 5 endlessly is considered equivalent to another program that runs forever without any output. For terminating computations, however, the definition of semantic equivalence agrees with our intuition for programs having the same behavior.

Natural Semantics

Structural operational semantics takes as its mission the description of the individual steps of a computation. It strives to capture the smallest possible changes in configurations. For this reason, structural operational semantics is sometimes called **small-step semantics** . An alternative semantics takes the opposite view—namely, to describe the computation in large steps providing a direct relation between initial and final states. This version of operational semantics is defined by inference systems to create a so-called **big-step semantics** . The most developed version of big-step semantics, called **natural semantics** , was proposed by a group in France led by Gilles Kahn.

To suggest the flavor of natural semantics, we show in Figure 8.9 several of the inference rules that are used to define the meaning of Wren. In natural semantics, configurations have several possible forms for each kind of language construct:

Expressions: <e,sto>, n, or b, and

Commands: <c,state> or state.

A transition defines a final result in one step, namely

<e,sto> ➡ n or <e,sto> ➡ b, and

<c,state> ➡ state',

We chose to investigate small-step semantics because big-step semantics closely resembles denotational semantics and, in fact, can be viewed as a notational variant of it. See Chapter 9 for a description of denotational semantics. For more on natural semantics as well as structural operational semantics, see the further readings at the end of this chapter.

$$\frac{\langle ie_1,sto\rangle \rightarrow n_1 \qquad \langle ie_2,sto\rangle \rightarrow n_2}{\langle ie_1 \text{ iop } ie_2,sto\rangle \rightarrow \mathit{compute}(iop,n_1,n_2)}$$

$$\frac{\langle be,sto\rangle \rightarrow \text{true} \qquad \langle c_1,st(in,out,sto)\rangle \rightarrow st(in',out',sto')}{\langle \textbf{if } be \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle \rightarrow st(in',out',sto')}$$

$$\frac{\langle be,sto\rangle \rightarrow \text{false} \qquad \langle c_2,st(in,out,sto)\rangle \rightarrow st(in',out',sto')}{\langle \textbf{if } be \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle \rightarrow st(in',out',sto')}$$

$$\frac{\langle c_1,state\rangle \rightarrow state' \qquad \langle c_2,state'\rangle \rightarrow state''}{\langle c_1 \textbf{ ; } c_2,state\rangle \rightarrow state''}$$

Figure 8.9: Some Inference Rules for the Natural Semantics of Wren

Exercises

1. Derive the computation sequence for the following Wren programs. Use [8,13,-1] as the input list.

   a) **read** a; **read** b; c:=a; a:=b; b:=c; **write** a; **write** b

   b) n:=3; f:=1; **while** n>1 **do** (f:=f*n; n:=n-1); **write** f

   c) s:=0; **read** a; **while** a≥0 **do** (s:=s+a; **read** a); **write** s

   d) **read** x; **if** x>5 **then** y := x+2 **else** y := 0

   e) p:=**true**; **read** m; **while** p **do** (**read** a; m:=m*a; p:=**not**(p)); **write** m

2. The following rule provides an alternate definition of the **while** command in Wren:

$$(7')\quad \frac{\langle \textbf{if}\ be\ \textbf{then}\ (c\ \textbf{;}\ \textbf{while}\ be\ \textbf{do}\ c)\ \textbf{else skip}, state\rangle \rightarrow \langle \textbf{skip}, state'\rangle}{\langle \textbf{while}\ be\ \textbf{do}\ c, state\rangle \rightarrow \langle \textbf{skip}, state'\rangle}$$

Show that the inference system with rule (7) replaced by this new rule is equivalent to the original system.

3. Add these language constructs to Wren and provide meaning for them by defining inference rules for their semantics.

a) $$\frac{be : bexp \qquad c : cmd}{\textbf{repeat}\ c\ \textbf{until}\ be : cmd}$$

b) $$\frac{c : cmd \qquad ie : iexp}{\textbf{begin}\ c\ \textbf{return}\ ie\ \textbf{end} : iexp}$$

c) $\textbf{swap}\ (id_1,id_2) : cmd \qquad id_1,id_2 \in Id$

d) Parallel assignment:

$$\frac{ie_1,ie_2 : iexp}{id_1,id_2\ \textbf{:=}\ ie_1,ie_2 : cmd} \quad id_1,id_2 \in Id$$

4. Verify the following semantic equivalences.

a) For any c:cmd, c ; **skip** ≡ c.

b) For any c:cmd, **skip** ; c ≡ c.

c) For any be:bexp and c:cmd, **if** be **then** c **else** c ≡ c, assuming the reduction of be does not become stuck.

d) For any be:bexp and c_1,c_2,c_3:cmd,
(**if** be **then** c_1 **else** c_2) ; c_3 ≡ **if** be **then** (c_1 ; c_3) **else** (c_2 ; c_3).

5. Prove that these pairs of commands are not semantically equivalent:

a) c_3 ; (**if** be **then** c_1 **else** c_2) and **if** be **then** (c_3 ; c_1) **else** (c_3 ; c_2)

b) id_1,id_2 := ie_1,ie_2 and id_1 := ie_1 ; id_2 := ie_2

6. Extend Wren to include a definite iteration command of the form

$$\frac{ie_1 : iexp \qquad ie_2 : iexp \qquad c : cmd}{\textbf{for}\ id := ie_1\ \textbf{to}\ ie_2\ \textbf{do}\ c\ \textbf{end} : cmd} \quad id \in Id$$

whose informal semantics agrees with the **for** command in Pascal. Add inference rules to the structural operational semantics of Wren to give a formal semantics for this new command.

8.7 LABORATORY: IMPLEMENTING STRUCTURAL OPERATIONAL SEMANTICS

In Chapter 2 we developed a scanner and parser that take a text file containing a Wren program and produce an abstract syntax tree. Now we continue, creating a prototype implementation of Wren based on its structural operational semantics.

The transcript below shows a sample execution with the operational interpreter. The program reads a positive decimal integer and converts it into binary by subtracting powers of two. The example illustrates how the input list can be handled in Prolog.

```
>>> Interpreting Wren via Operational Semantics <<<
Enter name of source file: tobinary.wren
    program tobinary is
         var n,p : integer;
      begin
         read n; p := 2;
         while p<=n do p := 2*p end while;
         p := p/2;
         while p>0 do
            if n>= p then write 1; n := n-p
                     else write 0 end if;
            p := p/2
         end while
      end
Scan successful
Parse successful
Enter input list followed by a period: [321].
Output = [1,0,1,0,0,0,0,0,1]
Final Store:
    n    int(0)
    p    int(0)
yes
```

Commands

As with the implementation of the SECD machine in section 8.3, we define a predicate transform(Config,NewConfig) that carries out one computation step for the transition function. A configuration is represented just as it was in section 8.6—namely, st(In,Out,Sto)—except that we need uppercase for Prolog variables. Then rules (3), (4), and (5) defining transitions for **if-then-else** commands become three Prolog clauses whose order requires rule (3) to be last, which means that its more general pattern applies only if the Boolean expression is not in normal form.

```
transform(cfg(if(bool(true),C1,C2),State), cfg(C1,State)).                % 4
transform(cfg(if(bool(false),C1,C2),State), cfg(C2,State)).               % 5
transform(cfg(if(Be,C1,C2),st(In,Out,Sto)),                               % 3
    cfg(if(Be1,C1,C2),st(In,Out,Sto))) :- transform(cfg(Be,Sto),cfg(Be1,Sto)).
```

The predicate transform that reduces expressions is defined later. Rules (6) and (7) in Figure 8.8 translate **while** and **if** commands according to their meaning. Two Prolog clauses perform the required translations.

```
transform(cfg(if(Be,C),State), cfg(if(Be,C,skip),State)).                 % 6
transform(cfg(while(Be,C),State), cfg(if(Be,[C,while(Be,C)],skip),State)). % 7
```

Remember from Chapter 2 that the parser produces a Prolog list of commands as the abstract syntax tree for a sequence of commands. Therefore the command "c_1 ; c_2 ; c_3 ; c_4" comes from the parser as [c_1, c_2, c_3, c_4]. So rule (9) becomes

```
transform(cfg([skip|Cs],State), cfg(Cs,State)).                           % 9
```

and rule (8), which must follow rule (9), becomes

```
transform(cfg([C|Cs],State), cfg([C1|Cs],State1)) :-                      % 8
    transform(cfg(C,State),cfg(C1,State1)).
```

Since a list of commands may become empty, we need an additional clause that has no analogue in Figure 8.8:

```
transform(cfg([ ],State), cfg(skip,State)).
```

Before considering the assignment command, we discuss how to model the finite function that comprises the store. We portray the store as a Prolog structure of the form

```
sto(a, int(3), sto(b, int(8), sto(c, bool(false), nil)))
```

for the store {a$\mapsto$3, b$\mapsto$8, c$\mapsto$false}. The empty store is given by the Prolog atom nil. The auxiliary functions for manipulating the store become predicates defined as follows:

```
updateSto(sto(Ide,V,Sto),Ide,Val,sto(Ide,Val,Sto)).

updateSto(sto(I,V,Sto),Ide,Val,sto(I,V,NewSto)) :-
                                    updateSto(Sto,Ide,Val,NewSto).

updateSto(nil,Ide,Val,sto(Ide,Val,nil)).

applySto(sto(Ide,Val,Sto),Ide,Val).

applySto(sto(I,V,Sto),Ide,Val) :- applySto(Sto,Ide,Val).

applySto(nil,Ide,undefined) :- write('Undefined variable'), nl, abort.
```

Note that when an identifier cannot be found in the store, applySto prints an error message and aborts the execution of the operational interpreter.

If the right side of an assignment is already in normal form, the new binding can be entered into the store immediately. Two clauses correspond to the two parts of rule (2).

```
transform(cfg(assign(Ide,int(N)),st(In,Out,Sto)),                    % 2a
    cfg(skip,st(In,Out,Sto1))) :- updateSto(Sto,Ide,int(N),Sto1).

transform(cfg(assign(Ide,bool(B)),st(In,Out,Sto)),                   % 2b
    cfg(skip,st(In,Out,Sto1))) :- updateSto(Sto,Ide,bool(B),Sto1).
```

We leave the tags produced by the scanner and parser on constants as we place the values in memory.

If the right side of an assignment is not yet in normal form, we call on the transition function for expressions to reduce the right side using the predicate transform(cfg(E,Sto),cfg(E1,Sto)) that provides the operational semantics for expressions. We can combine the two parts of rule (1) in the Prolog implementation, since Prolog is not strongly typed.

```
transform(cfg(assign(Ide,E),st(In,Out,Sto)),                         % 1
    cfg(assign(Ide,E1),st(In,Out,Sto))) :- transform(cfg(E,Sto),cfg(E1,Sto)).
```

Again, because of pattern matching, the more specialized clause head must precede the more general—that is, rule (2) comes before rule (1).

The **read** command is handled by two clauses, one to catch the dynamic error when the input list is empty and one to carry out the operation. Note that the head and tail functions are replaced by pattern matching.

```
transform(cfg(read(Ide),st([ ],Out,Sto)),cfg(skip,st([ ],Out,Sto))) :-   % 10
    write('Attempted read of empty file'), nl, abort.

transform(cfg(read(Ide),st([N|T],Out,Sto)), cfg(skip,st(T,Out,Sto1))) :-  % 10
     updateSto(Sto,Ide,int(N),Sto1).
```

The **write** command uses a Prolog predicate concat that concatenates two lists to affix a value to the right end of the output list.

```
transform(cfg(write(int(N)),st(In,Out,Sto)), cfg(skip,st(In,Out1,Sto))) :-      % 12
        concat(Out,[N],Out1).

transform(cfg(write(E),st(In,Out,Sto)), cfg(write(E1),st(In,Out,Sto))) :-       % 11
        transform(cfg(E,Sto),cfg(E1,Sto)).
```

We need a driver predicate interpret to call the transition predicate transform repeatedly until a normal form configuration with the **skip** command turns up or until the program aborts, if ever.

```
interpret(cfg(skip,FinalState),FinalState).

interpret(Config,FinalState) :-  transform(Config,NewConfig),
                                 interpret(NewConfig,FinalState).
```

Expressions

The three groups of rules for binary expressions must be handled from the most specific to the most general. Rules (7), (8), and (9), having both arguments in normal form, must come first.

```
transform(cfg(exp(Opr,int(N1),int(N2)),Sto), cfg(Val,Sto)) :-                   % 7
    compute(Opr,int(N1),int(N2),Val).

transform(cfg(bexp(Opr,int(N1),int(N2)),Sto), cfg(Val,Sto)) :-                  % 8
    compute(Opr,int(N1),int(N2),Val).

transform(cfg(bexp(Opr,bool(B1),bool(B2)),Sto), cfg(Val,Sto)) :-                % 9
    compute(Opr,bool(B1),bool(B2),Val).
```

For all three rules, the actual computation is isolated in the predicate compute(Opr,A1,A2,Result).

Expressions whose first argument is in normal form are treated in rules (4), (5), and (6). We use E2p for E2'.

```
transform(cfg(exp(Opr,int(N),E2),Sto), cfg(exp(Opr,int(N),E2p),Sto)) :-        % 4
        transform(cfg(E2,Sto),cfg(E2p,Sto)).

transform(cfg(bexp(Opr,int(N),E2),Sto), cfg(bexp(Opr,int(N),E2p),Sto)) :-      % 5
        transform(cfg(E2,Sto),cfg(E2p,Sto)).

transform(cfg(bexp(Opr,bool(B),E2),Sto),
                                cfg(bexp(Opr,bool(B),E2p),Sto)) :-              % 6
                    transform(cfg(E2,Sto),cfg(E2p,Sto)).
```

Those rules in which the left argument is not yet in normal form—namely (1), (2), and (3)—must come last.

```
transform(cfg(exp(Opr,E1,E2),Sto),cfg(exp(Opr,E1p,E2),Sto)) :-          % 1
        transform(cfg(E1,Sto),cfg(E1p,Sto)).

transform(cfg(bexp(Opr,E1,E2),Sto),cfg(bexp(Opr,E1p,E2),Sto)) :-        % 2+3
        transform(cfg(E1,Sto),cfg(E1p,Sto)).
```

Rules (2) and (3) can be folded together by letting Opr stand for both comparisons and Boolean operators. The compute predicate relies on native arithmetic in Prolog and simple pattern matching to carry out the computations. A few examples of clauses for this predicate are listed below.

```
compute(plus,int(M),int(N),int(R)) :- R is M+N.

compute(divides,int(M),int(0),int(0)) :- write('Division by zero'), nl, abort.
compute(divides,int(M),int(N),int(R)) :- R is M//N.

compute(equal,int(M),int(N),bool(true)) :- M =:= N.
compute(equal,int(M),int(N),bool(false)).

compute(neq,int(M),int(N),bool(false)) :- M =:= N.
compute(neq,int(M),int(N),bool(true)).

compute(less,int(M),int(N),bool(true)) :- M < N.
compute(less,int(M),int(N),bool(false)).

compute(and,bool(true),bool(true),bool(true)).
compute(and,bool(P),bool(Q),bool(false)).
```

Observe how a division by zero error causes the interpreter to abort. We use abort to signal a stuck configuration. Also note that the clauses for each operator depend on their order for correctness.

To complete the transition function for the operational semantics of expressions, we still need to handle the two unary operations, logical **not** and unary minus, which are left as exercises, and to deal with identifiers by probing the store (rules 12 and 13 for expressions). One clause defines the transition function for both integer and Boolean identifiers.

```
transform(cfg(ide(Ide),Sto), cfg(Val,Sto)) :- applySto(Sto,Ide,Val).      % 12+13
```

Finally, we need to define the driver predicate evaluate that propels and monitors the computation steps for expressions.

```
evaluate(cfg(int(N),Sto), int(N)).

evaluate(cfg(bool(B),Sto), bool(B)).

evaluate(Config, FinalValue) :-  transform(Config, NewConfig),
                                 evaluate(NewConfig, FinalValue).
```

Top-Level Driver

At the top level we call interpret with an initial configuration containing the command Cmd that makes up the body of the Wren program together with an initial state st(In,[ ],nil)) where In holds the input list obtained from the user. We depend on a predicate go to request the input and print the output.

```
go :-  nl,write('>>> Interpreting Wren via Operational Semantics <<<'), nl, nl,
       write('Enter name of source file: '), nl, readfilename(File), nl,
       see(File), scan(Tokens), seen, write('Scan successful'), nl, !,
       program(prog(Dec,Cmd),Tokens,[eop]), write('Parse successful'), nl, !,
       write('Enter input list followed by a period: '), nl, read(In), nl,
       interpret(cfg(Cmd,st(In,[ ],nil)),st(FinalIn,Out,Sto)), nl,
       write('Output = '), write(Out), nl, nl,
       write('Final Store:'), nl, printSto(Sto), nl.
```

Exercises

1. Supply Prolog definitions for the transition rules for the remaining expression types: **not** and unary minus.

2. Complete the definition of the compute predicate.

3. Extend the prototype interpreter to include the following language constructs.
   a) repeat-until commands
      Command ::= ... | **repeat** Command **until** Expression
   b) conditional expressions
      Expression ::= ... | **if** Expression **then** Expression **else** Expression
   c) expressions with side effects
      Expression ::= ... | **begin** Command **return** Expression **end**

4. Give a definition of the Prolog predicate printSto(Sto) that prints the bindings in the store Sto, one to a line.

8.8 FURTHER READING

Structural operational semantics originated in a seminal technical report by Gordon Plotkin [Plotkin81]. This initial presentation of an operational semantics based on inference rules defines a number of imperative programming constructs using small-step semantics. Early work with big-step semantics, called natural semantics by the group at INRIA, can be found in

[Kahn87]. The logic text [Reeves90] contains an introduction to natural deduction, the logic methodology that provides a basis for structural operational semantics. Also see [Prawitz65] for a more advanced description of natural deduction.

The introduction to formal semantics by Nielson and Nielson [Nielson92] treats both structural operational semantics and natural deduction. They suggest translating such operational definitions of programming languages into prototype implementations using Miranda.

Matthew Hennessy [Hennessy90] has aimed his text at an undergraduate audience, providing many examples of operational specifications using both small-step and big-step semantics, using the terms computation semantics and evaluation semantics. Hennessy considers imperative, functional, and concurrent programming languages in his examples. He also includes an extensive discussion of structural induction with numerous examples.

Egidio Astesiano [Astesiano91] gives a clear and logical presentation of operational semantics based on inference systems. He discusses the use of inference rules to specify abstract syntax and compares small-step and big-step operational semantics. Astesiano also describes the relation between natural semantics and denotational semantics.

Peter Landin's description of the SECD machine appears in [Landin64] and [Landin66]. Many recent texts on functional programming also contain material on SECD machines and their variants, including [Glaser84], [Henson87], [Field88], and [Reade89].

Peter Wegner's survey paper [Wegner72] covers the basics of the Vienna Definition Language. [Pagan81] describes VDL succinctly, using two small programming languages to illustrate this specification method.