Chapter 10

The SPARC Language

This chapter presents SPARC: a parallel and functional language used throughout the book for specifying algorithms.

SPARC is a "strict" functional language similar to the ML class of languages such as Standard ML or SML, Caml, and F#. In pseudo code, we sometimes use mathematical notation, and even English descriptions in addition to SPARC syntax. This chapter describes the basic syntax and semantics of SPARC; we introduce additional syntax as needed in the rest of the book.

1 Syntax and Semantics of SPARC

This section describes the syntax and the semantics of the core subset of the SPARC language. The term ***syntax*** refers to the structure of the program itself, whereas the term ***semantics*** refers to what the program computes. Since we wish to analyze the cost of algorithms, we are interested in not just what algorithms compute, but how they compute. Semantics that capture how algorithms compute are called ***operational semantics***, and when augmented with specific costs, ***cost semantics***. Here we describe the syntax of SPARC and present an informal description of its operational semantics. We will cover the cost semantics of SPARC in Cost Models Chapter . While we focus primarily on the core subset of SPARC, we also describe some ***syntactic sugar*** that makes it easier to read or write code without adding any real power. Even though SPARC is a strongly typed language, for our purposes in this book, we use types primarily as a means of describing and specifying the behavior of our algorithms. We therefore do not present careful account of SPARC's type system.

The definition below shows the syntax of SPARC. A SPARC program is an expression, whose syntax, describe the computations that can be expressed in SPARC. When evaluated an expression yield a value. Informally speaking, evaluation of an expression proceeds involves evaluating its sub-expressions to values and then combining these values to compute the value of the expression. SPARC is a strongly typed language, where every closed

expression, which have no undefined (free) variables, evaluates to a value or runs forever.

Definition 10.1 (SPARC expressions)**.**

Identifier	id	:=	$\ldots$	
Variables	x	:=	id	
Type Constructors	$tycon$	:=	id	
Data Constructors	$dcon$	:=	id	
Patterns	p	:=	x	variable
		|	(p)	parenthesis
		|	p_1, p_2	pair
		|	$dcon\ (p)$	data pattern
Types	τ	:=	$\mathbb{Z}$	integers
		|	$\mathbb{B}$	booleans
		|	$\tau\ [*\tau]^+$	products
		|	$\tau \to \tau$	functions
		|	$tycon$	type constructors
		|	dty	data types
Data Types	dty	:=	$dcon\ [\texttt{of}\ \tau]$	
		|	$dcon\ [\texttt{of}\ \tau]\ \texttt{|}\ dty$	
Values	v	:=	$0 \mid 1 \mid \ldots$	integers
		|	$-1 \mid -2 \mid \ldots$	integers
		|	$\texttt{true} \mid \texttt{false}$	booleans
		|	$\texttt{not} \mid \ldots$	unary operations
		|	$\texttt{and} \mid \texttt{plus} \mid \ldots$	binary operations
		|	v_1, v_2	pairs
		|	(v)	parenthesis
		|	$dcon\ (v)$	constructed data
		|	$\texttt{lambda}\ p\ \texttt{.}\ e$	lambda functions
Expression	e	:=	x	variables
		|	v	values
		|	e_1 op e_2	infix operations
		|	e_1, e_2	sequential pair
		|	$e_1 \texttt{||} e_2$	parallel pair
		|	(e)	parenthesis
		|	$\texttt{case}\ e_1\ [\texttt{|}\ p\ \texttt{=>}\ e_2]^+$	case
		|	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditionals
		|	$e_1\ e_2$	function application
		|	$\texttt{let}\ b^+\ \texttt{in}\ e\ \texttt{end}$	local bindings
Operations	op	:=	$+ \mid - \mid * \mid - \ldots$	
Bindings	b	:=	$x(p) = e$	bind function
		|	$p = e$	bind pattern
		|	$\texttt{type}\ tycon = \tau$	bind type
		|	$\texttt{type}\ tycon = dty$	bind datatype

Identifiers. In SPARC, variables, type constructors, and data constructors are given a name, or an ***identifier***. An identifier consist of only alphabetic and numeric characters

(a-z, A-Z, 0-9), the underscore character ("_"), and optionally end with some number of "primes". Example identifiers include, x', x_1, x_l, $myVar$, $myType$, $myData$, and my_data.

Program ***variables***, ***type constructors***, and ***data constructors*** are all instances of identifiers. During evaluation of a SPARC expression, variables are bound to values, which may then be used in a computation later. In SPARC, variable are ***bound*** during function application, as part of matching the formal arguments to a function to those specified by the application, and also by `let` expressions. If, however, a variable appears in an expression but it is not bound by the expression, then it is ***free*** in the expression. We say that an expression is ***closed*** if it has no free variables.

Types constructors give names to types. For example, the type of binary trees may be given the type constructor $btree$. Since for the purposes of simplicity, we rely on mathematical rather than formal specifications, we usually name our types behind mathematical conventions. For example, we denote the type of natural numbers by $\mathbb{N}$, the type of integers by $\mathbb{Z}$, and the type of booleans by $\mathbb{B}$.

Data constructors serve the purpose of making complex data structures. By convention, we will capitalize data constructors, while starting variables always with lowercase letters.

Patterns. In SPARC, variables and data constructors can be used to construct more complex ***patterns*** over data. For example, a pattern can be a pair (x, y), or a triple of variables (x, y, z), or it can consist of a data constructor followed by a pattern, e.g., $Cons(x)$ or $Cons(x, y)$. Patterns thus enable a convenient and concise way to pattern match over the data structures in SPARC.

Built-in Types. Types of SPARC include base types such as integers $\mathbb{Z}$, booleans $\mathbb{B}$, product types such as $\tau_1 * \tau_2 \ldots \tau_n$, function types $\tau_1 \to \tau_2$ with domain τ_1 and range τ_2, as well as user defined data types.

Data Types. In addition to built-in types, a program can define new ***data types*** as a union of tagged types, also called variants, by "unioning" them via distinct ***data constructors***. For example, the following data type defines a point as a two-dimensional or a three-dimensional coordinate of integers.

$$\begin{array}{lcl} \texttt{type}\ point & = & PointTwo\ \texttt{of}\ \mathbb{Z} * \mathbb{Z} \\ & | & Point3D\ \texttt{of}\ \mathbb{Z} * \mathbb{Z} * \mathbb{Z} \end{array}$$

Recursive Data Types. In SPARC recursive data types are relatively easy to define and compute with. For example, we can define a point list data type as follows

$\texttt{type}\ plist = Nil \mid Cons\ \texttt{of}\ point * plist.$

Based on this definition the list

$$Cons(PointTwo(0,0),\\ Cons(PointTwo(0,1),\\ Cons(PointTwo(0,2), Nil)))$$

defines a list consisting of three points.

Exercise 10.1 (Booleans)**.** Some built-in types such as booleans, $\mathbb{B}$, are in fact syntactic sugar and can be defined by using union types as follows. Describe how you can define booleans using data types of SPARC.

Solution. Booleans can be defined as follows.

$$\texttt{type}\ myBool = myTrue \mid myFalse$$

Option Type. Throughout the book, we use ***option*** types quite frequently. Option types for natural numbers can be defined as follows.

$$\texttt{type}\ option = None \mid Some\ \texttt{of}\ \mathbb{N}$$

Similarly, we can define option types for integers.

$$\texttt{type}\ intOption = INone \mid ISome\ \texttt{of}\ \mathbb{Z}$$

Note that we used a different data constructor for naturals. This is necessary for type inference and type checking. Since, however, types are secondary for our purposes in this book, we are sometimes sloppy in our use of types for the sake of simplicity. For example, we use throughout $None$ and $Some$ for option types regardless of the type of the contents.

Values. Values of SPARC, which are the irreducible units of computation include natural numbers, integers, Boolean values `true` and `false`, unary primitive operations, such as boolean negation `not`, arithmetic negation `-`, as well as binary operations such as logical and `and` and arithmetic operations such as `+`. Values also include constant-length tuples, which correspond to product types, whose components are values. Example tuples used commonly through the book include binary tuples or pairs, and ternary tuples or triples. Similarly, data constructors applied to values, which correspond to sum types, are also values.

As a functional language, SPARC treats all function as values. The anonymous function `lambda` $p.\ e$ is a function whose arguments are specified by the pattern p, and whose body is the expression e.

Example 10.1.

- The function `lambda` $x.x + 1$ takes a single variable as an argument and adds one to it.
- The function `lambda` $(x, y).\ x$ takes a pairs as an argument and returns the first component of the pair.

Expressions. Expressions, denoted by e and variants (with subscript, superscript, prime), are defined inductively, because in many cases, an expression contains other expressions. Expressions describe the computations that can be expressed in SPARC. Evaluating an expression via the operational semantics of SPARC produce the value for that expression.

Infix Expressions. An ***infix expression***, $e_1\ op\ e_2$, involve two expressions and an infix operator op. The infix operators include $+$ (plus), $-$ (minus), $*$ (multiply), / (divide), $<$ (less), $>$ (greater), `or`, and `and`. For all these operators the infix expression $e_1\ op\ e_2$ is just syntactic sugar for $f(e_1, e_2)$ where f is the function corresponding to the operator op (see parenthesized names that follow each operator above).

We use standard precedence rules on the operators to indicate their parsing. For example in the expression

```
3 + 4 * 5
```

the $*$ has a higher precedence than $+$ and therefore the expression is equivalent to $3+(4*5)$.

Furthermore all operators are left associative unless stated otherwise, i.e., that is to say that $a\ op_1\ b\ op_2\ c = (a\ op_1\ b)\ op_2\ c$ if op_1 and op_2 have the same precedence.

Example 10.2. The expressions $5 - 4 + 2$ evaluates to $(5 - 4) + 2 = 3$ not $5 - (4 + 2) = -1$, because $-$ and $+$ have the same precedence.

Sequential and Parallel Composition. Expressions include two special infix operators: ",," and ||, for generating ordered pairs, or tuples, either sequentially or in parallel.

The ***comma*** operator or ***sequential composition*** as in the infix expression (e_1, e_2), evaluates e_1 and e_2 sequentially, one after the other, and returns the ordered pair consisting of the two resulting values. Parenthesis delimit tuples.

The ***parallel*** operator or ***parallel composition*** "||", as in the infix expression $(e_1 \mid\mid e_2)$, evaluates e_1 and e_2 in parallel, at the same time, and returns the ordered pair consisting of the two resulting values.

The two operators are identical in terms of their return values. However, we will see later, their cost semantics differ: one is sequential and the other parallel. The comma and parallel operators have the weakest, and equal, precedence.

Example 10.3.

- The expression

  `lambda` $(x, y).\ (x * x, y * y)$

  is a function that take two arguments x and y and returns a pair consisting of the squares x and y.

- The expression

  $\texttt{lambda}\ (x, y).\ (x * x \mid\mid y * y)$

  is a function that take two arguments `x` and `y` and returns a pair consisting of the squares `x` and `y` by squaring each of `x` and `y` in parallel.

Case Expressions. A ***case expression*** such as

$\texttt{case}\ e_1$
$\texttt{|}\ \texttt{Nil} \Rightarrow e_2$
$\texttt{|}\ \texttt{Cons}\ (x, y) \Rightarrow e_3$

first evaluates the expression e_1 to a value v_1, which must return data type. It then matches v_1 to one of the patterns, Nil or $Cons\ (x, y)$ in our example, binds the variable if any in the pattern to the respective sub-values of v_1, and evaluates the "right hand side" of the matched pattern, i.e., the expression e_2 or e_3.

Conditionals. A conditional or an ***if-then-else expression***, $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, evaluates the expression e_1, which must return a Boolean. If the value of e_1 is true then the result of the if-then-else expression is the result of evaluating e_2, otherwise it is the result of evaluating e_3. This allows for conditional evaluation of expressions.

Function Application. A ***function application***, $e_1\ e_2$, applies the function generated by evaluating e_1 to the value generated by evaluating e_2. For example, lets say that e_1 evaluates to the function f and e_2 evaluates to the value v, then we apply f to v by first matching v to the argument of f, which is pattern, to determine the values of each variable in the pattern. We then substitute in the body of f the value of each variable for the variable. To ***substitute*** a value in place of a variable x in an expression e, we replace each instance of x with v.

For example if function $\texttt{lambda}\ (x, y).\ e$ is applied to the pair `(2,3)` then x is given value `2` and y is given value `3`. Any free occurrences of the variables x and y in the expression e will now be bound to the values `2` and `3` respectively. We can think of function application as substituting the argument (or its parts) into the free occurrences of the variables in its body e. The treatment of function application is why we call SPARC a ***strict*** language. In strict or call-by-value languages, the argument to the function is always evaluated to a value before applying the function. In contrast non-strict languages wait to see if the argument will be used before evaluating it to a value.

Example 10.4.

- The expression

  $(\texttt{lambda}\ (x, y).\ x/y)\ (8, 2)$

  evaluates to 4 since 8 and 2 are bound to x and y, respectively, and then divided.

- The expression

  (`lambda` $(f, x).\ f(x, x))\ (plus, 3)$

  evaluates to 6 because f is bound to the function $plus$, x is bound to 3, and then $plus$ is applied to the pair $(3, 3)$.

- The expression

  (`lambda` $x.$ (`lambda` $y.\ x + y)) \ 3$

  evaluates to a function that adds 3 to any integer.

Bindings. The ***let expression***,

`let` b^+ `in` e `end`,

consists of a sequence of bindings b^+, which define local variables and types, followed by an expression e, in which those bindings are visible. In the syntax for the bindings, the superscript $+$ means that b is repeated one or more times. Each binding b is either a variable binding, a function binding, or a type binding. The let expression evaluates to the result of evaluating e given the variable bindings defined in b.

A ***function binding***, $x(p) = e$, consists of a function name, x (technically a variable), the arguments for the function, p, which are themselves a pattern, and the body of the function, e.

Each ***type binding*** equates a type to a base type or a data type.

Example 10.5. Consider the following let expression.

```
let
  x = 2 + 3
  f(w) = (w * 4, w - 2)
  (y, z) = f(x - 1)
in
  x + y + z
end
```

The first binding the variable x to `2 + 3 = 5`; The second binding defines a function $f(w)$ which returns a pair; The third binding applies the function f to $x - 1 = 4$ returning the pair $(4 * 4, 4 - 2) = (16, 2)$, which y and z are bound to, respectively (i.e., $y = 16$ and $z = 2$. Finally the let expressions adds x, y, z and yields $5 + 16 + 2$. The result of the expression is therefore 23.

Note. Be careful about defining which variables each binding can see, as this is important in being able to define recursive functions. In SPARC the expression on the right of each binding in a `let` can see all the variables defined in previous variable bindings, and can

see the function name variables of all binding (including itself) within the `let`. Therefore the function binding

$$\texttt{x}(p) = e$$

is not equivalent to the variable binding

$$\texttt{x} = \texttt{lambda}\ p.e,$$

because in the prior x can be used in e and in the later it cannot. Function bindings therefore allow for the definition of recursive functions. Indeed they allow for mutually recursive functions since the body of function bindings within the same `let` can reference each other.

Example 10.6. The expression

$$\begin{array}{l}\texttt{let}\\ \quad f(i) = \texttt{if}\ (i < 2)\ \texttt{then}\ i\ \texttt{else}\ i * f(i-1)\\ \texttt{in}\\ \quad f(5)\\ \texttt{end}\end{array}$$

will evaluate to the factorial of 5, i.e., $5 * 4 * 3 * 2 * 1$, which is 120.

Example 10.7. The piece of code below illustrates an example use of data types and higher-order functions.

$$\begin{array}{l}\texttt{let}\\ \quad \texttt{type}\ \mathit{point} = \mathit{PointTwo}\ \texttt{of}\ \mathbb{Z} * \mathbb{Z}\\ \qquad\qquad \mid\ \mathit{PointThree}\ \texttt{of}\ \mathbb{Z} * \mathbb{Z} * \mathbb{Z}\\ \quad \mathit{injectThree}\ (\mathit{PointTwo}\ (x, y)) = \mathit{PointThree}\ (x, y, 0)\\ \quad \mathit{projectTwo}\ (\mathit{PointThree}\ (x, y, z)) = \mathit{PointTwo}\ (x, y)\\ \quad \mathit{compose}\ f\ g = f\ g\\ \quad p0 = \mathit{PointTwo}\ (0, 0)\\ \quad q0 = \mathit{injectThree}\ p0\\ \quad p1 = (\mathit{compose}\ \mathit{projectTwo}\ \mathit{injectThree})\ p0\\ \texttt{in}\\ \quad (p0, q0)\\ \texttt{end}\end{array}$$

The example code above defines a point as a two (consisting of x and y axes) or three dimensional (consisting of x, y, and z axes) point in space. The function $\mathit{injectThree}$ takes a 2D point and transforms it to a 3D point by mapping it to a point on the $z = 0$ plane. The function $\mathit{projectTwo}$ takes a 3D point and transforms it to a 2D point by dropping its z coordinate. The function $\mathit{compose}$ takes two functions f and g and composes them. The function $\mathit{compose}$ is a higher-order function, since id operates on functions.

The point $p0$ is the origin in 2D. The point $q0$ is then computed as the origin in 3D. The point $p1$ is computed by injecting $p0$ to 3D and then projecting it back to 2D by dropping the z components, which yields again $p0$. In the end we thus have $p0 = p1 = (0, 0)$.

Example 10.8. The following SPARC code, which defines a binary tree whose leaves and internal nodes holds keys of integer type. The function *find* performs a lookup in a given binary-search tree t, by recursively comparing the key x to the keys along a path in the tree.

```
type tree = Leaf of ℤ | Node of (tree, ℤ, tree)
find (t, x) =
  case t
  | Leaf y ⇒ x = y
  | Node (left, y, right) ⇒
    if x = y then
      return true
    else if x < y then
      find (left, x)
    else
      find (right, x)
```

Remark.
The definition

```
lambda x . (lambda y . f(x, y))
```

takes a function f of a pair of arguments and converts it into a function that takes one of the arguments and returns a function which takes the second argument. This technique can be generalized to functions with multiple arguments and is often referred to as ***currying***, named after Haskell Curry (1900-1982), who developed the idea. It has nothing to do with the popular dish from Southern Asia, although that might be an easy way to remember the term.