Introduction

The following presents a Functional Programming Language named **SK**. The design is strongly influenced by the St. Andrews Static Language (**SASL**) created by David Turner **[DT79]**. **SASL** has been a source of influence for Functional Programming languages, such as **Miranda** and **Haskell**, with which **SK** has many features in common.

A language processor for the **SK** language has been implemented in **Common Lisp**.

Notation

Expressions are “built up”, or Composed from Symbols and Terms as will be described below.

An Equal Sign (=) causes the **SK** symbol appearing to its left to be given a “Function Definition”, which will henceforth represent the expression appearing to its right.

Relational Equality (==) and its complement (!=) only appear in the meta-language used to describe **SK** below. They are not part of the **SK** language itself; but distinguish cases where meta-symbols such as σ1 and σ2 name the same **SK** symbol, or two symbols (with distinct names.)

An Equivalence Sign ≡ is also meta-language declaring that the expressions to its left and right have equivalent reductions and may therefore be considered interchangeable with each other.

Parentheses are only used to override default associativity, fulfilling their traditional mathematical role. Parentheses must be balanced properly.  
  
Double Right Arrow (⇒) is used to define a “Reduction”. The expression to its left will be said to “reduce” to (or to “be replaced” by) the expression appearing to its right.

Definitions

Symbols: σ have alphanumeric names which must begin with a letter.

Terms: τ are either Symbols or Pairs. Terms not referring to Pairs are sometimes said to be “Atomic”.

Pairs: π are 2-Tuples formed by an operation called **Pair**. The “Dot Operator” (•) is provided as an infix abbreviation for **Pair**. Thus, Pairs can be written in two equivalent ways:

τ1 • τ2 ≡ **Pair** τ1 τ2

The first term in a Pair is referred to as its **Head**, and the second term is referred to as its **Tail**.

The infix Dot Operator is *right associative*:

τ1 • τ2 • τ3 ≡ τ1 • (τ2 • τ3)

Lists

Pairs form Lists to which the Head element is prepended, or “pushed” in front of an existing List, which becomes the Tail of the newly formed List. Lists are either a Pair, whose Tail is another List, or they are the Empty List. Empty Lists are represented by a reserved symbol named **nil**.

The length of a List corresponds to the number of elements it contains. An Empty List has no elements and is therefore defined to be of Length Zero. Non-empty Lists have a length one greater than the length of their Tails. Non-empty Lists will have a Last Pair, the Head of which holds their Last Element.

Any list that has the symbol **nil** as its Tail, is said to form a **nil**-terminated Lists. **nil** and **nil**-terminated lists are sometimes also said to be “Proper Lists”.

Lists having a Last Pair that has some Symbol other than **nil** as its Tail are said to be “Dotted Lists”.

Written Representation of Lists

Just as infix multiplication operators are considered *implicit* between adjacent terms of an algebraic expression, the Dot Operator is similarly considered implicit, and may be elided where lists of terms are read or written:

τ1 τ2 ≡ τ1 • τ2 • **nil**τ1 τ2 • σ ≡ τ1 • τ2 • σ *where* σ*!=* **nil**

An Empty pair of parentheses is allowed as an alternative representation of the Empty List **nil**:

() ≡ **nil**

The Head of a Pair can be either a Symbol or another Pair. Thus, Lists may contain other Lists as elements. In this case, parentheses are required to make the associativity among the Pairs clear:

(τ1 τ2) τ3 τ4 ≡ (τ1 • τ2 • **nil**) • τ3 • τ4 • **nil**

Parentheses are required to distinguish a list containing τ1 as a single element, from the term τ1:

(τ1) ≡ τ1 • **nil**

Abstraction

The reserved symbol named lambda (**λ**) is reserved to represent the Abstraction Operator.

**Note:** Implementations should support use of Unicode characters such as the Greek Letter **λ**. However, Question Mark (**?**) will be accepted as a substitute for **λ** to support more constrained character sets.

Abstraction of a symbol from a term produces a new expression, as follows:

**λ**x x ⇒ **I**  
**λ**x y ⇒ **K** y *where x != y*  
**λ**x (τ1 τ2) ⇒ **S** (**λ**x τ1) (**λ**x τ2)

Currying

A more generalized form of abstraction known as Currying is also defined, as follows:

**λ nil** τ ⇒ **Knil** τ  
**λ** (head • tail) τ ≡ **U** (**λ**head (**λ**tail τ))

Here, abstraction is generalized to “Structures”, expressed as Lists of Lists, in addition to Atomic Symbols. Currying is applied recursively, where Structures contain nested Structures.

**U** implements an operation called Uncurrying, which is described in the next section.

The following syntactic enhancement allows Structures to be abstracted using the notation:

**[**head • tail**]** *τ* ≡ **λ** (head • tail) *τ***[**head**]** *τ* ≡ **λ** (head) *τ*

A pair of Square Brackets is allowed as an alternative representation of an Empty Structure:

**[]** *τ* ≡ **λnil** *τ*

Uncurrying

The Uncurrying operator **U** is used to reduce the result of a Curried expression, as follows:

**U** *τ* head • tail ⇒ *τ* head tail  
**U** *τ* **nil** ⇒ *τ*

Though **U** refers to Uncurrying, we could just as well say that it refers to an “Unpairing” operation which takes an Expression and an “Argument List” as input, applies the Expression to the Head of the List, and then applies the functional expression that resulted from the first application to the Tail of the List.

Uncurrying a Structure operates similarly to the **destructuring-bind** of Lisp. This operation is referred to as “Pattern Matching” in **SASL**.

Application

Evaluation of expressions (known as β-Reduction) is invoked as follows:

**β** *τ*

Reductions proceed from left to right, and are *left associative*. When reducing a pair of terms, the first term is said to be applied to the second. A pair of terms that can be reduced is referred to as a “Redex”. The result of this reduction will then be applied to any subsequent term. An expression that contains no Redex is said to be in “Normal Form”.

τ1 τ2 τ3 ≡ (τ1 τ2) τ3

Reduction

Reduction of terms:

**S** τ1 τ2 τ3 ⇒ τ1 τ3 (τ2 τ3)

**K** τ1 τ2 ⇒ τ1

**Knil**, introduced by Currying, works similarly. However, **Knil** τ1 τ2 reduces to τ1 *if and only if* τ2 == **nil**:

**Knil** τ1 **nil** ⇒ τ1

**I** τ ⇒ τ

**Y** τ ⇒ τ (**Y** τ)

Function Definition

Function Definition might be implemented using the simple Lambda Expression:

*func* x = *τ* ≡ *func* = **λ**x *τ*

However, functions are allowed to make recursive reference to themselves. This works because Function Definition is actually implemented with the help of the **Y** operator:

*func* = *τ* ≡ *func* = **Y** (**λ***func* *τ*)

Primitive Functions

All Functions, including **I** and **Y**, can be defined in terms of the two Primitive Functions **S** and **K**:

**I** = **S** **K** **K**

We can also implement YTuring, equivalent to **Y**, by first defining a function named Z:

Z z h = h (z z h)

Now, using Z:

YTuring = Z Z

Namespaces

The following symbols reserved for use by the **SK** language were introduced in preceding sections:  
  
**I** **S** **K** **Knil** **λ nil** **U** **Y**  
  
Reserved Symbols are defined within a reserved namespace with the name SK. These symbols appear in bold face to distinguish them from ordinary Function Definitions. Though beyond the scope of this brief overview, support for the definition and use of namespaces will be explained in a subsequent edition.

Data Types

Support for String and Numeric literals and their operators will be documented in a subsequent edition.

References

**[DT79]** "A New Implementation Technique for Applicative Languages" by David A. Turner, 1979,

Software-Practice and Experience [vol.9, pp.31-49] John Wiley & Sons, Ltd.