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**.

Notational Conventions

The Double Right Arrow (⇒) denotes a replacement operation known as “Reduction”. The expression to its left will be said to “reduce” to the expression to its right.

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

Relational Equality (**==**) and its complement (**!=**) appear in the meta-language used to describe **SK**, to indicate whether or not two meta-symbols σ1 and σ2 refer to the same actual symbol within **SK**.

An Equal Sign (=) indicates that the **SK** symbol appearing to its left is being given a “Function Definition” to represent the expression appearing to its right.

An Equivalence Sign ≡ is used to indicate that the two expressions to its left and right have equivalent reductions (or evaluations) and are therefore interchangeable with each other.

Matched open and close parentheses are introduced when necessary to override associativity, as has been their traditional role in mathematics.

Definitions

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

Terms: τ are either a Symbol or a Pair.

Pairs: π are formed via an internal function named **Pair**. An infix version of this function is called the “dot operator” (◦). Thus, a pair formed from terms τ1 and τ2 can be written:

τ1 ◦ τ2 ≡ **Pair** τ1 τ2

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

The Pairing Operator is *right associative*:

τ1 ◦ τ2 ◦ τ3 ≡ τ1 ◦ (τ2 ◦ τ3)

Lists

The Pair, or dot operator results in a new List in which the Head element is prepended, or “pushed” onto the front of an existing List, which becomes the Tail of the newly created List. A List is either the Empty List, or a Pair whose Tail is another List. Empty Lists are represented by the symbol **nil**.

The length of a List corresponds to the number of elements it contains. An Empty List has no elements and therefore has a Length of Zero. The length of a non-empty List is one greater than the length of its Tail. Any non-empty List will have a final Pair, the Head of which holds the “last element” of the list.

A non-Empty List that has a final Pair with the symbol **nil** as its Tail is said to be **nil**-terminated. Empty Lists and **nil**-terminated lists are also known as “Proper Lists”. If the tail of the final Pair is any symbol other than **nil**, the list is called a “Dotted List.”

Written Representation of Lists

In a manner similar to the way multiplication operators are considered *implicit* between adjacent terms of an algebraic expression, Pairing Operators are implicit, and may be elided when a list of terms is read or written:

τ1 τ2 ≡ τ1 ◦ τ2 ◦ **nil**τ1 τ2 ◦ σ ≡ τ1 ◦ τ2 ◦ σ

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

(τ1) ≡ τ1 ◦ **nil**

Empty pairs of parentheses are used as an alternative representation of the Empty List:

() ≡ **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**

Abstraction

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

**Note:** Implementations should support use of Unicode characters such as the Greek Letter **λ**. Question Mark (**?**) will be accepted as a substitute for **λ**, in more limited 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 generalized form of abstraction known as Currying is defined, as follows:

**λnil** τ ⇒ K**nil** τ  
**λ** (head ◦ tail) *expr* ≡ **U** (**λ**head (**λ**tail *expr*))

Here, abstraction is generalized to “Templates” which include Lists, in addition to atomic Symbols. Uncurrying will be applied recursively, where the head of a Template is itself a nested Template.

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

A syntactic enhancement allows Templates to be expressed concisely using the following notation:

**[**head ◦ tail**]** *expr* ≡ **λ** (head ◦ tail) *expr***[**head**]** *expr* ≡ **λ** (head) *expr*

A pair of Square Brackets can be used as an alternative representation for an Empty Template:

**[]** *expr* ≡ **λnil** *expr*

Uncurrying

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

**U** *expr* head ◦ tail ⇒ *expr* head tail  
**U** *expr* **nil** ⇒ *expr*

Though **U** refers to Uncurrying, we might just as well say it refers to an “Unpairing” operation that 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 Template operates similarly to **destructuring-bind** in Lisp. This operation is referred to as “Pattern Matching” in **SASL**.

Application

Evaluation of expressions (a.k.a., β-Reduction) is invoked as follows:

β *expr*

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 = *expr* ≡ *func* = **λ**x *expr*

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

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 in a reserved namespace with the name SK. They 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 provided.

Data Types

Internal support for String and Numeric literals and operators will be provided.

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.