Introduction

The following presents a Functional Programming Language named **SK** which is closely based on another such language named **SASL** (the St. Andrews Static Language) created by David Turner [DT79].

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

Conventions

A right arrow ⇒ is used to denote a “reduction” operation which is essentially a replacement operation. The language processor transforms the expression template on the left into the expression template on the right.

The notation := will be used in that the expression to the left is being defined to be the expression on the right.  
  
Relational equality signs = and ≠ may be used to distinguish cases where two symbols σ1 and σ2 are identical to each other.  
  
The equivalence (or identical) sign ≡ is used to indicate that the expressions to the left and to the right are operationally equivalent to each other. The two expressions have the same behavior and are interchangeable with each other.

Matched Open “(“ and Close “)” parentheses will be used to fill their traditional mathematical role – to override associativity. They are only introduced when necessary to ensure proper order of evaluation.

Definitions

Symbols: σ are represented by alphanumeric names which begin with a letter.

Terms: τ are either a symbol or a Pair.

Pairs: π are formed via an infix “dot” operator. The pair formed from terms τ1 and τ2 is written:

τ1 . τ2

The dot operator composes pairs using *right associativity*:

τ1 . τ2 . τ3 ≡ τ1 . (τ2 . τ3)

Note: Turner preferred use of a more legible “colon” to the “dot”. We keep to use of the dot here, following its more traditional use by the **Lisp** family of languages.  
  
The first term in a Pair is referred to as its Head and the second term will be referred to as its Tail.

## Lists

The Empty List is represented by the symbol **nil**. The Empty List has no elements and is therefore said to have a Length of Zero.  
  
An Element may be prepended, or “pushed” onto the front of any existing List by introducing a new Pair (via the dot operator) whose Head holds the new Element whose Tail holds the existing List.

Thus, a List is either the Empty List or a Pair whose Tail is another List. The length of any non-empty List is one greater than the length of its Tail.  
  
In a non-empty List, there will exist a final Pair, the head of which will contain the final element and the tail of which is the Empty List, represented by nil. Such “Proper Lists” are thus said to be nil-terminated.

An exceptional case may arise when the second element of the final Pair is some symbol other than **nil**. Such a Pair, and any Elements prepended onto them, are referred to as “Dotted Lists”.

Written Representation of Lists

Just as the multiplication operator is often elided in algebra, the “dot” operator can be elided when reading or writing a List. It is still implicitly present; and can even be made explicit on input.  
  
τ1. τ2 . σ ≡ τ1 τ2 . σ  
  
The Empty List contains no elements and need not be written explicitly following a final element.

τ1 . τ2 . nil ≡ τ1 τ2  
  
Because the Head of a Pair may be either a Symbol or a Term, Lists may contain other Lists as elements.  
  
Lists are comprised of Pairs. Parentheses may be required to make associativity among the Pairs clear:  
  
(τ1 τ2) τ3 τ4 ≡ (τ1 . τ2 . nil) . τ3 . τ4 . nil

Expressions

Lists are a “Data Structure” which will be used to represent Expressions.  
  
An expression: *expr* is a list of terms or abstractions.

Application

Evaluation of an expression is performed in using Beta Reduction, which is invoked as follows:  
  
beta *expr*

When two terms appear next to each other, the first term is said to be applied to the second. Applications are left associative:

τ1 τ2 τ3 ≡ (τ1 τ2) τ3

Abstraction

Abstraction of a symbol from a term, produces a term:

λx x ⇒ I

λx y ⇒ K y

λx (τ1 τ2) ⇒ S (λx τ1) (λx τ2)

Currying

Square brackets are “syntactic sugar” for an enhanced form of Abstraction known as Currying:

[head . tail] *expr* ≡ λ (head . tail) *expr*  
  
λ (head . tail) τ ≡ U (λhead (λtail τ))  
  
λnil τ ⇒ Knil τ

Note: Currying behaves similarly to the Lisp **destructuring-bind** operation.

Uncurrying

The Uncurrying Combinator U implements (or “realizes”) the semantics of Currying:

U *func* (head . tail) ⇒ *func* head tail

Reduction

Reduction of terms:

S τ1 τ2 τ3 ⇒ τ1 τ3 (τ2 τ3)

K τ1 τ2 ⇒ τ1

A similar combinator, introduced by Currying, only reduces where τ2 is nil:

Knil τ1 nil ⇒ τ1

I τ ⇒ τ

Y τ ⇒ τ (Y τ)

Function Definition

Functions: *func* are defined in terms of expressions; and may make recursive reference to themselves:

def *func* x = *expr* ≡ def *func* = λx *expr*

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

Note: I and Y can be defined in terms of S and K:

I ≡ S K K

And given:

def Z z h = h (z z h)

Y ≡ Z Z

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.