SimPL Interpreter Documentation

Yanheng Wang

1 Overview

SimPL is a variant of *ML* that supports both functional and imperative programming. This documentation begins with introducing the syntax and semantics of the language itself. Then, we explicate the type system of the interpreter, including the implementation of type inference algorithm. The next few sections are concerned with library functions, streams and other features. Finally, we give several example programs to demonstrate the power of *SimPL*.

2 Leximes

Literals

- Integers: any decimal numbers between 0 and $2^{31} - 1$.
- Booleans: `true` and `false`.
- Unit: `()`.
- Empty list: `nil`.

Identifiers

Identifiers are matched by the regular expression `[_a-z][_a-zA-Z0-9']`, i.e. begins with a lower case letter or underscore, followed by letters, digits, underscores or quotes.

Keywords and Operators

- Branching: `if`, `then`, `else`.
- Sequencing: `;`
- Looping: `while`, `do`.
- Binding values to names: `let`, `=`, `in`, `end`.
- Allocating memory: `ref`.
- Assignment and dereference: `:=`, `!`.
- Defining functions/recursions/streams: `fn`, `rec`, `stream`, `=>`.
- Arithmetic operations: `+`, `-`, `*`, `/`, `%`, `~`.
- Comparison: `=`, `<>`, `<`, `<=`, `>`, `>=`
- Boolean operations: `not`, `andalso`, `orelse`.
- List/pair/stream operations: `::`, `,`, `>>`, `(`, `)`, `[`, `]`.

See the next two sections for their usage and meanings.

3 Syntax

$t ::=$	x	(identifiers)
$\mid$	n	(integers)
$\mid$	`true`	(true)
$\mid$	`false`	(false)
$\mid$	`nil`	(empty list)
$\mid$	(t, t)	(pair)
$\mid$	$\sharp t$	(unary operation)
$\mid$	$t \otimes t$	(binary operation)
$\mid$	`fn` x `=>` t	(anonymous function $x \mapsto t$)
$\mid$	`rec` x `=>` t	(recursion of name x)
$\mid$	`stream` t `=>` t	(stream with seed and generator)
$\mid$	$t\ t$	(function application)
$\mid$	`if` t `then` t `else` t	(branching)
$\mid$	`let` $x = t$ `in` t `end`	(binding)
$\mid$	`while` t `do` t	(loop)
$\mid$	(t)	(group)
$\mid$	$()$	(unit)

where $\sharp \in \{\texttt{~}, \texttt{not}, \texttt{!}\}$, and $\otimes \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{::}, \texttt{=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{<>}, \texttt{:=}, \texttt{andalso}, \texttt{orelse}\}$.

Of course, the internals of our interpreter are not text-based; the source program is parsed into an abstract syntax tree at the beginning. Each kind of syntactic construct is a subclass of Expr. Take t_1+t_2 for example: It is represented by a class Add extends Expr which has two pointers that point to t_1 and t_2 respectively.

The lexer and parser are automated by *JFlex* and *Java CUP* libraries. You can refer to the `simpl.lex` and `simpl.grm` files for specification.

4 Operational Semantics

In Theory

In order to avoid clutter, we choose to present the semantics in its puriest form – using substitution rather than environment. The practical considerations will be addressed later. In the following rules, M is the metavariable for memory configurations.

$$\frac{n \text{ is an integer literal}}{M, n \Downarrow M, \mathsf{intVal}(n)} \quad \text{(E-Int)}$$

$$\frac{b = \texttt{true or false}}{M, b \Downarrow M, \mathsf{boolVal}(b)} \quad \text{(E-Bool)}$$

$$\frac{}{M, () \Downarrow M, \mathsf{unitVal}} \quad \text{(E-Unit)}$$

$$\frac{}{M, \texttt{nil} \Downarrow M, \mathsf{nilVal}} \quad \text{(E-Nil)}$$

$$\frac{M, t \Downarrow M', v}{M, (t) \Downarrow M', v} \quad \text{(E-Group)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2 \Downarrow M'', v_2}{M, (t_1, t_2) \Downarrow M'', \mathsf{pairVal}(v_1, v_2)} \quad \text{(E-Pair)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2 \Downarrow M'', v_2}{M, t_1 :: t_2 \Downarrow M'', \mathsf{consVal}(v_1, v_2)} \quad \text{(E-Cons)}$$

$$\frac{}{M, \texttt{fn}\ x\ \texttt{=>}\ t \Downarrow M, \mathsf{funVal}(x, t)} \quad \text{(E-Fun)}$$

$$\frac{M, t[x \mapsto (\texttt{rec}\ x\ \texttt{=>}\ t)] \Downarrow M', v}{M, \texttt{rec}\ x\ \texttt{=>}\ t \Downarrow M', v} \quad \text{(E-Rec)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{funVal}(x, t) \quad M', t_2 \Downarrow M'', v \quad M'', t[x \mapsto v] \Downarrow M''', v'}{M, t_1\ t_2 \Downarrow M''', v'} \quad \text{(E-App)}$$

$$\frac{M, t \Downarrow M', v \quad v' = \sharp v}{M, \sharp t \Downarrow M', v'} \quad \text{(E-Unary)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2 \Downarrow M'', v_2 \quad v = v_1 \otimes v_2}{M, t_1 \otimes t_2 \Downarrow M'', v} \quad \text{(E-Binary)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{boolVal}(\texttt{true}) \quad M', t_2 \Downarrow M'', v_2}{M, \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow M'', v_2} \quad \text{(E-If-True)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{boolVal}(\texttt{false}) \quad M', t_3 \Downarrow M'', v_3}{M, \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow M'', v_3} \quad \text{(E-If-False)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{boolVal}(\texttt{true}) \quad M', \texttt{while}\ t_1\ \texttt{do}\ t_2 \Downarrow M'', v}{M, \texttt{while}\ t_1\ \texttt{do}\ t_2 \Downarrow M'', v} \quad \text{(E-Loop)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{boolVal}(\texttt{false})}{M, \texttt{while}\ t_1\ \texttt{do}\ t_2 \Downarrow M', \mathsf{unitVal}} \quad \text{(E-Break)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2 \Downarrow M'', v_2}{M, t_1; t_2 \Downarrow M'', v_2} \quad \text{(E-Seq)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2[x \mapsto v_1] \Downarrow M'', v_2}{M, \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \Downarrow M'', v_2} \quad \text{(E-Let)}$$

$$\frac{M, t \Downarrow M', v \quad l \text{ is a new cell}}{M, \texttt{ref}\ t \Downarrow M'[l \mapsto v], \mathsf{refVal}(l)} \quad \text{(E-Ref)}$$

$$\frac{M, t \Downarrow M', \mathsf{refVal}(l)}{M, !t \Downarrow M', M'(l)} \quad \text{(E-Deref)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{refVal}(l) \quad M', t_2 \Downarrow M'', v_2}{M, t_1 := t_2 \Downarrow M''[l \mapsto v_2], \mathsf{unitVal}} \quad \text{(E-Assign)}$$

$$\frac{M, t_1 \Downarrow M', v_1 \quad M', t_2 \Downarrow M'', v_2}{M, \texttt{steam}\ t_1\ \texttt{=>}\ t_2 \Downarrow M'', \mathsf{streamVal}(v_1, v_2, I, \infty)} \quad \text{(E-Stream)}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{streamVal}(v_1, v_2, f, n) \quad M', t_2 \Downarrow M'', \mathsf{intVal}(m) \quad l = \min(m, n)}{M, t_1\,[t_2] \Downarrow M'', \mathsf{streamVal}(v_1, v_2, f, l)} \tag{E-Trunc}$$

$$\frac{M, t_1 \Downarrow M', \mathsf{streamVal}(v_1, v_2, f, n) \quad M', t_2 \Downarrow M'', f'}{M, t_1 \texttt{ >> } t_2 \Downarrow M'', \mathsf{streamVal}(v_1, v_2, f \circ f', n)} \tag{E-Pass}$$

$$\frac{M, t \Downarrow M', \mathsf{streamVal}(v_1, v_2, f, n)}{M, t \texttt{ >> end} \Downarrow M'', L(v_1, v_2, f, n)} \tag{E-Dispatch}$$

Note that we omit some technical details in rules E-Unary and E-Binary. The operations $\sharp$ and $\otimes$ carry a natural meaning. For example, if $\otimes$ is $<$, then it returns a boolVal that indicates whether the first value is smaller than the second. What's more, the logic operators actually support short-circuit evaluation, which is not presented in the rule.

The last four rules deserve some explanations. They are intended for *stream operations*. Rule E-Stream creates a stream with four parameters:

- v_1 is the seed, i.e. the first value of the stream;
- v_2 is the generator function. Applying it iteratively on the seed will generate an infinite sequence of values;
- I is the processing pipeline. It's initialized to identity function;
- ∞ is the limit of the stream. It's infinity at the beginning.

Bearing this in mind, E-Trunc doesn't need further explanation. The rule E-Pass modifies the processing pipeline by "appending a process stage". Finally, the rule E-Dispatch signals the end of pipeline and "dispatches" the stream for processing. Please be noted that the stream is evaluated *lazily*. That is, unless we trigger E-Dispatch, the stream will not produce any sequence.

Here we are sort of abusing notation: $L(v_1, v_2, f, n)$ should be treated as a blackbox. It takes the seed, generator, pipeline and limit of the stream, and does all the work in a loop. Specifically, it generates elements one by one until the limit is met; for each element x it currently considers, it applies all the pipelined process f on x before collecting the result into a list. When the work is done, it returns the final list in the format of listVal.

In Practice

In the interpreter, every class inherited from Expr implements a eval(state) method, which returns the evaluation of itself under the current state. This is basically a top-down and recursive procedure in the syntax tree.

Conceptually, the current state contains a memory allocation table (i.e. which memory cells are in use, and which are not). We implement it by a HashMap from integers to values. For example, if cell i stores value v, then we have an entry $i \mapsto v$ in the hash map.

The allocation always starts from cell 0 and all the way up. In other words, the very first call of E-Ref would allocate cell 0, the second call would allocate cell 1, and so on. Therefore, we must maintain an additional accumulator (the *top* pointer) in our current state, in order to keep track of the next cell to be allocated.

Concerning efficiency, it should come at no surprise that we prefer the environment model than the substitution model. So we also keep a dynamic binding table that matches the identifiers to values. In principle, however, the evaluation rules are quite similar.

5 Type System

Types

SimPL is a strongly-typed language. Every term has a type. Below is a full list of possible types.

$$
\begin{array}{rll}
T ::= & \mathsf{Int} & \text{(integer type)} \\
| & \mathsf{Bool} & \text{(boolean type)} \\
| & \mathsf{Unit} & \text{(unit type)} \\
| & \mathsf{List}(T) & \text{(list types)} \\
| & \mathsf{Ref}(T) & \text{(reference types)} \\
| & \mathsf{Stream}(T) & \text{(stream types)} \\
| & T \times T & \text{(pair types)} \\
| & T \to T & \text{(function types)} \\
| & X & \text{(free type)} \\
| & \mathsf{Poly}(T, S) & \text{(polymorphic types)}
\end{array}
$$

Type Inference and Checking

Even though the language is strongly-typed, users are not expected to annotate the types explicitly. All types can be automatically inferred from the syntax structure. This feature gives programmers much freedom.

The type inference algorithm shares many similarities with the well-known *Gaussian elimination* in linear algebra. Briefly speaking, the procedure posits every term a free type (i.e. a type variable), and introduces equations (constraints) when necessary. When it finishes scanning the abstract syntax tree, it has collected a bunch of (possibly inconsistent) equations. Then it tries to solve the system. It considers the equations one by one, eliminating one type variable at a time by replacing it with other variables. This stage is called *unification*, a standard term in logic.

The following table exemplifies how the equations are generated.

Situation	Equation(s) generated	Return type
$t_1 + t_2$	$T_1 = \mathsf{Int}$ and $T_2 = \mathsf{Int}$	Int
$t_1 < t_2$	$T_1 = \mathsf{Int}$ and $T_2 = \mathsf{Int}$	Bool
`not` t	$T = \mathsf{Bool}$	Bool
`nil`	none	$\mathsf{List}(X)$
$t_1 :: t_2$	$T_2 = \mathsf{List}(T_1)$	T_2
(t_1, t_2)	none	$T_1 \times T_2$
$t_1\ t_2$	$T_1 = T_2 \to Y$	Y
`stream` t_1 `=>` t_2	$T_2 = T_1 \to T_1$	$\mathsf{Stream}(T_1)$
t_1`[`t_2`]`	$T_1 = \mathsf{Stream}(X)$ and $T_2 = \mathsf{Int}$	T_1
t_1 `>>` t_2	$T_1 = \mathsf{Stream}(X)$ and $T_2 = X \to Y$	$\mathsf{Stream}(Y)$

And the next table illustrates how the type variables are unified.

Equation	Solution / Method to find solution
$X = \mathsf{Int}$	$[X \mapsto \mathsf{Int}]$
$\mathsf{Bool} = X$	$[X \mapsto \mathsf{Bool}]$
$\mathsf{Int} = \mathsf{Unit}$	type error
$X = Y$	$[X \mapsto Y]$
$Y \to \mathsf{Bool} = X \to Z$	solving $Y = X$ and $\mathsf{Bool} = Z$
$\mathsf{Bool} = \mathsf{Int} \to Z$	type error
$\mathsf{Ref}(T) = \mathsf{Stream}(T)$	type error

Although this approach is fine, there is a minor flaw: It has to memorize all the equations it collected, before it finally enters the stage of unification. This leads to unnecessary consumption in space and time.

We could optimize the algorithm by doing unification "on thr fly". That is, when a new equation is generated, we do the unification right away. (Clearly, there's no difference from collecting first and solving later.) In this way, when the scan completes, the solving completes. The corresponding implementation is particularly intricate, where we have to ensure the eliminated variables *indeed disappear in every corner*, preventing them from "dangling" in their parent nodes of the tree. You are invited to read through the source code for detail.

Let-Polymorphism

The type inference system gives rise to a useful feature – the let-polymorphism. It allows a single function (declared in `let`) to be *instantiated into different types of the same shape* in different places.

As an example, let-polymorphism enables us to write the following program which was ill-typed before.

```
1 let applyTwice = (fn f => fn x => f (f x)) in
2     applyTwice (fn x => x+1) 0
3     ...
4     applyTwice (fn b => not b) true
5 end
```

Suppose we are working on `let` $x = t_1$ `in` t_2. Let-polymorphism can be accomplished by the following steps:

(1) Get $t_1 : T_1$, where T_1 is the principal type.
(2) Collect the set $S := \{X \mid X \text{ occurs in } T_1 \text{ but not in } \Gamma\}$. That is, we seek for all free variables in T_1 that are *not* introduced by Γ. Variables in Γ are excluded because they bear contraints from the parent nodes, and thus are not free at all.
(3) Typecheck t_2 under the typing context $\Gamma \cup \{\mathsf{Poly}(T_1, S)\}$.
(4) When the interpreter retrieves a polymorphic type $\mathsf{Poly}(T, S)$ from the typing context, it doesn't return it directly. Instead, it first replaces all free variable $X \in S$ in the type T by fresh ones. In other words, the type $\mathsf{Poly}(T, S)$ serves as a template that may reproduce infinitely many copies on demand. The copies are of the same shape, but they are mutually independent.

We shall see several useful library functions in the next section. Without the aid of let-polymorphism, they are hardly of any use.

6 Library Functions

It appears that some pieces are missing in the language, namely, the ability to access the components in a pair, the facilitiy of retrieving head and tail of a list, and so on. Actually, these features are shipped *as library functions rather than syntactic constructs*. They fit in the general framework and behave as ordinary functions. This gives us great freedom to reuse names, since identifiers such as `fst` and `snd` are *not* reserved as keywords.

Operations on Pair

- `fst` : $X \times Y \to X$ retrieves the first component.
- `snd` : $X \times Y \to Y$ retrieves the second component.

Operations on List

- `hd` : $\mathsf{List}(X) \to X$ retrieves the head element. (Throws runtime error if the list is empty.)
- `tl` : $\mathsf{List}(X) \to \mathsf{List}(X)$ retrieves the tail list. (Throws runtime error if the list is empty.)
- `toStream` : $\mathsf{List}(X) \to \mathsf{Stream}(X)$ transform the list to a finite stream.
- `sum` : $\mathsf{List}(\mathsf{Int}) \to \mathsf{Int}$ sums up all the elements in an integer list. (0 if it's empty.)
- `min` : $\mathsf{List}(\mathsf{Int}) \to \mathsf{Int}$ returns the minimum in an integer list. (Throws runtime error if it's empty.)
- `max` : $\mathsf{List}(\mathsf{Int}) \to \mathsf{Int}$ returns the maximum in an integer list. (Throws runtime error if it's empty.)

Number Theory Functions

- `pred` : $\mathsf{Int} \to \mathsf{Int}$ returns the predecessor of a natural number.
- `succ` : $\mathsf{Int} \to \mathsf{Int}$ returns the successor of a natural number.
- `iszero` : $\mathsf{Int} \to \mathsf{Bool}$ compares a natural number with 0.

Garbage Collection

- `gc` : `unit` is a special unit value that collects garbage. Not being a function, it is *not* invoked by function application. Rather, it's invoked when retrieving identifier `gc` from the environment. Typically, users would write something like `...; gc; ...`. At the moment the interpreter tries to figure out the meaning of the identifier `gc`, the garbage collection mechanism will be invoked automatically.

For the curious, let me explain how the garbage collection mechanism works. It uses a mark-and-sweep algorithm to clear all memory cells that are not referenced (directly or indirectly). There are three key issues to address:

How are the mark bits stored? Recall that our memory allocation is represented as a HashMap from integers to values. Now, We extend it to include a mark bit. That is, a HashMap from integers to value-mark pairs.

Where should we start marking? From all the "living" identifiers at present. We maintain a stack in the current state. Each time an identifier is introduced by the `let` statement,

we push the identifier onto the stack; at the point we leave the `let` statement, we pop the identifier out. We start marking from *both the stack and the current environment*. (They could be unequal because of the E-App rule. That's why we need a stack – it maintains the *global* living identifiers.)

How do we mark the cells so that nobody is left off? This is basically a recursive process. That is, if some value is made up of other values, then we recursively go inside and repeat the marking process. (For the function value, we also go into its bound environment.) But we have to be careful: never go to a memory cell that has already been marked. Otherwise, the process might not terminate.

7 More About Streams

Stream operations are surprisingly powerful and succinct. In this section, I give several examples on its applications.

Truncating a List

Suppose we have a list `lst` of 100 elements, but we are only concerned with its first 20 elements. With streams, this is just a one-liner:

```
let lst' = (toStream lst)[20] >> end in ... end
```

Modifying a List

For the same list, we want to double each element in it. Again, stream is a nice helper:

```
let lst' = (toStream lst) >> (fn x => 2*x) >> end in ... end
```

Generating a List

In practical situations, we frequently want to generate a specific list by certain rules. Just accomplish this with streams!

```
let s = (stream 1 => (fn x => x+1)) in
  (* s is an infinite stream 1, 2, ... *)
  let l1 = (s[8] >> end) in
  let l2 = (s >> (fn x => x*x) [10] >> end) in
  let l3 = (s[3] >> (fn x => (x+1, false)) >> end) in
    (* l1 = [1,2,3,...,8]
       l2 = [1,4,9,...,100]
       l3 = [(2,false), (3,false), (4,false)] *)
  end end end
end
```

8 Advanced Features

Mutual Recursion

SimPL doesn't provide a specialized syntax for mutual recursions. However, it is not formidable to build one by the tools at hand. Suppose we want to define mutually recursive functions `even` : Int → Bool and `odd` : Int → Bool. We may write

```
let funcPair = rec p =>
  ( fn x => if x=0 then true else if x=1 then false else (snd p) (x−1),
    fn x => if x=1 then true else if x=0 then false else (fst p) (x−1) )
in
let even = fst funcPair in
let odd = snd funcPair in
  ...
end end
end
```

In other words, we are defining a "function pair" `p` that is able to refer to itself. The first component of the pair gives `even`, while the second gives `odd`.

Tail Recursion Optimization

In functional programming, the term *tail recursion* means a function calls itself at the tail of the its body. We will define it formally later, but let us use this intuition at the moment. In sight of a tail recursion, the local parameters can be overriden by the new ones directly. Why? Imagine this: when the recursive call returns, there's no more operation to perform (since we are at the tail of the function body), thus the local parameters are useless even if we kept them there.

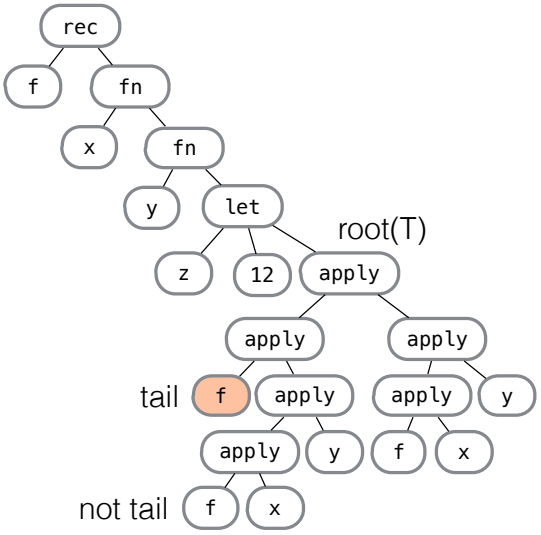
Overriding the parameters brings us an advantage: the space consumed remains constant when the recursion goes deeper and deeper, so we are free of stack overflow error.

Now comes the core question: How do we spot a tail recursion and optimize it? In compiled languages such as C, this is relatively easy because the parameters are passed *as a whole* through stack. However, in interpreted languages that uses currying, the answer is quite complicated.

Definition 1 (tail)**.** A *tail* in a recursive function f is an occurence of f that, after applying it to parameters consecutively, it would return all the way up to the header of f, without further evaluation. (But of course, before it returns, it shall run the recursive call.)

The general pattern is that, a tail always appears as the leftmost leaf u in a subtree T where (a) Returning from root(T) to the function header, no more evaluation is needed; and (b) Every node in the path root(T) $\rightsquigarrow u$ is of class App. This property follows directly from the definition of tail.

The following diagram illustrates the idea.



The discussion points out an obvious way to spot and mark the tails. It is again a top-down procedure. But after that, how do we optimize them? Our solution is to *defer* it when we encounter a tail call:

(1) When passing an argument to a tail, we restrain ourselves from calling the tail in the real. Instead, we wrap the tail and the argument in a LazyAppValue and return directly, pretending that the call is finished.
(2) Similarly, when passing another argument to a LazyAppValue, we pick up the argument into the LazyAppValue as well. This resembles rolling a snowball all the way up.
(3) Finally, when we arrive at the header of the recursion, we don't return as usual. Instead, we unpack the LazyAppValue, override the parameters in the environment by those arguments collected along the way, and then call the tail recursively.

Such implementation of the optimization is entirely transparent to the user, and has no impact on runtime efficiency. (In fact, it speeds up the program because the stack is kept flat.)

9 Example Programs

In this section, we explore four possibilities of performing the same task in *SimPL*. The task we consider is computing $\sum_{i=1}^{50} i^2$.

Imperative Programming

```
let sum = ref 0 in    (* allocate a cell containing 0 *)
  let i = ref 1 in    (* allocate an induction variable *)
    while !i <= 50 do
      sum := !sum + (!i)*(!i);
      i := !i + 1
  end;
  !sum    (* return the result *)
end
```

Functional Programming

```
let squareSum = rec f =>
  (* f can be referenced by the function body to make recursive call *)
  fn n =>
    if n = 1 then 1 else n*n + f (n-1)
in
  squareSum 50
end
```

Functional Programming with Tail Recursion

```
let squareSum = rec f =>
  (* f can be referenced by the function body to make recursive call *)
  fn n => fn result =>
    if n = 0 then result else f (n-1) (n*n + result)
in
  squareSum 50 0
end
```

Functional Programming with Stream

```
let squareStream = stream 1 => (fn x => x+1) in
    sum (squareStream[50] >> (fn x => x*x) >> end)
end
```