Project Report: Generator in Lambda Calculus

Yijia Huang[1]

Iowa State University, Ames, IA, 50011, USA,
hyj@iastate.edu,

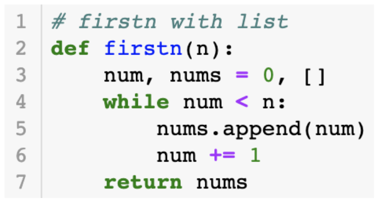
Abstract. One of the beauties for programming languages is to optimize massively redundant program instructions in order to lower the implementation effort by human beings. *while* as one of the most significant features in programming languages is used for iterating over a *list* with the repeated instructions. However, memory usage could not be affordable if the stream data contains a large number of values. In that case, we proposed a generator that yields items instead of returning a *list* for memory efficiency. It is very similar to the implementation that built a list in memory but has the memory usage characteristic of the iterator implementation. Nonetheless, if the iteration needs to be performed multiple times, then generating a list in memory might be worth it since making an integer is a very expensive process. In other words, a generator will provide performance benefits only if we do not intend to use that set of generated values more than once and the memory usage of the *list* is reasonable.

1 Introduction

For this course project, I extended the lambda calculus language with the generator feature for memory efficiency in iterating operations.

1.1 Motivation

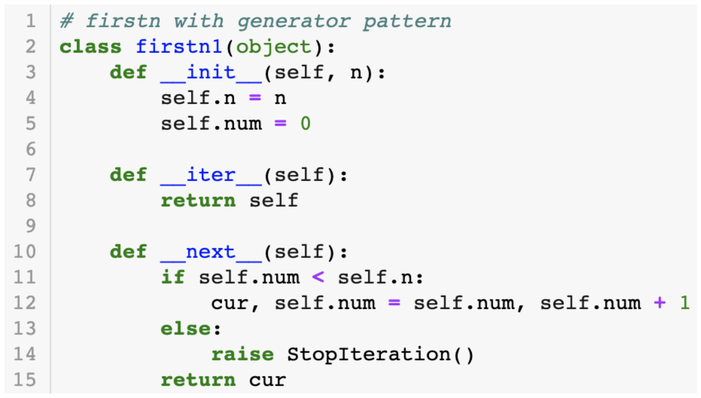
Generator functions can generate a function that behaves like an iterator. They allow programmers to make an iterator in a fast, easy and clean way without too much memory cost. To illustrate this, let us consider a simple Python example of building a list and return it[1].



```python
# firstn with list
def firstn(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums
```

Fig. 1: Firstn with List

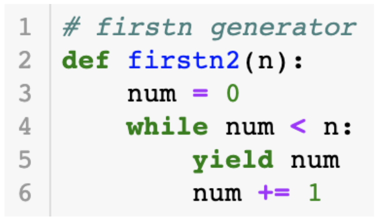
In figure 1, the function firstn return a full list with length n in memory. If n is really big number and each integer keeps 10 megabyte in memory, then we need to cost a lot of memory to extend our RAM.



```python
# firstn with generator pattern
class firstn1(object):
    def __init__(self, n):
        self.n = n
        self.num = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num + 1
        else:
            raise StopIteration()
        return cur
```

Fig. 2: Firstn with List

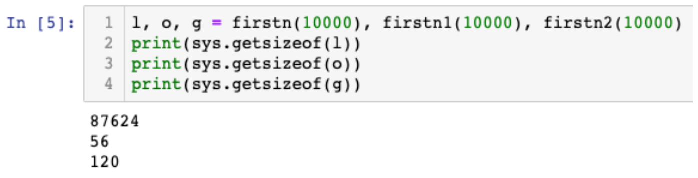
To save memory space, we can implement the firstn1 as an object with generator pattern, fig 2. Class firstn1 is iterable and it will perform as we expect. However, we need to write a bunch lines of code to implement it and the logic is expressed in a convoluted way.



```python
# firstn generator
def firstn2(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

Fig. 3: Firstn Generator

Python provides generator feature in each functions. In the figure 3, the keyword *yield* indicates that the function firstn2 is a generator that yields items in the iteration instead of returning a list. Compare the actually object sizes in these approaches with the same input 10k. The figure 4 shows that the generator has a huge advantage not only in memory efficiency but also in clear and natural logic.



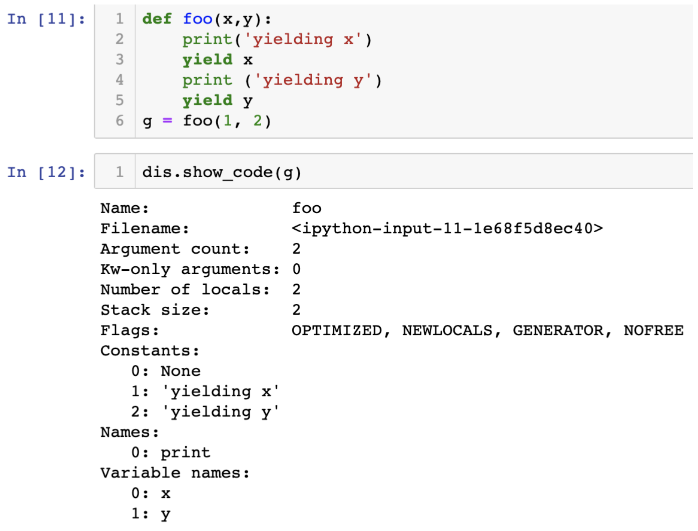
```
In [5]:  l, o, g = firstn(10000), firstn1(10000), firstn2(10000)
         print(sys.getsizeof(l))
         print(sys.getsizeof(o))
         print(sys.getsizeof(g))

87624
56
120
```

Fig. 4: Memory Sizes

1.2 Background

In this section, I will show necessary background about this generator functions. For ordinary functions, they have a single entry point and multiple exit points(return statements). When we call a function, the code runs from the first line of the function until it finds an exit point. After that, the function's stack of local variables are cleared and the corresponding memory reclaimed by the OS[2].

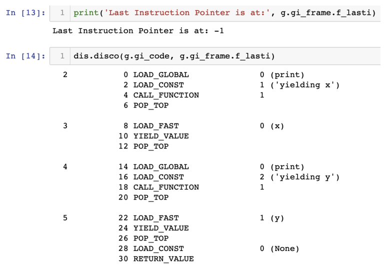
However, the generator function have multiple entry and exit points. The function's stack of local variables are allocated on heap memory instead of stack memory. Each *yield* statement will defines an exit point and a re-entry point in the same location. The generator function runs until a *yield* statement is encountered. At that point, the function is paused. And the flow of control is yielded to the caller of the generator function and then back to the re-entry point to resume the function.



```
In [11]:  def foo(x,y):
              print('yielding x')
              yield x
              print ('yielding y')
              yield y
          g = foo(1, 2)

In [12]:  dis.show_code(g)

Name:              foo
Filename:          <ipython-input-11-1e68f5d8ec40>
Argument count:    2
Kw-only arguments: 0
Number of locals:  2
Stack size:        2
Flags:             OPTIMIZED, NEWLOCALS, GENERATOR, NOFREE
Constants:
   0: None
   1: 'yielding x'
   2: 'yielding y'
Names:
   0: print
Variable names:
   0: x
   1: y
```

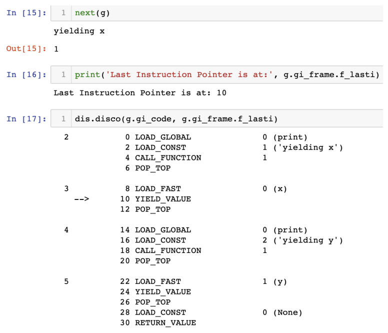
Fig. 5: Function *foo* with Yield

In the figure 5, variable *g* indicates the generator *foo(1, 2)*. When the CPython compiler finds the *yield* keyword in a function, it sets a GENERATOR flag. Then, the compiler returns a generator object which is iterable. If we apply *next* function with the generator, then it returns the values of variable *x, y* in the first two calls respectively as shown in Figure 6.



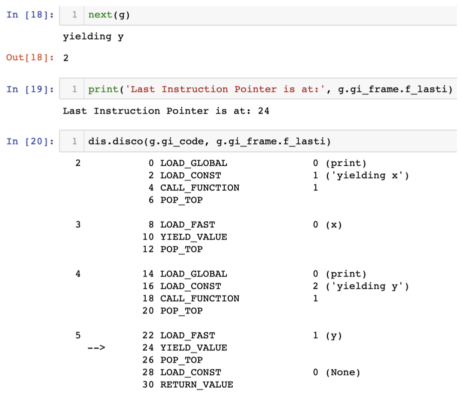
```
In [13]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
         Last Instruction Pointer is at: -1

In [14]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)
             10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5          22 LOAD_FAST                1 (y)
             24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(a) Before Next



```
In [15]: next(g)
         yielding x
Out[15]: 1

In [16]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
         Last Instruction Pointer is at: 10

In [17]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)
    -->      10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5          22 LOAD_FAST                1 (y)
             24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(b) First Next



```
In [18]: next(g)
         yielding y
Out[18]: 2

In [19]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
         Last Instruction Pointer is at: 24

In [20]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)
             10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5          22 LOAD_FAST                1 (y)
    -->      24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(c) Second Next

Fig. 6: Iterable Generator Foo

2 Approach

2.1 Exit Point and Re-entry Point

The key idea of the generator function is how to encounter and build exit point and re-entry point in the function. To solve this problem, the first step is to implement sequence function and we can write

```
r; r := succ(r)
```

as an abbreviation for

```
(\x:Unit. (r := succ(r)) (r)
```

Let's suppose the sequence is in a function. When it encounter a *yield* term as shown below, we need a exit point that can pause the function and return the content in the *yield* term and a re-entry point to resume the rest code of function.

```
yield r; r := succ(r)
```

To achieve this goal, I designed a naive *tseq* function shown as below that takes two terms *t1* and *t2*. If the *t1* is a *yield* term, then it returns a *exit* pair of terms (t1', (abs "_" Unit t2)), where t1' is the actual content returned at that exit point and (abs "_" Unit t2) is the remainder fo the function without reduction or execution.

```
Definition tseq t1 t2 :=
match t1 with
| gyield t1' => pair t1' (abs "_" Unit t2)
| _ => app (abs "_" Unit t2) t1
end.
```

By applying our *tseq* function, the output of the yield sequence shown above would be:

```
(r,  (\_:Unit. r := succ(r))
```

However, I implemented a sequence term *seq* in my project instead of the *tseq* function for easing *seq* concatenation.

2.2 Yield in Iteration

To make sure the *tseq* function would work in a iteration, more explicitly the second term of the returned *exit* pair should contain the rest iterations of a loop, we need to define the loop recursively with *fixed point*. A *while* loop can be defined as below, where p is a predicate function of input r.

```
while (p(r)) {
  yield r;
  r := succ(r)
}
```

Then we can define a *while* function with the *fixed point*.

```
def while (r) {
  if p(x) do
      yield r;
      r := succ(r);
      while(r)
}
```

In this way, there is a *exit* pair returned in each *while* iteration. The corresponding proof is provided for the *while* term in the operation semantic section.

2.3 Generator Feature

I will show how my generator feature works in my language. First, let's give an example about an ordinary function *F*. In line 2, the *x* is the number argument passing into the function and *x'* is the reference for *x*. In line 3 and 4, we increase and decrease the number by 1 in the reference *x'*. Then in line 5, we apply a *while* term constructed by a predicate function P and a body function B to *x'*. At last, we return the value of under the reference cell *x'* in line 6.

```
1 F = \x: Nat.
2     let x' = (ref x) in
3     x' := succ (deref x');
4     x' := pred (deref x');
5     while P B x';
6     !x'
```

The predicate and body functions in the *while* term are:

```
P = \x':(Ref Nat). (deref x') < 5
B = \x':(Ref Nat). x' := succ !x'
```

Next, let's see an example for a generator function. The generator *G* is similar to the function *F*. However, the generator returns a reference of a unit function with a sequence body instead of a pure sequence term in two main reasons:

1. A generator function will be turned to a iterable object by given an input.
2. A unit function can prevent the reduction process of the *seq* term in the *ref* term. Because I only want the *seq* term be reduced in the *next* term.

```
G = \x: Nat.
    let x' = (ref x) in
    ref ( \_: Unit.
          x' := succ (deref x');
          yield !x';
          x' := pred (deref x');
          yield !x';
          while P B x';
          yield !x')
```

The predicate and body functions in the *while* term are:

```
P = \x':(Ref Nat). (deref x') < 5
B = \x':(Ref Nat).
    yield !x';
    x' := succ !x'
```

I defined a *gen* term to construct a iterable generator object in type (Itr T) by taken a generator function and a corresponding input. To iterate each item returned at the *yield* exit point of a *gen* term, we use a *next* function shown as below.

```
let g = gen G x in
next(g);
next(g)
```

In the following sections, I provide the approaches including syntax, operation semantics, and typing of this generator feature.

3 Syntax

Abstract syntax is a structure used in the representation of text in computer languages, which are stored in a tree structure as an abstract syntax tree.

```
Syntax:

     t ::=                        Terms
       | ...                      (other terms same as before)
       | seq t1 t2                sequence
       | ref t                    reference
       | deref t                  dereference
       | assign t1 t2             assignment
       | loc nat                  reference cell location
       | let x=t1 in t2           let-binding
       | fix f                    fixed-point operator
       | while p b                while function
       | gen t1 t2                iterable generator
       | yeild t                  yeild term
       | next t                   next function

     T ::=                        Types
       | ...
       | (Itr T)                  Iterator type

     V ::=                        Values
       | ...
       | loc l                    reference cell location
       | yield t                  yeild term
```

Terms: The *seq* term is used to execute each terms in a order. The reference term is used to tracking variable value globally, so *ref, deref, assign, loc* are needed. The *let* can bind a name *x* to *t1* while reduce *t2*. The *fix* can be applied to a abstraction in order to return its *fixed point* for recursion. The *while* takes a predicate function and a body function to form a while loop function. The *gen* takes a generator function and a argument used to pass into the generator. The *yield* term is used to trigger the exit point and re-entry point in a *seq* term. The *next* takes a iterable term.

For the other required terms, I am not going to show more details here since they are similar to those terms in chapter *MoreStlc*. The *number* is required to count down the rest iterations. The *boolean* is required for conditional statement. The *list* is required for iteration. The *pair* is required for return pair values when the execution encounters a exit point in the generator. The fix is required for build *while* term.

Types: The $next$ function should only take iterable terms. So, I added a iterator type to identify the iterable generator. And other types like Arrow, Nat, Bool, List, Unit, and Prod are similar to the terms in chapter *MoreStlc*.

Values: Both *loc l* and *yield t* are considered as value terms without any further reduction.

4 Operational Semantics

The operational semantics of a programming language is used to interpret a valid program into sequences of computational steps. And these sequences are the meaning of the program. In this section, I will describe the operational semantics for the generator feature.

4.1 Sequence

When the *seq* term encounter a *yield* term(ST_Seq3), it returns *exit* pair containing the return value and the rest sequence. When the *seq* term encounter a non-*yield* term(ST_-Seq4), it execute *v1, v2* in order.

```
Reduction:

              t1 / st --> t1' / st'
        ---------------------------              (ST_Seq1)
        seq t1 t2 / st --> seq t1' t2 / st'

                 value v1
          t2 / st --> t2' / st'
        ---------------------------              (ST_Seq2)
        seq v1 t2 / st --> seq v1 t2' / st'

              value v1
              value v2
              yeild_tm v1
        ------------------------------         (ST_Seq3)
      seq v1 t2 / st
        --> pair v1 (abs "_" Unit v2) / st'

              value v1
              value v2
          not (yeild_tm v1)
        ------------------------------         (ST_Seq4)
        seq v1 t2 / st
        --> app(abs "_" Unit v2) v1 / st'
```

4.2 Reference

```
Reduction:

          ------------------------------         (ST_RefValue)
            ref v1 / st -->
```

```
                        loc |st| / st,v1

                        t / st --> t' / st'
                ----------------------------------   (ST_Ref)
                    ref t / st --> ref t' / st'

                      t / st --> t' / st
                ----------------------------------   (ST_Deref)
                      !t / st --> !t' / st'

                        l < |st|
                  ----------------------------      (ST_DerefLoc)
               !(loc l) / st --> lookup l st / st

                    t1 / st --> t1' / st
                ----------------------------------   (ST_Assign1)
                 t1 := t2 / st --> t1' := t2  / st

                  t2 / st --> t2' / st
              ---------------------------------     (ST_Assign2)
              t1 := t2 / st --> t1 := t2'  / st

                        l < |st|
                ----------------------------------   (ST_Assign)
                 loc l := v2 / st --> unit / [l:=v2]st
```

4.3 Let

Let bindings are syntactic sugar in choosing a standard *call-by-value* evaluation order. So the first term bind to variable *x* must be fully reduced before reduction of the body term. For its substitution as shown in below, if x equals y then $t2$ remains else substitute x with 2 in $t2$.

```
Substitution:
let x = 2 in
let y = t1 in t2

Reduction:

                  t1 / st --> t1' / st'
              ----------------------------------     (ST_Let1)
          let x=t1 in t2 / st --> let x=t1' in t2 / st'

                       value v1
                ----------------------------      (ST_LetValue)
          let x=v1 in t2  / st --> [x:=v1]t2  / st'
```

4.4 Fix

Lambda calculus is a universal model of computation which is well defined in Chapter *Stlc*. Then, can we define a general recursion with this. Let us define the *fixed point*

$Y = \lambda f.(\lambda g.f(g\ g))(\lambda g.f(g\ g))$. Then the recursion function f could be defined as follow:

$$\begin{aligned}
Yf &= \lambda f.((\lambda g.f(g\ g))(\lambda g.f(g\ g)))f \\
&= (\lambda g.f(g\ g))(\lambda g.f(g\ g)) \\
&= f((\lambda g.f(g\ g))(\lambda g.f(g\ g))) \\
&= f(\lambda f.((\lambda g.f(g\ g))(\lambda g.f(g\ g)))f) \\
&= f(Yf) \\
&= f(f(Yf)) \\
&= ...
\end{aligned}$$

```
Reduction:

                    t1  / st --> t1' / st'
               --------------------------------        (ST_Fix1)
                fix t1  / st --> fix t1' / st'

        ------------------------------------------    (ST_FixAbs)
           fix (\xf:T1.t2) / st -->
                  [xf:=fix (\xf:T1.t2)] t2  / st
```

4.5 While

Since *while* loop can be implemented in recursion as shown below, then we can form a *while* function either in inductive or a lambda calculus[3].

```
                              def $while(x)$ :
while $p(x)$ do                   if $p(x)$ do
    $x := b(x)$      $\iff$             return $while(b(x))$
                                  else
                                      return $x$
```

Suppose the x is a number then we can formalize *while* in:

$$\begin{aligned}
while &= \lambda x : Nat.\ \text{if } p(x) \text{ then } while(b(x)) \text{ else } x \\
&= \text{fix}\lambda f : Nat \to Nat.\lambda x : Nat.\ \text{if } p(x) \text{ then } f(b(x)) \text{ else } x \\
&= \lambda p : Nat \to bool.\lambda b : Nat \to Nat. \\
&\quad \text{fix } \lambda f : Nat \to Nat.\lambda x : Nat.\ \text{if } p(x) \text{ then } f(b(x)) \text{ else } x
\end{aligned}$$

Then, we can replace Nat type by any type T and the define the operation semantics of *while* function as follow.

```
Reduction:
```

```
           t1 / st --> t1' / st'
    ------------------------------------          (ST_While1)
    while t1 t2 / st --> while t1' t2 / st'

          value v1
          t2 / st --> t2' / st'
    ------------------------------------          (ST_While2)
    while v1 t2 / st --> while v1 t2' / st'

---------------------------------------------------- (ST_WhileFix)
  while (\x1:T. p) (\x2:T. b) / st
  --> tfix (\f:T->T.
          (\x:T. (if (\x1:T. p) x
                 then f ((\x2:T. b) x)
                 else x))) / st'
```

4.6 Generator

The *next* term takes a *gen* term. First, (ST_Gnext2, ST_Gnext3)the *next* de-reference the generator and execute the body sequence under the unit function in the reference cell. (ST_Gnext4)If it returns a *exit* pair, then update generator reference and return the value. (ST_Gnext5) If it returns only *yield* term, then assign the generator reference with unit and return value in the *yield* term.

```
Reduction:

           t1 / st --> t1' / st'
    --------------------------------------      (ST_Gen1)
    gen t1 t2 / st --> gen t1' t2 / st'

          value v1
      t2 / st --> t2' / st'
------------------------------------------      (ST_Gen2)
gen v1 t2 / st --> gen v1 t2' / st'


--------------------------------------------- (ST_Gen3)
 gen (\x:T. g) v / st
   --> ref (app (\x:T. g) v) / st'

          t / st --> t' / st'
    -------------------------------------      (ST_Gyeild)
    gyeild t / st --> gyeild t' / st'

          t / st --> t' / st'
    ---------------------------------          (ST_Gnext1)
    gnext t / st --> gnext t' / st'
```

```
          ——————————————————————————————————————   (ST_Gnext2)
       gnext (loc l) / st
         --> gnext (pair (loc l) (deref loc l)) / st


      ——————————————————————————————————————————   (ST_Gnext3)
        gnext (pair (loc l) (abs "_" Unit)) / st
    --> gnext (pair (loc l) (app (abs "_" Unit) unit)) / st


        ——————————————————————————————————————————   (ST_Gnext4)
          gnext (pair (loc l) (pair t1 t2)) / st
       --> seq (assign (loc l) (ref t2)) t1 / st


          ———————————————————————————————————————   (ST_Gnext5)
          gnext (pair (loc l) (gyeild t)) / st
          --> seq (assign (loc l) unit) t / st
```

5 Typing

The type system is used to assign a property called type to various terms in order to remove stuck terms. In my language, the types such as *Arrow*, *Nat*, *Bool*, *List*, *Unit*, and *Prod* are similar to the types in chapter *MoreStlc*. They are used to specify the types for lambda calculus, natural number, boolean, list, unit, and pair. I also added a new type *Itr T* to identify the iterable term like *gen*. The first term of the *next* function must be a iterable term.

5.1 Let

The *let* term has a type $T2$ if only if that 1) $t1$ has a type $T1$ and 2) $t2$ has a type $T2$ in the $Gamma$ where the x has type $T1$ in.

```
Typing:

                  Gamma |- t1 \in T1
                x |->T1; Gamma |- t2 \in T2
             ——————————————————————————————————   (T_Let)
               Gamma |- let x=t1 in t2 \in T2
```

5.2 Fix

The *fix* term has a type $T1$ if only if that $t1$ has a type $T1 \to T1$.

```
Typing:
```

```
          Gamma |-  t1  \in  T1  ->  T1
       ---------------------------------------   (T_Fix)
          Gamma |-  fix  t1  \in  T1
```

5.3 While

The *while* term has a type T if only if that the predicate function $t1$ has a type $T \to Bool$ and the body function $t2$ has a type $T \to T$.

```
Typing:

          Gamma |-  t1  \in  T -> Bool
          Gamma |-  t2  \in  T -> T
       ---------------------------------------   (T_While)
          Gamma |-  while  t1  t2  \in  T
```

5.4 Generator

The *gen* term has a type $(Itr\ T)$ if only if that the predicate function $t1$ has a type $T \to Bool$ and the body function $t2$ has a type $T \to T$. And the *next* term has a type $(T * T)$ if only if the $t1$ iterable with a type $(Itr\ T)$ and the body function $t2$ has a type T.

```
Typing:

          Gamma |-  t1  \in  T -> Bool
          Gamma |-  t2  \in  T -> T
       ---------------------------------------   (T_Gen)
        Gamma |-  gen  t1  t2  \in  (Itr  T)

          Gamma |-  t1  \in  (Itr  T)
          Gamma |-  t2  \in  T
       ---------------------------------------   (T_Next)
        Gamma |-  next  t1  t2  \in  (T * T)
```

References

1. AshishSingh: Generators. `https://wiki.python.org/moin/Generators` Accessed: 2018-03-08.
2. Abbas, S.K.: The magic behind python generator functions. `https://hackernoon.com/the-magic-behind-python-generator-functions-bc8eeea54220` Accessed: Aug 1, 2017.
3. chi (https://cs.stackexchange.com/users/43599/chi): λ-calculus, is there encoding of for or while? Computer Science Stack Exchange URL:https://cs.stackexchange.com/q/88637 (version: 2018-03-01).