Homework 3
COSE312, Spring 2023

Hakjoo Oh

Due: 04/30, 23:59

The goal of this assignment is to implement a translator that converts a subset of Python into a low-level language. The template code is available at

`https://github.com/kupl-courses/COSE312-2023spring/tree/main/homework/hw3`

Source Language The source language, SPY (Small Python), is defined as follows:

P	$\rightarrow$	S^*	
S	$\rightarrow$	`def` $f(x^*)$ S^*	function definition
	$\mid$	`return`	function return without value
	$\mid$	`return` E	function return with value
	$\mid$	$E^* = E$	assignment
	$\mid$	E *binop*$=$ E	augmented assignment
	$\mid$	`for` E E S^*	for loop
	$\mid$	`while` E S^*	while loop
	$\mid$	`if` E S^* S^*	conditinoal statement
	$\mid$	assert E	assert statement
	$\mid$	`break`	break statement
	$\mid$	`continue`	continue statement
	$\mid$	`pass`	pass statement
E	$\rightarrow$	*boolop* E^*	boolean operator
	$\mid$	E *binop* E	binary operator
	$\mid$	*uop* E	unary operator
	$\mid$	E `if` E else E	conditional expression
	$\mid$	`[` E `(for` E `in` E `(if` E`)`*`)`* `]`	list comprehension
	$\mid$	E *cmpop* E	comparison operator
	$\mid$	E E^*	function call (including built-in functions)
	$\mid$	n	integer constant
	$\mid$	s	string constant
	$\mid$	`True` $\mid$ `False`	boolean constant
	$\mid$	`None`	none value
	$\mid$	$E.x$	attribute
	$\mid$	E`[`E`]`	subscript
	$\mid$	x	variable
	$\mid$	`[`E^*`]`	list
	$\mid$	`(`E^*`)`	tuple
	$\mid$	`lambda` x^* E	lambda function
boolop	$\rightarrow$	`&&` $\mid$ `||`	
binop	$\rightarrow$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `**`	
cmpop	$\rightarrow$	`>` $\mid$ `>=` $\mid$ `<` $\mid$ `<=` $\mid$ `==` $\mid$ `!=`	
uop	$\rightarrow$	`+` $\mid$ `-` $\mid$ `!`	

In OCaml datatype,

```
type identifier = string
type constant =
  | CInt of int
  | CString of string
  | CBool of bool
  | CNone

type program = stmt list

and stmt =
  | FunctionDef of identifier * identifier list * stmt list
  | Return of expr option
  | Assign of expr list * expr
  | AugAssign of expr * operator * expr
  | For of expr * expr * stmt list
  | While of expr * stmt list
  | If of expr * stmt list * stmt list
  | Assert of expr
  | Expr of expr
  | Break
  | Continue
  | Pass

and expr =
  | BoolOp of boolop * expr list
  | BinOp of expr * operator * expr
  | UnaryOp of unaryop * expr
  | IfExp of expr * expr * expr
  | ListComp of expr * comprehension list
  | Compare of expr * cmpop * expr
  | Call of expr * expr list
  | Constant of constant
  | Attribute of expr * identifier
  | Subscript of expr * expr
  | Name of identifier
  | List of expr list
  | Tuple of expr list
  | Lambda of identifier list * expr

and boolop = And | Or
and comprehension = expr * expr * expr list
and operator = Add | Sub | Mult | Div | Mod | Pow
and unaryop = Not | UAdd | USub
and cmpop = Eq | NotEq | Lt | LtE | Gt | GtE
```

SPY supports the following built-in functions and methods in Python: `print`, `input`, `len`, `int`, `range`, `isinstance`, `append`.

Target Language The target language, SPVM (Small Python Virtual Machine), is defined as follows:

$$x, y, z, f \in \mathit{Id}, \quad n \in \mathit{Integer}, \quad s \in \mathit{String}, \quad l \in \mathit{Label}$$

$$
\begin{array}{rcll}
P & \rightarrow & \mathit{LabeledInstruction}^* & \\
\mathit{LabeledInstruction} & \rightarrow & \mathit{Label} \times \mathit{Instruction} & \\
\mathit{Instruction} & \rightarrow & \mathsf{skip} & \\
 & | & \mathsf{def}(f, x, \mathit{linstrs}) & \text{function definition} \\
 & | & x = \mathsf{call}(f, y) & \text{function call} \\
 & | & \mathtt{return}\ x & \text{function return} \\
 & | & x = \mathsf{range}(y, z) & \text{range} \\
 & | & x = [] & \text{empty list} \\
 & | & \mathsf{append}(x, y) & \text{list append} \\
 & | & \mathsf{insert}(x, y) & \text{list insert} \\
 & | & \mathsf{reverse}(x) & \text{list reverse} \\
 & | & x = () & \text{empty tuple} \\
 & | & \mathsf{tupinsert}(x, y) & \text{typle insert} \\
 & | & x = y[z] & \text{load} \\
 & | & x[y] = z & \text{store} \\
 & | & x = \mathsf{len}(y) & \text{length} \\
 & | & x = y\ \mathit{bop}\ z & \text{binary operator} \\
 & | & x = y\ \mathit{bop}\ n & \text{binary operator} \\
 & | & x = \mathit{uop}\ y & \text{unary operator} \\
 & | & x = y & \text{copy} \\
 & | & x = n & \text{integer assignment} \\
 & | & x = s & \text{string assignment} \\
 & | & x = \mathsf{none} & \text{none assignment} \\
 & | & \mathsf{goto}\ l & \text{unconditional branch} \\
 & | & \mathsf{if}\ x\ \mathsf{goto}\ l & \text{conditional branch} \\
 & | & \mathsf{iffalse}\ x\ \mathsf{goto}\ l & \text{conditional branch} \\
 & | & \mathsf{read}\ x & \text{read} \\
 & | & \mathsf{write}\ x & \text{write} \\
 & | & x = \mathsf{int}(y) & \text{int of string} \\
 & | & x = \mathsf{isinstance}(y, s) & \text{isintance} \\
 & | & \mathsf{assert}\ x & \text{assertion} \\
 & | & \mathsf{halt} & \\
\mathit{bop} & \rightarrow & \mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \mathtt{/} \mid \mathtt{\%} \mid \mathtt{**} \mid & \\
 & & \mathtt{>} \mid \mathtt{>=} \mid \mathtt{<} \mid \mathtt{<=} \mid \mathtt{==} \mid \mathtt{!=} \mid \mathtt{\&\&} \mid \mathtt{||} & \\
\mathit{uop} & \rightarrow & \mathtt{+} \mid \mathtt{-} \mid \mathtt{!} &
\end{array}
$$

In OCaml datatype:

```
type program = linstr list
and linstr = label * instr                (* labeled instruction *)
and instr =
  | SKIP
  | FUNC_DEF of id * id list * linstr list  (* def f(args): body *)
  | CALL of id * id * id list              (* x = call(f, args )*)
  | RETURN of id                           (* return x *)
  | RANGE of id * id * id                  (* x = range(lo, hi) *)
  | LIST_EMPTY of id                       (* x = [] *)
  | LIST_APPEND of id * id                 (* append(x,y) *)
  | LIST_INSERT of id * id                 (* insert(x,y) *)
  | LIST_REV of id                         (* reverse(x) *)
  | TUPLE_EMPTY of id                      (* x = () *)
  | TUPLE_INSERT of id * id                (* tupinsert(x,y) *)
  | ITER_LOAD of id * id * id              (* x = a[y] *)
```

```
  | ITER_STORE of id * id * id              (* a[x] = y *)
  | ITER_LENGTH of id * id                  (* x = len(y) *)
  | ASSIGNV of id * bop * id * id           (* x = y bop z *)
  | ASSIGNC of id * bop * id * int          (* x = y bop n *)
  | ASSIGNU of id * uop * id                (* x = uop y *)
  | COPY of id * id                         (* x = y *)
  | COPYC of id * int                       (* x = n *)
  | COPYS of id * string                    (* x = s *)
  | COPYN of id                             (* x = None *)
  | UJUMP of label                          (* goto L *)
  | CJUMP of id * label                     (* if x goto L *)
  | CJUMPF of id * label                    (* ifFalse x goto L *)
  | READ of id                              (* read x *)
  | WRITE of id                             (* write x *)
  | INT_OF_STR of id * id                   (* x = int(y) *)
  | IS_INSTANCE of id * id * string         (* x = isinstance(y, typ) *)
  | ASSERT of id                            (* assert x *)
  | HALT
and id = string
and label = int
and bop = ADD | SUB | MUL | DIV | MOD | POW |
          LT | LE | GT | GE | EQ | NEQ | AND | OR
and uop = UPLUS | UMINUS | NOT
```

The semantics is defined as a state transition system, $(State, \Rightarrow, s_0)$, where $State$ denotes the set of program states, $(\Rightarrow) \subseteq State \times State$ the transition relation, and s_0 the initial state. We first define the program states:

$$
\begin{array}{rcl}
a \in Addr & = & \text{Memory Addresses} \\
v \in Value & = & \{\mathsf{none}\} + Integer + String + Addr + Tuple + List + Closure \\
(v_1, v_2, \ldots) \in Tuple & = & Value^* \\
\langle v_1, v_2, \ldots \rangle \in List & = & Value^* \\
c \in Closure & = & Id \times Id \times LabeledInstruction^* \\
m \in Mem & = & Addr \to Value \\
e \in Env & = & Id \to Addr \\
\sigma \in CallStack & = & StackFrame^* \\
(f, l_{ret}, a_{ret}, e) \in StackFrame & = & Id \times Label \times Addr \times Env \\
(l, \sigma, m) \in State & = & Label \times CallStack \times Mem
\end{array}
$$

A state (l, σ, m) includes a program counter l, a call stack σ, and a memory m. A call stack is a sequence of stack frames, where a stack frame (f, l_{ret}, a_{ret}, e) consists of the name f of the called function, the return label l_{ret}, the return address a_{ret}, and the environment e of the function. The initial state s_0 is

$$s_0 = (l_0, \langle(dummy, dummy, dummy, \emptyset)\rangle, \emptyset)$$

where l_0 denotes the first instruction of the program.

The following auxiliary functions will be used by the transition relation:

$$cmd(l) = \text{the command at label } l$$

$$succ(l) = \text{the successor label of } l$$

$$\sigma(x) = \begin{cases} e(x) & \sigma = (f, l_{ret}, a_{ret}, e) :: \sigma', x \in \mathsf{Dom}(e) \\ \sigma'(x) & \sigma = (f, l_{ret}, a_{ret}, e) :: \sigma', x \notin \mathsf{Dom}(e) \\ \mathsf{error} & \sigma = \epsilon \end{cases}$$

$$\mathsf{alloc}(m) = (a, m[a \mapsto 0]) \text{ where } a \notin \mathsf{Dom}(m)$$

$$\mathsf{lookup}(x, (\sigma, m)) = \begin{cases} (e(x), (\sigma, m)) & x \in \mathsf{Dom}(e) \\ (a, ((f, l_{ret}, a_{ret}, e[x \mapsto a]) :: \sigma', m')) & x \notin \mathsf{Dom}(e), (a, m') = \mathsf{alloc}(m) \end{cases}$$
$$\text{where } \sigma = (f, l_{ret}, a_{ret}, e) :: \sigma'$$

Now we are ready to define the transition relation $(\Rightarrow) \subseteq State \times State$. Given a state (l, σ, m), the next state is defined depending on $cmd(l)$:

- $cmd(l) = \mathsf{skip}$:

$$(l, \sigma, m) \Rightarrow (succ(l), \sigma, m)$$

- $\mathsf{def}(f, x, linstrs)$:

$$\frac{(a', m') = \mathsf{alloc}(m)}{(l, (f', l_{ret}, a_{ret}, e) :: \sigma', m) \Rightarrow (succ(l), (f', l_{ret}, a_{ret}, e[f \mapsto a']) :: \sigma', m'[a' \mapsto (f, x, linstrs)])}$$

- $x = \mathsf{call}(f, y)$:

$$\frac{(f'', x', (l', \_) :: \_) = m(\sigma(f)) \quad v = m(\sigma(y)) \quad (a'_{ret}, m') = \mathsf{alloc}(m) \quad (a_{x'}, m'') = \mathsf{alloc}(m')}{\begin{array}{l}(l, (f', l_{ret}, a_{ret}, e) :: \sigma', m) \\ \quad \Rightarrow (l', (f'', succ(l), a'_{ret}, [x' \mapsto a_{x'}]) :: (f', l_{ret}, a_{ret}, e[x \mapsto a'_{ret}]) :: \sigma', m''[a_{x'} \mapsto v])\end{array}}$$

- `return` x:

$$(l, (f, l_{ret}, a_{ret}, e) :: \sigma', m) \Rightarrow (l_{ret}, \sigma', m[a_{ret} \mapsto m(\sigma(x))])$$

- $x = \mathsf{range}(y, z)$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad (a', m'') = \mathsf{alloc}(m') \quad n_1 = m(\sigma(y)) \quad n_2 = m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m''[a_x \mapsto a', a' \mapsto \langle n_1, \ldots, n_2 - 1\rangle])}$$

- $x = []$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad (a', m'') = \mathsf{alloc}(m')}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m''[a_x \mapsto a', a' \mapsto \langle\rangle])}$$

- $\mathsf{append}(x, y)$:

$$\frac{a = m(\sigma(x)) \quad \langle v_1, \ldots, v_k\rangle = m(a)}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m[a \mapsto \langle v_1, \ldots, v_k, m(\sigma(y))\rangle])}$$

- $\mathsf{insert}(x, y)$:

$$\frac{a = m(\sigma(x)) \quad \langle v_1, \ldots, v_k\rangle = m(a)}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m[a \mapsto \langle m(\sigma(y)), v_1, \ldots, v_k\rangle])}$$

- $\mathsf{reverse}(x)$:

$$\frac{a = m(\sigma(x)) \quad \langle v_1, \ldots, v_k\rangle = m(a)}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m[a \mapsto \langle v_k, \ldots, v_1\rangle])}$$

- $x = ()$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto ()])}$$

- $\mathsf{tupinsert}(x, y)$:

$$\frac{(v_1, \ldots, v_k) = m(\sigma(x)) \quad v_y = m(\sigma(y))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m[\sigma(x) \mapsto (v_y, v_1, \ldots, v_k)])}$$

- $x = y[z]$:

$$\frac{a = m(\sigma(y)) \quad \langle v_1, \ldots, v_k \rangle = m(a) \quad (a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad n = m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto v_n])}$$

$$\frac{s = m(\sigma(y)) \quad (a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad n = m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto s_n])}$$

$$\frac{(v_1, \ldots, v_k) = m(\sigma(y)) \quad (a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad n = m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto v_n])}$$

- $x[y] = z$:

$$\frac{a = m(\sigma(x)) \quad n = m(\sigma(y)) \quad \langle v_1, \ldots, v_n, \ldots, v_k \rangle = m(a) \quad v'_n = m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m[a \mapsto \langle v_1, \ldots, v'_n, \ldots, v_k \rangle])}$$

- $x = \mathsf{len}(y)$:

$$\frac{a = m(\sigma(y)) \quad \langle v_1, \ldots, v_k \rangle = m(a) \quad (a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto k])}$$

$$\frac{s = m(\sigma(y)) \quad (a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto |s|])}$$

- $x = y \ bop \ z$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad v_x = m(\sigma(y)) \ bop \ m(\sigma(z))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto v_x])}$$

- $x = y \ bop \ n$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad v_x = m(\sigma(y)) \ bop \ n}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto v_x])}$$

- $x = uop \ y$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad v_x = uop \ m(\sigma(y))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto v_x])}$$

- $x = y$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto m(\sigma(y))])}$$

- $x = n$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto n])}$$

- $x = s$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto s])}$$

- $x = \mathsf{none}$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto \mathsf{none}])}$$

- goto l':

$$(l, \sigma, m) \Rightarrow (l', \sigma, m)$$

- if x goto l':

$$\frac{n = m(\sigma(x)) \quad n \neq 0}{(l, \sigma, m) \Rightarrow (l', \sigma, m)} \qquad \frac{n = m(\sigma(x)) \quad n = 0}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m)}$$

- iffalse x goto l':

$$\frac{n = m(\sigma(x)) \quad n = 0}{(l, \sigma, m) \Rightarrow (l', \sigma, m)} \qquad \frac{n = m(\sigma(x)) \quad n \neq 0}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m)}$$

- read x:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad s_x \text{ is the input string}}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto s_x])}$$

- write x:

$$(l, \sigma, m) \Rightarrow (succ(l), \sigma, m)$$

- $x = \mathsf{int}(y)$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad s = m(\sigma(y)) \quad n = \mathsf{int\_of\_str}(s)}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto n])}$$

- $x = \mathsf{isinstance}(y, s)$:

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad n = m(\sigma(y)) \quad s = \text{“int”}}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto 1])}$$

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m)) \quad a = m(\sigma(y)) \quad \langle v_1, \ldots, v_k \rangle = m(a) \quad s = \text{“list”}}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto 1])}$$

$$\frac{(a_x, (\sigma', m')) = \mathsf{lookup}(x, (\sigma, m))}{(l, \sigma, m) \Rightarrow (succ(l), \sigma', m'[a_x \mapsto 0])}$$

- assert x:

$$\frac{m(\sigma(x)) \neq 0}{(l, \sigma, m) \Rightarrow (succ(l), \sigma, m)}$$

- halt:

$$(l, \sigma, m) \not\Rightarrow$$

Translator Your job is to implement the function:

```
translate :  Spy.program -> Spvm.program
```

which takes a SPY program and produces a semantically-equivalent SPVM program. A frontend, which parses a Python program and translates it into SPY, as well as the interpreter for SPVM are provided, so you can execute a SPY program written in Python via translation into SPVM.

For example, the SPY program

```
def fact(n):
  i = 1
  r = 1
  while i <= n:
      r *= i
      i += 1
  return r

def factorial(n): return fact(n)

print(factorial(10))
```

is translated into the SPVM program

```
24 : def fact(n)
     3 : .t1 = 1
     4 : i = .t1
     5 : .t2 = 1
     6 : r = .t2
     7 : SKIP
     9 : .t4 = i
    10 : .t5 = n
    11 : .t3 = .t4 <= .t5
    21 : iffalse .t3 goto 8
    12 : .t7 = r
    13 : .t8 = i
    14 : .t6 = .t7 * .t8
    15 : r = .t6
    16 : .t10 = i
    17 : .t11 = 1
    18 : .t9 = .t10 + .t11
    19 : i = .t9
    20 : goto 7
     8 : SKIP
    22 : .t12 = r
    23 : return .t12

29 : def factorial(n)
    25 : .t14 = fact
    26 : .t15 = n
    27 : .t13 := call(.t14, (.t15))
    28 : return .t13

30 : .t18 = factorial
31 : .t19 = 10
32 : .t17 := call(.t18, (.t19))
33 : .t20 = " "
35 : write .t17
34 : write .t20
36 : .t21 = "\n"
37 : write .t21
38 : .t16 = None
2 : HALT
```

which is executed by the SPVM interpreter to obtain the result:

```
3628800
The number of instructions executed : 168
```