

Homework 3

COSE312, Spring 2026

Hakjoo Oh

Due: 4/12 23:59

Problem 1 Implement a compiler that translates **S** to **T** and **G**. The source language **S** was defined in class as follows.

```
program → block
block   → decls stmts
decls  → decls decl |  $\epsilon$ 
decl   → type x
type   → int | int[n]
stmts  → stmts stmt |  $\epsilon$ 

stmt   → lv = e
          | if e stmt stmt
          | while e stmt
          | do stmt while e
          | read x
          | print e
          | block

lv     → x | x[e]

e       → n                integer
          | lv              l-value
          | e+e | e-e | e*e | e/e | -e    airthmetic operation
          | e==e | e<e | e<=e | e>e | e>=e conditional operation
          | !e | e||e | e&&e           boolean operation
```

In OCaml (**s.ml**):

```
type program = block
and block = decls * stmts
and decls = decl list
```

```

and decl = typ * id
and typ  = TINT | TARR of int
and stmts = stmt list
and id   = string
and stmt = ASSIGN of lv * exp
          | IF of exp * stmt * stmt
          | WHILE of exp * stmt
          | DOWHILE of stmt * exp
          | READ of id | PRINT of exp
          | BLOCK of block
and lv   = ID of id | ARR of id * exp
and exp  = NUM of int | LV of lv | ADD of exp * exp
          | SUB of exp * exp | MUL of exp * exp
          | DIV of exp * exp | MINUS of exp
          | NOT of exp | LT of exp * exp
          | LE of exp * exp | GT of exp * exp
          | GE of exp * exp | EQ of exp * exp
          | AND of exp * exp | OR of exp * exp

```

The source language T:

$$\begin{aligned}
\text{program} &\rightarrow \text{LabeledInstruction}^* \\
\text{LabeledInstruction} &\rightarrow \text{Label} \times \text{Instruction} \\
\text{Instruction} &\rightarrow \text{skip} \\
&| x = \text{alloc}(n) \\
&| x = y \text{ bop } z \\
&| x = y \text{ bop } n \\
&| x = \text{uop } y \\
&| x = y \\
&| x = n \\
&| \text{goto } L \\
&| \text{if } x \text{ goto } L \\
&| \text{ifFalse } x \text{ goto } L \\
&| x = y[i] \\
&| x[i] = y \\
&| \text{read } x \\
&| \text{write } x \\
\text{bop} &\rightarrow + | - | * | / | > | >= | < | <= | == | \&\& | || \\
\text{uop} &\rightarrow - | !
\end{aligned}$$

In OCaml (t.ml):

```

type program = linstr list
and linstr = label * instr
and instr =
  | SKIP
  | ALLOC of var * int (* x = alloc(n) *)
  | ASSIGNV of var * bop * var * var (* x = y bop z *)
  | ASSIGNC of var * bop * var * int (* x = y bop n *)
  | ASSIGNU of var * uop * var (* x = uop y *)
  | COPY of var * var (* x = y *)
  | COPYC of var * int (* x = n *)
  | UJUMP of label (* goto L *)
  | CJUMP of var * label (* if x goto L *)
  | CJUMPF of var * label (* ifFalse x goto L *)
  | LOAD of var * arr (* x = a[i] *)
  | STORE of arr * var (* a[i] = x *)
  | READ of var (* read x *)
  | WRITE of var (* write x *)
  | HALT
and var = string
and label = int
and arr = var * var
and bop = ADD | SUB | MUL | DIV | LT | LE | GT | GE | EQ | AND | OR
and uop = MINUS | NOT

```

G's control-flow graph consists of nodes that include the commands:

$$c \rightarrow x = \text{alloc}(n) \mid lv = e \mid \text{assume}(e) \mid \text{skip} \mid \text{read } x \mid \text{print } e$$

In OCaml (g.ml):

```

module type Node = sig
  type instr =
    | I_alloc of id * int
    | I_assign of lv * exp
    | I_assume of exp
    | I_skip
    | I_read of id
    | I_print of exp
  type t
  val create_alloc : id -> int -> t
  val create_assign : lv -> exp -> t
  val create_assume : exp -> t
  val create_skip : unit -> t

```

```

val create_read : id -> t
val create_print : exp -> t
val get_nodeid : t -> int
val get_instr : t -> instr
val to_string : t -> string
val compare : t -> t -> int
end

module type Cfg = sig
  type t
  val empty : t
  val nodesof : t -> Node.t list
  val succs : Node.t -> t -> NodeSet.t
  val preds : Node.t -> t -> NodeSet.t
  val add_node : Node.t -> t -> t
  val add_nodes : Node.t list -> t -> t
  val add_edge : Node.t -> Node.t -> t -> t
  val add_loophead : Node.t -> t -> t
  val is_loophead : Node.t -> t -> bool
  val get_entry : t -> Node.t
  val get_exit : t -> Node.t
  val print : t -> unit
  val dot : t -> unit
end

```

Implement the two functions in `translator.ml`. Submit `translator.ml` only.

1. The function that translates S to T:

```
s2t : S.program -> T.program
```

2. The function that translates S to G:

```
s2cfg : S.program -> Cfg.t
```

Examples:

- `tests/t1.s`:

```

$ dune exec -- ./main.exe tests/t1.s
== S ==
{
  int x;

```

```

x = -1;
if !x {
  print -1;
}
else {
  print 2;
}
}
== executing S ==
2
== translating S to CFG ==
cfg has been saved in cfg.dot
== executing CFG ==
2
== translating S to T ==
0 : x = 0
0 : .t2 = 1
0 : .t1 = -.t2
0 : x = .t1
0 : .t4 = x
0 : .t3 = !.t4
0 : if .t3 goto 2
0 : goto 3
2 : SKIP
0 : .t6 = 1
0 : .t5 = -.t6
0 : write .t5
0 : goto 4
3 : SKIP
0 : .t7 = 2
0 : write .t7
0 : goto 4
4 : SKIP
0 : HALT
== executing T ==
2
The number of instructions executed : 13

```

- tests/loop1.s:

```

$ dune exec -- ./main.exe tests/loop1.s
== S ==

```

```

{
  int x;
  int y;
  int n;
  int i;
  int[10] fib;
  x = 0;
  y = 1;
  i = 0;
  n = 10;
  while i < n {
    x = y;
    y = x + y;
    fib[i] = y;
    i = i + 1;
  }

  i = 0;
  while i < n {
    print fib[i];
    i = i + 1;
  }
}

== executing S ==
2
4
8
16
32
64
128
256
512
1024

== translating S to CFG ==
cfg has been saved in cfg.dot
== executing CFG ==
2
4
8
16

```

```

32
64
128
256
512
1024
== translating S to T ==
0 : x = 0
0 : y = 0
0 : n = 0
0 : i = 0
0 : fib = alloc (10)
0 : .t1 = 0
0 : x = .t1
0 : .t2 = 1
0 : y = .t2
0 : .t3 = 0
0 : i = .t3
0 : .t4 = 10
0 : n = .t4
2 : SKIP
0 : .t6 = i
0 : .t7 = n
0 : .t5 = .t6 < .t7
0 : iffalse .t5 goto 3
0 : .t8 = y
0 : x = .t8
0 : .t10 = x
0 : .t11 = y
0 : .t9 = .t10 + .t11
0 : y = .t9
0 : .t12 = i
0 : .t13 = y
0 : fib[.t12] = .t13
0 : .t15 = i
0 : .t16 = 1
0 : .t14 = .t15 + .t16
0 : i = .t14
0 : goto 2
3 : SKIP
0 : .t17 = 0

```

```
0 : i = .t17
4 : SKIP
0 : .t19 = i
0 : .t20 = n
0 : .t18 = .t19 < .t20
0 : iffalse .t18 goto 5
0 : .t22 = i
0 : .t21 = fib[.t22]
0 : write .t21
0 : .t24 = i
0 : .t25 = 1
0 : .t23 = .t24 + .t25
0 : i = .t23
0 : goto 4
5 : SKIP
0 : HALT
== executing T ==
2
4
8
16
32
64
128
256
512
1024
The number of instructions executed : 347
```

You can visualize the generated control-flow graph as follows:

```
$ dot -Tpng cfg.dot > cfg.png
```

CFG for the second example:

