Homework 4
COSE312, Spring 2025

Hakjoo Oh

Due: 5/18 23:59

Problem 1 이번 숙제에서는 출발언어 `S`를 수업에서 정의한 도착언어 `T`와 `G`로 변환하는 컴파일러를 구현해 봅시다. 출발 언어(source language) `S`의 문법과 의미는 수업시간에 정의한 바와 같습니다.

program	→	*block*	
block	→	*decls stmts*	
decls	→	*decls decl* | ϵ	
decl	→	*type* x	
type	→	`int` | `int[`n`]`	
stmts	→	*stmts stmt* | ϵ	
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

OCaml 자료형으로는 아래와 같이 정의됩니다 (`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
```

도착 언어(target language) T는 다음과 같습니다.

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

OCaml 자료형으로 아래와 같이 정의됩니다 (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
```

G의 실행 흐름 그래프(Control-flow graph)는 아래의 명령어들을 포함하는 노드들로 구성됩니다.

$$c \to x = alloc(n) \mid lv = e \mid assume(e) \mid skip \mid read\ x \mid print\ e$$

OCaml로 아래와 같이 정의할 수 있습니다 (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
```

translator.ml내에 있는 아래 두 함수를 작성하세요. (translator.ml 파일만 제출합니다)

1. 주어진 S 프로그램을 동일한 의미를 가지는 T 프로그램으로 변환하는 함수:

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

2. 주어진 S 프로그램을 동일한 의미를 가지는 G 프로그램으로 변환하는 함수:

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

실행 예:

- 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
```

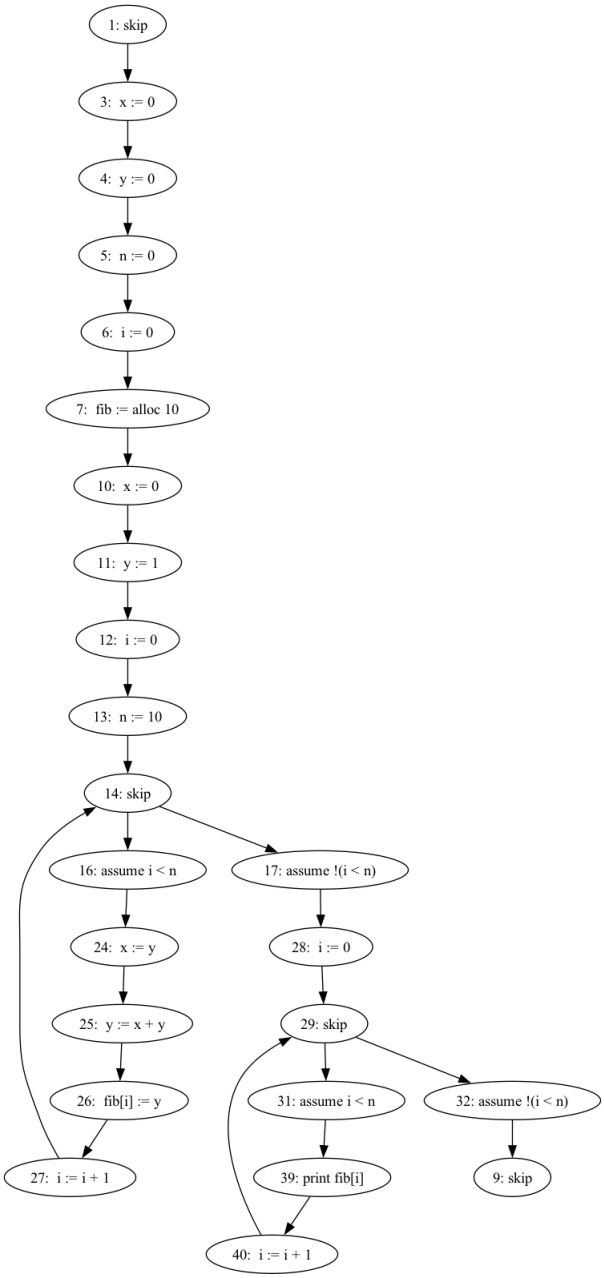
```
16
32
64
128
256
512
1024
== translating S to T ==
0 : x = 0
0 : y = 0
0 : n = 0
0 : i = 0
0 : fib = alloc (10)
0 : .t1 = 0
0 : x = .t1
0 : .t2 = 1
0 : y = .t2
0 : .t3 = 0
0 : i = .t3
0 : .t4 = 10
0 : n = .t4
2 : SKIP
0 : .t6 = i
0 : .t7 = n
0 : .t5 = .t6 < .t7
0 : iffalse .t5 goto 3
0 : .t8 = y
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
```

생성된 실행 흐름 그래프는 아래와 같이 이미지 파일로 변환하여 확인할 수 있습니다.

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

예를 들어, 두 번째 예제의 경우 아래와 같은 그래프가 생성됩니다.


1: skip
3: x := 0
4: y := 0
5: n := 0
6: i := 0
7: fib := alloc 10
10: x := 0
11: y := 1
12: i := 0
13: n := 10
14: skip
16: assume i < n
17: assume !(i < n)
24: x := y
28: i := 0
25: y := x + y
29: skip
26: fib[i] := y
31: assume i < n
32: assume !(i < n)
27: i := i + 1
39: print fib[i]
9: skip
40: i := i + 1