

《编译原理》第四次书面作业

截止日期：2024 年 12 月 10 日

若发现问题，或有任何想法（改进题目、调整任务量等等），请及时联系助教

Q1. 参考讲义中进行语句翻译的 L-翻译模式片断及所用到的语义函数。

1. 若在基础文法中增加产生式 $E \rightarrow E \bar{\wedge} E$ ，其中与非运算符 $\bar{\wedge}$ 定义为 $P \bar{\wedge} Q \equiv \neg(P \wedge Q)$ ，试给出该产生式的语义动作集合。请利用短路运算进行翻译。
2. 若在基础文法中增加产生式 $S \rightarrow \text{repeat } S \text{ until } E$ ，试给出该产生式的语义动作集合。

参考答案：

1.

$$\begin{aligned} E \rightarrow \{ & E1.\text{false} := E.\text{true}; E1.\text{true} := \text{newlabel} \} E_1 \bar{\wedge} \\ & \{ E2.\text{false} := E.\text{true}; E2.\text{true} := E.\text{false} \} E_2 \\ & \{ E.\text{code} := E1.\text{code} \parallel \text{gen}(E1.\text{true} \text{ ':'}) \parallel E2.\text{code} \} \end{aligned}$$

2.

$$\begin{aligned} S \rightarrow \text{repeat } \{ & S1.\text{next} := \text{newlabel} \} S_1 \text{ until} \\ & \{ E.\text{true} = S.\text{next}; E.\text{false} = \text{newlabel} \} E \\ & \{ S.\text{code} := \text{gen}(E.\text{false} \text{ ':'}) \parallel S1.\text{code} \parallel \text{gen}(S.\text{next} \text{ ':'}) \parallel E.\text{code} \} \end{aligned}$$

Q2. 考虑一个简单的栈式虚拟机。该虚拟机维护一个存放整数的栈，并支持如下3条指令：

- Push n ：将整数 n 压栈；
- Plus：弹出栈顶元素 n_1 和次栈顶元素 n_2 ，计算 $n_1 + n_2$ 的值，把结果压栈；
- Minus：弹出栈顶元素 n_1 和次栈顶元素 n_2 ，计算 $n_1 - n_2$ 的值，把结果压栈。

一条或多条指令构成一个指令序列。初始状态下，虚拟机的栈为空。给定一个仅包含加法和减法的算术表达式语言：

$$A \rightarrow A + A \mid A - A \mid (A) \mid \underline{int}$$

终结符 \underline{int} 表示一个整数，用 $\underline{int}.val$ 取得语法符号对应的语义值。任何一个算术表达式都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，栈中仅含一个元素，且它恰好为表达式的值。简单起见，我们用 “||” 来拼接两个指令序列。例如，算术表达式 $1 + 2 - 3$ 可翻译成指令序列

Push 3 || Push 2 || Push 1 || Plus || Minus

执行完成后，栈顶元素为 0。

1. 上述翻译过程可描述成如下S-翻译模式，其中综合属性 $A.instr$ 表示 A 对应的指令序列。请补全空缺的部分。

$$\begin{aligned} A &\rightarrow A_1 + A_2 \{ A.instr := \dots \} \\ A &\rightarrow A_1 - A_2 \{ A.instr := \dots \} \\ A &\rightarrow (A) \{ A.instr := A_1.instr \} \\ A &\rightarrow \underline{int} \{ A.instr := \text{Push } \underline{int}.val \} \end{aligned}$$

2. 给上述虚拟机新增一个变量表，支持读取和写入变量对应的整数值。新增如下指令：

- Load x : 从表中读取变量 x 的值，将其压栈；
- Store x : 将栈顶元素弹出，作为变量 x 的值写入表中；
- Cmp: 若栈顶元素大于或等于0，则修改栈顶元素为1；否则，修改栈顶元素为0；
- Cond: 若栈顶元素非0，则弹出栈顶元素；否则，弹出栈顶元素和次栈顶元素后，压入整数0。

考虑一个仅支持赋值语句的简单语言 L:

$$\begin{aligned} S &\rightarrow \underline{id} := E \mid S; S \\ E &\rightarrow A \mid E \underline{if} B \\ A &\rightarrow A + A \mid A - A \mid (A) \mid \underline{int} \mid \underline{id} \\ B &\rightarrow A > A \mid B \wedge B \mid \neg B \mid \underline{true} \mid \underline{false} \end{aligned}$$

终结符 \underline{id} 表示一个变量，用 $\underline{id}.val$ 取得语法符号对应的语义值。算术表达式新增 \underline{id} ，用来读取变量 \underline{id} 的值。赋值语句 $\underline{id} := E$ 表示将表达式 E 的值写入变量 \underline{id} 。条件表达式 $E \underline{if} B$ 的语义为：若布尔/关系表达式 B 求值为真，则该表达式的值为 E 的值，否则为 0。布尔/关系表达式中， $>$ 为大于， \wedge 为逻辑与， \neg 为逻辑非， \underline{true} 为真， \underline{false} 为假。设 P 为 L 语言的一个程序，若 P 中所有被读取的变量，在读取之前都已经被赋过值，那么称 P 为合法程序。任何一个 L 语言的合法程序都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，对任意程序中出现的变量，表中所存储的值等于程序执行后的实际值。

上述翻译过程可描述成如下 S-翻译模式（与(1)中相同的部分已省略）。请补全空缺的部

分。

```

 $S \rightarrow \underline{id} := E \{ S.instr := E.instr \parallel Store \underline{id} \}$ 
 $S \rightarrow S_1; S_2 \{ S.instr := S1.instr \parallel S2.instr \}$ 
 $E \rightarrow E_1 \underline{if} B \{ E.instr := \dots \}$ 
 $A \rightarrow \underline{id} \{ A.instr := Load \underline{id}.val \}$ 
 $B \rightarrow A_1 > A_2 \{ B.instr := \dots \}$ 
 $B \rightarrow B_1 \wedge B_2 \{ B.instr := \dots \}$ 
 $B \rightarrow \neg B_1 \{ B.instr := \dots \}$ 
 $B \rightarrow \underline{true} \{ B.instr := Push 1 \}$ 
 $B \rightarrow \underline{false} \{ B.instr := Push 0 \}$ 

```

参考答案：

1.

```

 $A \rightarrow A_1 + A_2 \{ A.instr := A2.instr \parallel A1.instr \parallel Plus \}$ 
 $A \rightarrow A_1 - A_2 \{ A.instr := A2.instr \parallel A1.instr \parallel Minus \}$ 

```

2. 答案如下所示，其中 $A_1 > A_2$ 等价于 $(A_1 - A_2 \geq 0) \underline{if} (A_1 - A_2 \neq 0)$ ； $B_1 \wedge B_2$ 等价于 $B_2 \underline{if} B_1$ ； $\neg B_1$ 等价于 $1 \underline{if} (B_1 - 1 \neq 0)$ 。

```

 $E \rightarrow E_1 \underline{if} B \{ E.instr := E1.instr \parallel B.instr \parallel Cond \}$ 
 $B \rightarrow A_1 > A_2 \{ B.instr := A2.instr \parallel A1.instr \parallel Minus \parallel Cmp$ 
 $\parallel A2.instr \parallel A1.instr \parallel Minus \parallel Cond \}$ 
 $B \rightarrow B_1 \wedge B_2 \{ B.instr := B2.instr \parallel B1.instr \parallel Cond \}$ 
 $B \rightarrow \neg B_1 \{ B.instr := Push 1 \parallel Push 1 \parallel B1.instr \parallel Minus \parallel Cond \}$ 

```

Q3. 对于以下 C 语言函数片段，其运行时活动记录按照右图方式组织。

```

1 static int N, L;
2 int foo() {
3     int a[100];
4     int s = 3;
5     int b[N];
6     register int i;
7     for (i = 0; i < L; i++) {
8         b[i] = i * i;
9     }
10    a[10] = b[5];
11    return s;
12 }

```

高地址	
...	旧 SP
返回地址 RA	
旧帧指针 FP	
数组 a	
变量 s	FP
数组 b	SP
低地址	

其中，i 保存在寄存器 T0 中（而非栈上）。本题的活动记录中首先保存返回地址 RA 与旧帧指针 FP，之后按照定义顺序保存静态数组 a 和变量 s，最后保存动态数组 b。其中用 FP 保存分配动态数组（即 b）之前的栈顶指针 SP。本题的活动记录中数组中的元素根据编号由低地

址向高地址存储在栈中，例如 `b[0]` 存储在 `SP` 所指位置。假设全局变量 `N` 存放的内存位置为 `0x4000`。字长为 4 字节。

1. 该函数的栈帧（过程活动记录）总大小为多少字节？写成关于 `N` 的表达式。
2. 在进入函数时，需要先为所有确定大小的变量（数组 `a` 和变量 `s`）和旧 `FP`、`RA` 分配栈空间，之后再为 `b` 动态分配空间。生成的 RISC-V 目标代码如下所示，请补全其中的偏移量。

```

1 func_prologue:
2 addi sp, sp, ----- # SP <- SP + ???
3 sw ra, ----- # Mem[SP + ???] <- RA
4 sw fp, ----- # Mem[SP + ???] <- FP
5 mv fp, sp # FP <- SP
6 lw a0, 0x4000(zero) # A0 <- Mem[0x4000]
7 slli a0, a0, 2 # A0 <- A0 * 4
8 sub sp, sp, a0 # SP <- SP - A0

```

3. 函数中 `a[10] = b[5]`；这条语句对应如下的目标代码，请补全其中的偏移量。

```

1 lw a0, -----(sp) # A0 <- Mem[SP + ???]
2 sw a0, -----(fp) # Mem[FP + ???] <- A0

```

4. 源程序中存在一个缓冲区溢出漏洞，如果 `L` 的值大于 `N` 时会导致 `b` 数组访问越界，此时可能会覆盖掉栈上的一些数据。覆盖数据可能会导致函数返回错误的值，更严重的是，如果被覆盖的数据恰好是返回地址，函数返回时就会跳转到错误的地址。试问 `L` 取值大于等于多少时会覆盖函数的返回值？取值大于等于多少时会覆盖函数的返回地址？写成关于 `N` 的表达式。

参考答案：

1. $4N + 412$ 字节。

2. 如下所示。

```

1 func_prologue:
2 addi sp, sp, -412
3 sw ra, 408
4 sw fp, 404
5 mv fp, sp
6 lw a0, 0x4000(zero)
7 slli a0, a0, 2
8 sub sp, sp, a0

```

3. 如下所示。

```

1 lw a0, 20(sp)
2 sw a0, 44(fp)

```

4. $L \geq N + 1$ 时覆盖返回值， $L \geq N + 102$ 时覆盖返回地址。

Q4. 以下是某简单语言的一段代码。其语法基本与 C 语言一致，但支持嵌套的函数声明，且允许全局语句（类似 Python）。简洁起见，我们目前讨论的函数都没有参数和返回值，每一个函数对应一个静态作用域，程序执行也遵循静态作用域规则。过程活动记录包括控制信息（静态链 SL、动态链 DL、返回地址 RA）和当前作用域声明的变量。右图是程序执行到 `foo()` 被第二次调用时的运行栈状态，其中变量名代表该位置存储了相应变量的值，左侧为单元编号（注意，通常来讲地址从高向底增长，但此处编号从低到高增长）。

```

1  int a, b;
2  void foo() {
3      int x;
4      void bar() {
5          int x, a;
6          a = 3;
7          if (a > b) baz();
8          x = ...;
9      }
10     bar();
11     x = ...;
12 }
13 void baz() {
14     int x;
15     if (a < b) foo();
16     x = ...;
17 }
18 a = 1;
19 b = 2;
20 baz();

```

0	0 (SL)
1	0 (DL)
2	RA
3	a
4	b
5	0 (SL)
6	0 (DL)
7	RA
8	baz.x
9	0 (SL)
10	5 (DL)
11	RA
12	foo.x
13	9 (SL)
14	9 (DL)
15	RA
16	bar.x
17	bar.a
18	----- (SL)
19	----- (DL)
20	-----
21	-----
22	----- (SL)
23	----- (DL)
24	RA
25	x

1. 补全运行栈中缺失的部分。
2. 采用 Display 表来代替静态链。假设采用只在活动记录保存一个 Display 表项的方法，且该表项占居图中 SL 的位置。试指出当前运行状态下 Display 表的内容，以及各活动记录中所保存的 Display 表项的内容（即图中所有 SL 位置的新内容）
3. 若采用动态作用域规则，该程序的执行效果与之前有何不同？（指出第一次控制流不同的时机即可）

参考答案：

1. 单元 18-23 的内容依次为: 0, 13, RA, baz.x, 0, 18。
2. 当前 Display 表为: $D[0] = 0, D[1] = 22, D[2] = 13$ 。活动记录中单元 0, 5, 9, 13, 18, 22 中的内容分别为: 无效, 无效, 5, 无效, 9, 18。
3. 在第二次执行到第 15 行时, 若是静态作用域规则, 则 $a = 1, b = 2$, 因此会再次调用 `foo()`; 若是动态作用域规则, 则 $a = 3, b = 2$, 因此不会调用 `foo()`。