CMPT 379 - Sample Final

The CMPT 379 final exam will probably not be as long as this sample final. The final will be held in C9000 on Dec 13, 2012 (Thu) from 8:30AM to 11:30AM. I expect the exam to be 1.5 hours long, so make sure you arrive at least by 9:30AM, but if you tend to take longer to finish exams, and wish to use 3 hours to write your final, make sure you arrive at 8:30AM.

(1) (10pts) Let the synthesized attribute *val* give the decimal floating point value of the binary number generated by S in the following grammar.

$$\begin{aligned} S &\rightarrow L \,.\, L \mid L \\ L &\rightarrow L\, B \mid B \\ B &\rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

For example, on input `101.101`, the integer part of the number is

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

and the fractional part is

$$1 \times \frac{1}{2^1} + 0 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} = \frac{5}{8}$$

providing the value of $5\frac{5}{8} = 5.625$ for the synthesized attribute $S.val$.

a. Provide an attribute grammar for a syntax-directed definition which determines $S.val$ for all strings in the language. The definition must obey the following condition: the only synthesized attribute for B is c which indicates the contribution of the bit generated by B to the final value. For example, the contribution c of the first and last bits of the input `101.101` is 4 and $\frac{1}{8}$ respectively.

Answer:

```
S -> L
  {
    1.in = (0, 1);
    0.val = 1.val.first;
  }
S -> L . L
  {
    1.in = (0, 1);
    3.in = (0, -1);
    0.val = 1.val.first + 3.val.first;
  }
L -> L B
  {
    1.in = (0.in.first+1 , 0.in.second);
    2.in = (0.in.second < 0) ? -(1.val.second) : 0.in.first;
    0.val = (1.val.first + 2.c , 1.val.second+1);
  }
L -> B
  {
    1.in = (0.in.second < 0) ? -1 : 0.in.first;
    0.val = (1.c, 1);
  }
B -> 0
  { 0.c = 0; }
B -> 1
  { 0.c = 2^0.in; }
```

b. Draw the parse tree for the input `101.101` and decorate the nodes using your syntax-directed definition.

Answer:

```
(S              # S -> L . L                 val = 5 + 0.625 = 5.625
   (L           # L -> L B    0.in = (0,1)   val = (4 + 1, 2 + 1) = (5,3)
     (L         # L -> L B    0.in = (1,1)   val = (4 + 0, 1 + 1) = (4,2)
       (L       # L -> B      0.in = (2,1)   val = (4, 1)
         (B 1)) # B -> 1      0.in = 2       c   = 2^2 = 4
       (B 0))   # B -> 0      0.in = 1       c   = 0
     (B 1))     # B -> 1      0.in = 0       c   = 2^0 = 1
   .
   (L           # L -> L B    0.in = (0,-1)  val = (0.5 + 0.125, 2 + 1) = (0.625,3)
     (L         # L -> L B    0.in = (1,-1)  val = (0.5, 1 + 1) = (0.5,2)
       (L       # L -> B      0.in = (2,-1)  val = (0.5, 1)
         (B 1)) # B -> 1      0.in = -1      c   = 2^(-1) = 1/2 = 0.5
       (B 0))   # B -> 0      0.in = -2      c   = 0
     (B 1)))    # B -> 1      0.in = -3      c   = 2^(-3) = 1/8 = 0.125
```

c. Eliminate left-recursion from the grammar and provide a new syntax-directed definition that works with this modified grammar.

(2) Consider the following expression grammar.

$$
\begin{array}{lcl}
E & \rightarrow & E \text{ `+'} T \\
 & | & T \\
T & \rightarrow & T \text{ `*'} F \\
 & | & F \\
F & \rightarrow & \texttt{exp} \text{ `('} E \text{ `)'} \\
 & | & \texttt{ln} \text{ `('} E \text{ `)'} \\
 & | & \text{`-'} F \\
 & | & \text{`x'} \\
 & | & \texttt{c}
\end{array}
$$

We assume a lexical analyzer that provides the tokens we need. For instance, `c` is an integer constant token. Note that `exp` is the exponential function, i.e. `exp(x)` is e^x and `ln` is the natural logarithm, i.e. `ln(x)` is $ln(x)$ also written as $log_e(x)$.

a. Provide a L-attributed syntax directed definition that computes the derivative of an input expression. Explain each attribute used in your attribute grammar.

D[input string]	output string = derivative(input string)
$D[c]$	0
$D[x]$	1
$D[x + c]$	1
$D[E_1 + E_2]$	$D[E_1] + D[E_2]$
$D[-E]$	$-D[E]$
$D[c * E]$	$c * D[E]$
$D[E_1 * E_2]$	$E_1 * D[E_2] + E_2 * D[E_1]$
$D[exp(x)]$	$exp(x)$
$D[ln(x)]$	$1/x$
$D[f(E)]$	$D[E] * f'(E)$, f' is the derivative of f if $f(E)$ is $exp(E)$, $f'(E)$ is $exp(E)$ if $f(E)$ is $ln(E)$, $f'(E)$ is $1/E$

Answer: The following solution also simplifies the expressions. The simplification is not required but it is nice to have.

Production	Semantic Rule
$E \rightarrow E$ **'+'** T	dv := simplify(+, 1.dv, 3.dv); ex := '1.ex + 2.ex';
$E \rightarrow T$	dv := 1.dv; ex := 1.ex;

$T \rightarrow T$ **'*'** F	t1 := simplify(*, 1.ex, 3.dv);
	t2 := simplify(*, 3.ex, 1.dv);
	dv := simplify(+, t1, t2); ex := '1.ex * 2.ex';

$T \rightarrow F$	dv := 1.dv; ex := 1.ex;
$F \rightarrow$ `exp` **'('** E **')'**	dv := simplify(*, 3.dv, 'exp(3.ex)'); ex := 'exp(3.ex)';
$F \rightarrow$ `ln` **'('** E **')'**	dv := simplify(*, 3.dv, '1 / 3.ex'); ex := 'ln(3.ex)';
$F \rightarrow$ **'-'** F	0.dv := '− 1.dv'; 0.ex = '− 1.ex';
$F \rightarrow$ **'x'**	0.dv = 1; 0.ex = x;
$F \rightarrow$ `c`	0.dv := 0; 0.ex := c.lexval;

Pseudo-code to simplify an expression:

```
string simplify (string op, string a, string b)
{
  if (isInteger(a) and isInteger(b)) {
    if (op eq '+') return string(int(a) + int(b));
    if (op eq '*') return string(int(a) * int(b));
  }
  if (op eq '+') {
    if (a eq '0') return b;
    if (b eq '0') return a;
  }
  if ((op eq '*') and ((a eq '0') or (b eq '0'))) return '0';
  return 'a op b';
}
```

b. Using your syntax-directed definition provide the derivative for the input string shown below. Provide the parse tree for the input string and the attribute values at each node in the tree.

`exp(2 * x + 4)`

Answer:

```
2 * exp(2 * x + 4)

(E                      # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
  (T                    # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
    (F                  # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
      exp
      \(
      (E                # dv = 2 + 0 = 2, ex = 2 * x + 4
         (E             # dv = 2, ex = 2 * x
           (T           # dv = 2 * 1 + x * 0 = 2, ex = 2 * x
             (T         # dv = 0, ex = 2
               (F 2))   # dv = 0, ex = 2
             *
             (F x)))    # dv = 1, ex = x
         +
         (T             # dv = 0, ex = 4
           (F 4)))      # dv = 0, ex = 4
      \))))
```

c. *(optional; no marks)* Extend your syntax-directed definition so that it can handle second derivatives, and third derivatives. For example, the derivative of `x * x + 2 * x` will be `x + x + 2`, and the second derivative (derivative of the derivative) will be `2` and the third derivative will be `0`.

(3) **Code Generation**

The following attribute grammar implements code-generation for a fragment of a programming language. The following functions are used for code generation (and only these functions can be used in your answers):

- *label(L)* creates a new label *L* : each time it is called, e.g. calling *label(loop)* for the first time would result in `loop1:` generated in the output, and subsequent calls to *label(loop)* would generate `loop2:`, `loop3:`, etc.
- *goto(label)* creates a goto statement to the label *label*.
- *newloc(x)* creates a new location each time it is called, e.g. `x1`, `x2`, etc. just like the *label* function. The location is either allocated on the stack or assigned to a register – it does not matter for this question.
- *load(loc, value)* loads *value* into a location *loc*.
- *if(cond, label1, label2)* creates a conditional branch: it generates an instruction that does *goto label1* if *cond* is true and *goto label2* if false.
- *x := val* creates a *local* variable *x* with value *val*.
- Other functions are explained directly in the definition as comments (preceded by //).

Rules	Syntax-directed definition
$P \rightarrow S$	\$0.code = \$1.code + return(0); // return zero constant
$S \rightarrow A$	\$0.code = \$1.code;
$S \rightarrow$ '{' A ';' A '}'	\$0.code = \$2.code + \$4.code;

$S \rightarrow$ **while** '(' E ')' S	loop := label(loop);
	cond := loop + \$3.code;
	body := label(body);
	end := label(end);
	branch := if(\$3.loc, body, end);
	\$0.code = ? // *incomplete*

$E \rightarrow C$ **and** C	result := newloc(andtmp);
	instr := and(result, \$1.loc, \$3.loc); // new *and* instruction
	\$0.code = \$1.code + \$3.code + instr;
	\$0.loc = result;

$E \rightarrow C$ **or** C	result := newloc(ortmp);
	instr := or(result, \$1.loc, \$3.loc); // new *or* instruction
	\$0.code = \$1.code + \$3.code + instr;
	\$0.loc = result;

$E \rightarrow C$	\$0.code = \$1.code; \$0.loc = \$1.loc;

$C \rightarrow$ **true**	result := newloc(truetmp);
	instr := load(result, true);
	\$0.code = instr;
	\$0.loc = result;

$C \rightarrow$ **false**	result := newloc(falsetmp);
	instr := load(result, false);
	\$0.code = instr;
	\$0.loc = result;

$A \rightarrow$ **nop**	\$0.code = nop; // does nothing
$A \rightarrow$ **break**	?
$A \rightarrow$ **continue**	?

A is a statement, E is an expression, and C is a boolean. **and** , **or** are the usual boolean operators. The string concatenation operator + concatenates three-address instructions and labels and inserts the appropriate whitespace for readable output code. A label can be concatenated with instructions simply using the + operator appropriately. For example, *L := label(loop); C := L + goto(L)* stores the value `L1:goto L1` into the local variable C.

a. (1pt) Is the attribute *code* in the above syntax directed definition a *synthesized* attribute or an *inherited* attribute?

Answer: synthesized

b. (3pts) Complete the code generation for the *while* statement.

Rules	Syntax-directed definition

$S \rightarrow$ **while** '(' E ')' S	loop := label(loop);
	cond := loop + $3.code;
	body := label(body);
	end := label(end);
	branch := if($3.loc, body, end);
	$0.code = ? // *incomplete*

Answer:

Rules	Syntax-directed definition
$S \rightarrow$ **while** '(' E ')' S	whilebody := body + $5.code + goto(loop); $0.code = cond + branch + whilebody + end;

c. (2pts) Provide the output generated by the syntax directed definition for the input:
`while (true) { nop; nop }`

Answer:

```
loop1:
  load(truetmp1, true)
  if truetmp1 goto body1, goto end1
body1:
  nop
  nop
  goto loop1
end1:
  return 0
```

d. (4pts) Use *inherited* attributes to add the correct syntax directed definition for the rules $A \rightarrow$ **break** and $A \rightarrow$ **continue**

Answer:

Rules	Syntax-directed definition
$S \rightarrow$ **while** '(' E ')' S	$5.loopbegin = loop; $5.loopend = end;
$S \rightarrow A$	$1.loopbegin = $0.loopbegin; $1.loopend = $0.loopend;
$S \rightarrow$ '{' A ';' A '}'	$2.loopbegin = $4.loopbegin = $0.loopbegin; $2.loopend = $4.loopend = $0.loopend;
$A \rightarrow$ **break**	$0.code = goto($0.loopend);
$A \rightarrow$ **continue**	$0.code = goto($0.loopbegin);

e. (5pts) Write down a brief description of short-circuit evaluation for the **and** operator. Add boolean short-circuit evaluation to the rule: $E \rightarrow C$ **and** C. You must use a ϕ function to get the right result in \$0.*loc*.

Answer: If the first argument to **and** is false then E must be false and the second argument is not evaluated.

Rules	Syntax-directed definition

$E \rightarrow C$ **and** C	noskct := label(noskct);
	skct := label(skct);
	end := label(end);
	ifinstr := \$1.code + if(\$1.loc, noskct, skct);
	noskctresult := newloc(andtmp);
	andinstr := and(noskctresult, \$1.loc, \$3.loc);
	noskctinstr := noskct + \$3.code + andinstr + goto(end);
	skctresult := newloc(skctresult);
	skctinstr := skct + load(skctresult, \$1.loc) + goto(end);
	result := newloc(result);
	phinode := ϕ(result, noskctresult, skctresult);
	\$0.code = \$1.code + ifinstr + noskctinstr + skctinstr + end + phinode;
	\$0.loc = result;

(4) Consider the following three-address code (TAC) program:

```
    i := m - 1
    j := n
    t1 := 4 * n
    v := A[ t1 ]
L1: i := i + 1
    t2 := 4 * i
    t3 := A[ t2 ]
    if t3 < v goto L1
L2: j := j - 1
    t4 := 4 * j
    t5 := A[ t4 ]
    if t5 > v goto L2
    if i >= j goto L3
    t6 := 4 * i
    x := A[ t6 ]
    t7 := 4 * i
    t8 := 4 * j
    t9 := A[ t8 ]
    A[ t7 ] := t9
    t10 := 4 * j
    A[ t10 ] := x
    goto L1
L3: t11 := 4 * i
    x := A[ t11 ]
```

```
t12 := 4 * i
t13 := 4 * n
t14 := A[ t13 ]
A[ t12 ] := t14
t15 := 4 * n
A[ t15 ] := x
```

a. Construct the control flowgraph for the above TAC program.

Answer:

```
B1:
  i = m-1
  j = n
  t1 = 4 * n
  v = A[t1]

B2:
  i = i+1
  t2 = 4 * i
  t3 = A[t2]
  if t3<v goto B2

B3:
  j = j-1
  t4 = 4*j
  t5 = A[t4]
  if t5>v goto B3

B4:
  if i>=j goto B6

B5:
  t6 = 4*i
  x = A[t6]
  t7 = 4*i
  t8 = 4*j
  t9 = A[t8]
  A[t7] = t9
  t10 = 4*j
  A[t10] = x
  goto B2

B6:
  t11 = 4*i
  x = A[t11]
  t12 = 4*i
  t13 = 4*n
  t14 = A[t13]
  A[t12] =  t14
  t15 = 4*n
  A[t15] = x

B1 --> B2 --> B3 --> B4 --> B5
B2 --> B2
B3 --> B3
B5 --> B2
```

b. Perform *local* common subexpression elimination (i.e. only eliminate common subexpressions within each basic block) and provide the revised control flowgraph.

Answer:

```
B5:
  t6 = 4*i
  x = A[t6]
  t8 = 4*j
  t9 = A[t8]
  A[t6] = t9
  A[t8] = x
  goto B2

B6:
  t11 = 4*i
  x = A[t11]
  t13 = 4*n
  t14 = A[t13]
  A[t11] = t14
  A[t13] = x
```

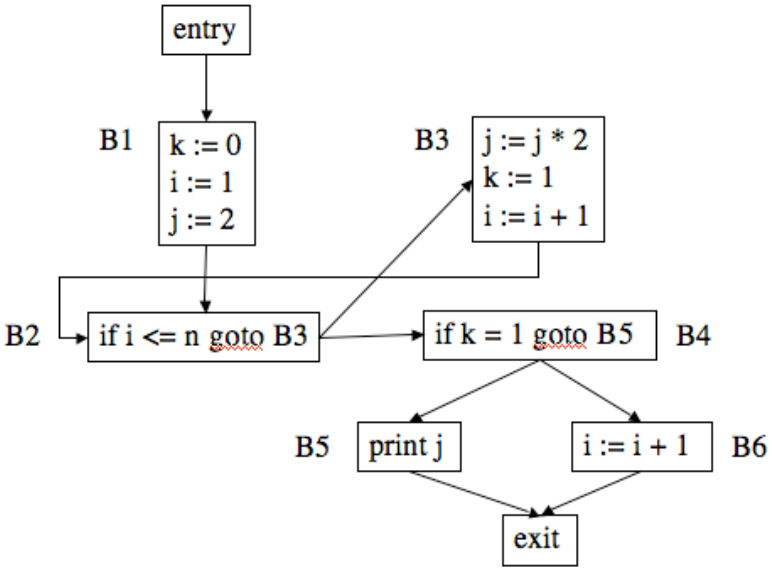
c. The instruction `t4 := 4*j` is repeatedly executed inside an inner loop (as can be seen in the control flow graph). Analyze the change in values in `t4` and reduce the strength of this instruction by replacing the multiplication operation with the cheaper operation (such as addition or subtraction). In order to do this, you can add new instructions using expensive operations like multiplication as long as these instructions are added outside the inner loop.

Answer:

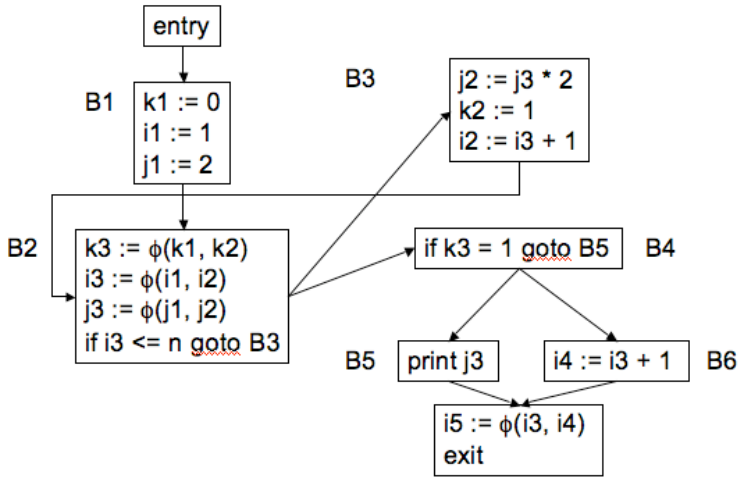
```
B1:
  i = m-1
  j = n
  t1 = 4*n
  v = A[t1]
  t4 = 4*j

B3:
  j = j-1
  t4 = t4-4
  t5 = A[t4]
  if t5>v goto B3
```

d. For the following flowgraph construct the flowgraph in minimal Static Single Assignment (SSA) form. A *minimal* SSA form has no redundant static variable definitions.



Answer:



(5) **Code Optimization and Static Single-Assignment Form**

Consider the following intermediate code fragment:

```
L1: a := 0
L2: b := a + 1
    if c > 0 goto L3, else goto L4
L3: c := c + b
    a := b * 2
    if a < 100 goto L2, else goto L4
L4: return c
```

a. (1pt) Draw the control flow graph for the above intermediate code. Use the labels in the code above (or just the label numbers) as the name for each basic block.

Answer:

```
L1: a := 0

L1 --> L2

L2: b := a + 1
    c > 0

L2: true--> L3;
L2: false--> L4

L3: c := c + b
    a := b * 2
    a < 100

L3: true--> L2;
L3: false--> L4

L4: return c
```

b. (4pts) Liveness analysis computes *in* and *out* sets for each basic block. If $succ(b)$ is a function that defines the successor for a basic block b, and *use* defines the variables occurring in *r*-values, and *def* defines the variables occurring in *l*-values, then for each basic block b:

$$\begin{aligned} in[b] &= use[b] \cup (out[b] - def[b]) \\ out[b] &= \bigcup_{s \in succ(b)} in[s] \end{aligned}$$

Write down the *in*, *out* sets for the control flow graph from (5a). Provide the *use*, *def* sets and the *in*, *out* sets at each iteration of the fixed-point computation of *in*, *out* for each basic block.

Answer:

	use/def	1st in/out	2nd in/out	3rd in/out	4th in/out
L1	/ a	/	/ ac	c / ac	c / ac
L2	ac / b	ac /	ac / abc	ac / abc	ac / abc
L3	abc / ac	abc / ac	abc / ac	abc / ac	abc / ac
L4	c /	c	c /	c /	c /

c. (2pts) A node X in a control flow graph dominates node Y if all paths from the start node to Y goes through X. Provide the set of nodes dominated by each node in the above control flow graph.

Answer:

$$\begin{aligned} D(1) &= \{2,3,4\} \\ D(2) &= \{3,4\} \\ D(3) &= \{\} \\ D(4) &= \{\} \end{aligned}$$

d. (2pts) X strictly dominates Y if X dominates Y and $X \neq Y$. The dominance frontier of X is the set of of all nodes Y where X dominates a predecessor of Y and X does not strictly dominate Y. Provide the dominance frontier of each node in the control flow graph.

Answer:

$$\begin{aligned} DF(1) &= \{\} \\ DF(2) &= \{2\} \\ DF(3) &= \{2,4\} \\ DF(4) &= \{\} \end{aligned}$$

e. (4pts) Use the (iterated) dominance frontier condition to write down the control flow graph from (5a) in static single assignment (SSA) form. If a variable is not *used* in a basic block then a ϕ function is not required for that variable.

Answer:

```
L1: a1 := 0

L1 --> L2

L2: a2 := phi(a1,a3)
    c2 := phi(c1,c3)
    b := phi(b,b) <== not required, but ok as per DF
    b1 := a2 + 1
    c2 > 0

L2: true--> L3;
L2: false--> L4

L3: c3 := c2 + b1
    a3 := b1 * 2
    a3 < 100

L3: true--> L2;
L3: false--> L4

L4: c4 := phi(c2,c3)
    a := phi(a,a) <== not required, but ok as per DF
    return c4
```

f. (2pts) Assume that variable `c := 0`. Perform constant propagation in the SSA form from (5e) and provide the new SSA form. Insert the ϕ functions based on the dominance frontiers for each node, and then add subscripts to variables. Do not include any ϕ functions for

variables that are not used in a basic block.

Answer:

```
L1:

L1 --> L2

L2: a2 := phi(0,a3)
    c2 := phi(0,c3)
    b1 := a2 + 1
    c2 > 0

L2: true--> L3;
L2: false--> L4

L3: c3 := c2 + b1
    a3 := b1 * 2
    a3 < 100

L3: true--> L2;
L3: false--> L4

L4: c4 = phi(c2,c3)
    return c4
```

(6) (10pts) Consider the following (valid) C program:

```
 1    int foo(int f(int,int), int g(int,int), int x) {
 2      int y = g(x,x);
 3      return f(y,y);
 4    }

 5    int sq(int x, int y) {
 6      if (x == 1) { return y; }
 7      return pl(y, sq(x-1, y));
 8    }

 9    int pl(int x, int y) {
10      if (x == 0) { return y; }
11      return pl(x-1, y+1);
12    }

13    int main() {
14      printf("output=%d\n", foo(sq, pl, 1));
15    }
```

a. Draw the activation tree showing the runtime execution of the above C program. Also,

provide the output printed out by the program.

Answer:

```
main()
 |  prints:
 |   output=4
 |
 +--foo(sq, pl, 1) returns 4
     |
     +--pl(1,1) returns 2
     |   |
     |   +--pl(0,2) returns 2
     |
     +--sq(2,2) returns 4
         |
         +--pl(2,sq(1,2)) returns 4
             |   |
             |   +--sq(1,2) returns 2
             |
             +--pl(1,3) returns 4
                 |
                 +--pl(0,4) returns 4
```

b. Consider a hardware architecture where the activation frame for each function, including the `main` function, takes at least 32 bytes. Assume that *all* callee-saved registers for the local variables can be stored within the activation frame of size 32 bytes. Provide the *maximum* amount of stack memory in bytes used by the program during its execution. If you've forgotten how to multiply two numbers it is permissible to write down ($x \times 32$) bytes as the answer.

Answer:

```
main() -- foo(sq,pl,1) -- sq(2,2) -- pl(2,2) -- pl(1,3) -- pl(0,4)
==> 6*32 = 192 bytes.

push main() +32
 push foo(sq,pl,1) +32
   push pl(1,1) +32
     push pl(0,2) +32
     pop pl(0,2) -32
   pop pl(1,1) -32
   push sq(2,2) +32
     push sq(1,2) +32
     pop sq(1,2) -32
     push pl(2,2) +32
       push pl(1,3) +32
         push pl(0,4) +32
         pop pl(0,4) -32
       pop pl(1,3) -32
     pop pl(2,2) -32
   pop sq(2,2) -32
 pop foo(sq,pl,1) -32
pop main()  -32
```