The CPU's view of a Program

From our perspective, we see lot's of complicated things going on in our programming languages: inheritance, overloading, user defined operators, anonymous functions, closure, garbage collection, monads, generic programming, user defined types, . . . The list goes on and on.

But **from the CPU's point of view**, the only things a program[1] does is swap values to and fro registers and the main memory cells, compute with integers and floating point values, and fetch instructions with spurts of sequential reads fragmented by frenzied jumps in one particular zone of memory.

While the **machinery** used for these specific tasks can be complex — the **tasks themselves are simple and small in number**.

A userland **program**, an OS kernel actually gets to talk to the outside world.

The CPU's view of a Program

The point being...

i. We've seen how to count and allocate registers for the evaluation of expression trees (REGISTERNEEDS).
ii. We've seen how to generate machine code instructions for expression results (TREECG)

The CPU's view of a Program

The point being...

i. We've seen how to count and allocate registers for the evaluation of expression trees (REGISTERNEEDS).
ii. We've seen how to generate machine code instructions for expression results (TREECG)
iii. **Assignment through =** to a LHS identifier is typically managed by treating = as just another binary operation of expression trees (`treeCG` again).

The CPU's view of a Program

The point being...

i. We've seen how to count and allocate registers for the evaluation of expression trees (REGISTERNEEDS).
ii. We've seen how to generate machine code instructions for expression results (TREECG)
iii. **Assignment through =** to a LHS identifier is typically managed by treating = as just another binary operation of expression trees (`treeCG` again).
iv. Calculating the Boolean truth of an expression with a relational operator (==, <) is straightforward, CPU instruction sets might set bits in a general or specialized register when a `COMPARE` instruction is executed; or leave a 1 or 0 in the resultant register ($x < y \rightarrow$ `LT R4,R6,R4` for x in `R6`, y in `R4`). So **again**, the predicate value calculation instructions are generated in `treeCG`.

The CPU's view of a Program

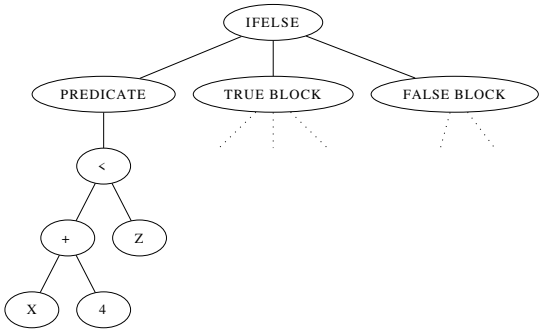
The point being...

i. We've seen how to count and allocate registers for the evaluation of expression trees (REGISTERNEEDS).
ii. We've seen how to generate machine code instructions for expression results (TREECG)
iii. **Assignment through =** to a LHS identifier is typically managed by treating = as just another binary operation of expression trees (`treeCG` again).
iv. Calculating the Boolean truth of an expression with a relational operator (==, <) is straightforward, CPU instruction sets might set bits in a general or specialized register when a `COMPARE` instruction is executed; or leave a 1 or 0 in the resultant register ($x < y \rightarrow$ `LT R4,R6,R4` for x in `R6`, y in `R4`). So **again**, the predicate value calculation instructions are generated in `treeCG`.
v. **All that remains** for our basic compiler is learning to manage the program counter (and various other registers) to implement **standard control structures** and **function calls**.

PC Control for `if-else`

Control structures such as `if`, `if-else`, `do-while`, `while-do`, `foreach`, `for i=...` also manipulate the PC.

There implementation is straight-forward. We begin with the control element's essential AST structure after **syntax directed translation**.



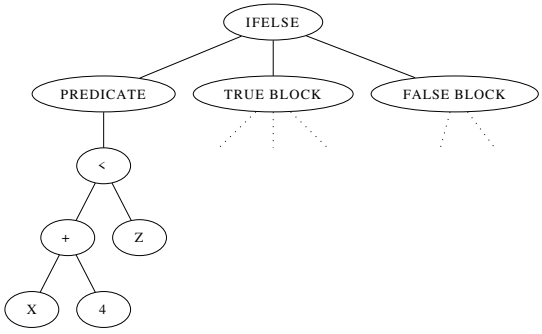
`if-else` logic in the AST. Predicate expression tree shown, subtrees of `TRUE BLOCK` and `FALSE BLOCK` possibly holding many programming statements not shown.

PC Control for `if-else`

On the `treeCG` visitor pass, we generate code for the prediciate...

If X is a parameter whose value is on the stack 12 words away from FP, and Z is a local variable 7 words away from FP, what sequence of code will be generated by

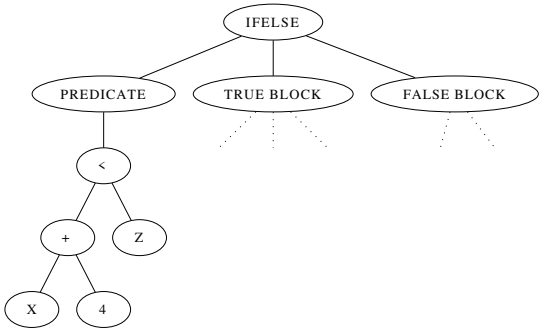
TREECG($PREDICATE$, $\{R5, R6, R7, R8\}$)?



PC Control for `if-else`

On the `treeCG` visitor pass, we generate code for the prediciate...
If X is a parameter whose value is on the stack 12 words away from FP, and Z is a local variable 7 words away from FP, what sequence of code will be generated by

TREECG($PREDICATE$, $\{R5, R6, R7, R8\}$)?




```
treeCG(lt,[R5,R6,R7,R8])
  treeCG(plus,[R5,R6,R7,R8])
    treeCG(X,[R5,R6,R7,R8])
      generate(load,R5,X)       LD R5,@12w(FP)
    generate(immadd,R5,R5,4)    ADD R5,R5,#4
  treeCG(Z,[R6,R7,R8])
    generate(load,R6,Z)         LD R6,@7w(FP)
  generate(lt,R5,R5,R6)         LT R5,R5,R6
```

PC Control for `if-else`

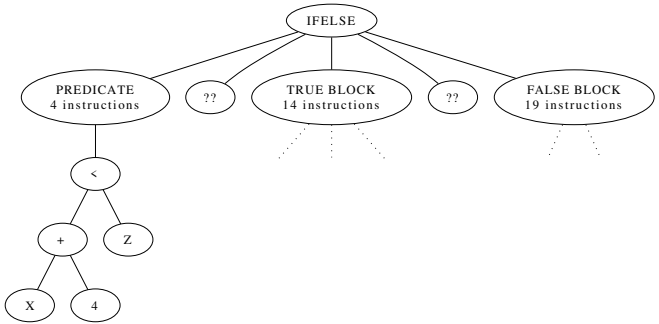
After the `treeCG` visitor pass, we make a “control structures” pass.

For each control structure encountered, its subtree is augmented with PC manipulation logic (**alternatively**, these placeholder branching nodes could be **added during SDT**).

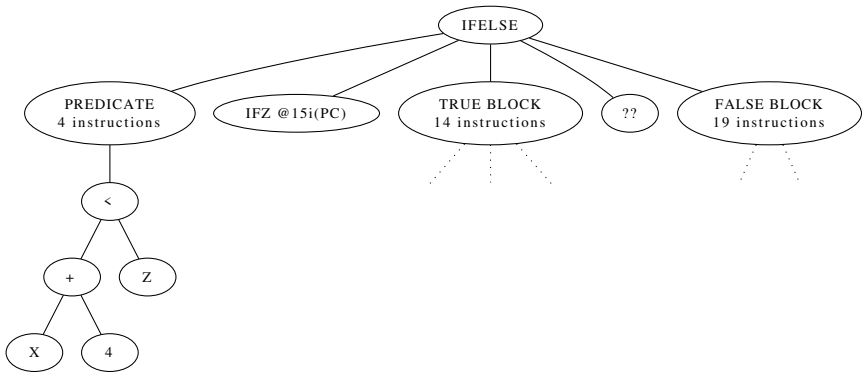
If `treeCG` used **register targetting** to put the result of

$$X + 4 < Z$$

into R5, what type of branch or jump statement is needed after the predicate?



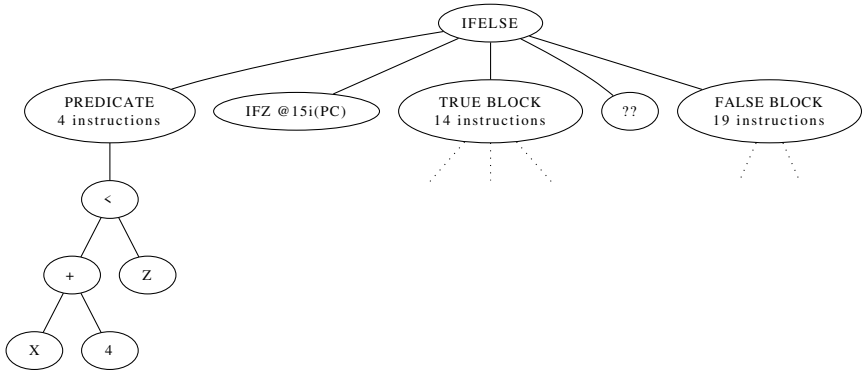
PC Control for `if-else`



`IFZ R5,@15i(PC)` — Jump forward $15 = 14 + 1$ instructions from PC if the value in R5 is zero.

We prefer PC jumps with **relative offsets**, this allows code to be loaded into arbitrary memory addresses without alot of address fix-ups by the OS loader.

PC Control for `if-else`

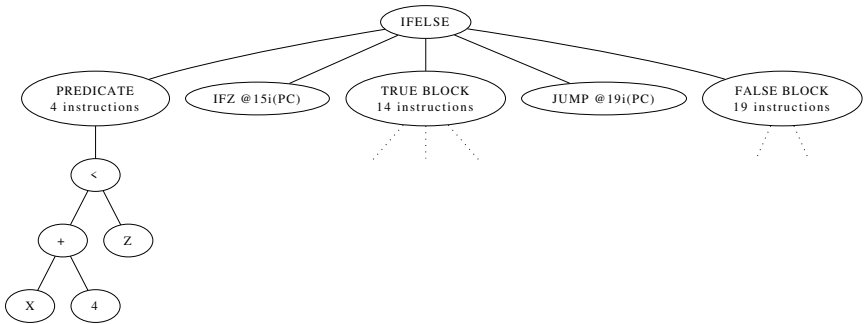


When manipulating the PC with relative offsets, the PC value is **one instruction word past** the instruction being executed.

So why did we add 15 instead of 14?

PC Control for `if-else`

Because we need a JUMP instruction after the TRUE BLOCK statements to avoid the FALSE BLOCK.



`JUMP @19i(PC)` — Jump forward 19 instructions from PC (unconditionally).

Basic Function Call Management?

CALLCG(*functNode*, *regList*, *paramRegs*)

The exact code sequences vary according to hardware and operating system conventions.

Crafting a Compiler §13.1.3

In this case “operating systems” means the **runtime environment** of the instructions.

So as not to confuse things (too much), we’ll use the essential function call interface presented in the text (§13.1.3).

CALLCG(*functNode*, *regList*, *paramRegs*)

A quick reminder of context... treeCG.pdf

```
  ) else (
    // There are enough registers, but we must eval the "larger" first
    if ( left.regCount > right.regCount ) then (
      treeCG( left, regList )      // result in r1 due to register targeting
      treeCG( right, tail(regList) )      // result in r2
      <binaryOp> r1,r1,r2     // r1 ← r1 <binaryOp> r2
    ) else (
      treeCG( right, regList )
      treeCG( left, tail(regList) )
      <binaryOp> r1,r2,r1     // r1 ← r2 <binaryOp> r1
    )
  ) else if ( T.kind is FunctionCall ) then (
    callCG( T, regList, ··· )     // Future lecture :)
  )
)
```

CALLCG(*functNode*, *regList*, *paramRegs*)

i. Many architectures pass a certain number of parameters (**arguments**) via dedicated registers, we'll call these A_i, $i \in \{1, 2, \ldots, a\}$. They may be dedicated registers (MIPS `$a0`, `$a1`) or general purpose registers "dedicated" by convention (R12,R13).

ii. The function call has n parameters ($n \geq 0$) labeled p_1, p_2, ..., p_n

iii. $allocatableRegs$ is a list of general purpose registers available for $regsList$ use (**not working registers**!)

iv. Return values are stored on the stack (again, some architectures use a dedicated register).

v. $\mathcal{R}$ is the set of calle**r** saved registers, $\mathcal{E}$ is the set of call**ee** saved regs.

vi. `@function` means the address of a `function`'s first instruction; and `CALL @function` saves `RA` $\leftarrow$ `PC` and stores `PC` $\leftarrow$ `@function`. `RETURN` stores `PC` $\leftarrow$ `RA`.

vii. Recall that in this course the SP "grows" downward and "pops up".

viii. **Work registers** are `W0`, `W1`, ...

ix. $*Sz$ and $*Offset$ terms are in absolute bytes, $Sz(\cdot)$ calculates bytes consumed by a register set.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ `treeCG(`p_n`,`*regList*`)`; `PUSH` *head*(*regList*)	`PUSH RA` store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ `treeCG(`p_{n-1}`,`*regList*`)`; `PUSH` *head*(*regList*)	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ `treeCG(`p_{n-2}`,`*regList*`)`; `PUSH` *head*(*regList*)	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
	...	`MV FP,SP` Set the **callee's** FP	14
6	if $a > 0$ `treeCG(`p_a, $[A_a]$ + *regList*`)`)	function body logic	15
7	if $a-1 > 0$ `treeCG(`p_{a-1}, $[A_{a-1}]$ + *regList*`)`)	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
	...	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
8	if $a \geq 1$ `treeCG(`p_1, $[A_1]$ + *regList*`)`)	`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		`POP RA` restore the RA from stack (optional?)	19
9	`CALL @function` $\Rightarrow$	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ `RETURN`	21
23	`POP` *head*(*regList*)		
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Steps for orchestrating a function call are performed in **Seq** order, steps 1–9 and 22–24 are performed by the **caller**, steps 10–21 are performed by the **callee**.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

How does the **caller** know which *allocatable* registers are in use?

$$CallerRegsInUse = (set(allocatableRegs) - set(regsList))$$

Remember **we are in a special case of `treeCG`**, any registers in *regList* **are not storing values** needed elsewhere in `treeCG` logic. There may be other $CallerRegsInUse$, such as A_1 if the caller takes one argument.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
3	`if` $n > a$ `treeCG(`p_n`,`$regList$`);PUSH`$head(regList)$	`PUSH RA` store the **callee's** RA on stack (optional?)	11
4	`if` $n-1 > a$ `treeCG(`p_{n-1}`,`$regList$`);PUSH`$head(regList)$	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	`if` $n-2 > a$ `treeCG(`p_{n-2}`,`$regList$`);PUSH`$head(regList)$	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
	...	`MV FP,SP` Set the **callee's** FP	14
6	`if` $a > 0$ `treeCG(`p_a`,` $[A_a] +$ `regList))`	function body logic	15
7	`if` $a-1 > 0$ `treeCG(`p_{a-1}`,` $[A_{a-1}] +$ `regList))`	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
	...	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
8	`if` $a \geq 1$ `treeCG(`p_1`,` $[A_1] +$ `regList))`	`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		`POP RA` restore the RA from stack (optional?)	19
9	`CALL @function` $\Rightarrow$	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ `RETURN`	21
23	`POP` $head(regList)$		
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Target architectures requiring double word alignment typically require SP and (or) FP to be double aligned on `CALL`. The compiler assumes these registers are aligned on **function entry**, which means it can assure they are aligned on subsequent calls.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot		
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
	...	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
		MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a] + regList$))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}] + regList$))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1] + regList$))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

We don't need to hold parameter values in registers if they end up on the stack. Push them immediately after calculation.

In this case, the register counting solution for function parameters *might* be as simple as $max\{\texttt{registerNeeds}(p_i) - (i \leq a)\}_{i=1}^{n}$.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
3	`if` $n > a$ `treeCG(`p_n`,`*regList*`);` `PUSH` *head*(*regList*)	`PUSH RA` store the **callee's** RA on stack (optional?)	11
4	`if` $n-1 > a$ `treeCG(`p_{n-1}`,`*regList*`);` `PUSH` *head*(*regList*)	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	`if` $n-2 > a$ `treeCG(`p_{n-2}`,`*regList*`);` `PUSH` *head*(*regList*)	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
	...	`MV FP,SP` Set the **callee's** FP	14
6	`if` $a > 0$ `treeCG(`p_a`,` $[A_a] +$ *regList*`))`	function body logic	15
7	`if` $a-1 > 0$ `treeCG(`p_{a-1}`,` $[A_{a-1}] +$ *regList*`))`	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
	...	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
8	`if` $a \geq 1$ `treeCG(`p_1`,` $[A_1] +$ *regList*`))`	`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		`POP RA` restore the RA from stack (optional?)	19
9	`CALL @function` $\Rightarrow$	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ `RETURN`	21
23	`POP` *head*(*regList*)		
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Parameters passed via registers use register targeting **and are stored in their dedicated A_i register**.
Notice that each parameter calculated gets `PUSH`'d or stored to a dedicated A_i register.
Suppose our architecture has $a = 2$, the caller takes two parameters and the callee takes three. What unmentioned danger exists when calculating the value for the callee's p_1?

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Step 9 caller $\Rightarrow$ step 10 in callee...

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
3	`if` $n > a$ `treeCG(`p_n`,`*regList*`);` `PUSH` *head*(*regList*)	`PUSH RA` store the **callee's** RA on stack (optional?)	11
4	`if` $n-1 > a$ `treeCG(`p_{n-1}`,`*regList*`);` `PUSH` *head*(*regList*)	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	`if` $n-2 > a$ `treeCG(`p_{n-2}`,`*regList*`);` `PUSH` *head*(*regList*)	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
	...	`MV FP,SP` Set the **callee's** FP	14
6	`if` $a > 0$ `treeCG(`p_a`,` $[A_a] +$ *regList*`))`	function body logic	15
7	`if` $a-1 > 0$ `treeCG(`p_{a-1}`,` $[A_{a-1}] +$ *regList*`))`	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
	...	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
8	`if` $a \geq 1$ `treeCG(`p_1`,` $[A_1] +$ *regList*`))`	`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		`POP RA` restore the RA from stack (optional?)	19
9	`CALL @function` $\Rightarrow$	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ `RETURN`	21
23	`POP` *head*(*regList*)		
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Optional? Because **callee** may not make any function calls itself, in which case the RA back into **caller** doesn't need to be preserved.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

How does **callee** know which registers it might use? Because it has register counts for all expression trees in its logic.

$$neededRegs = \max \{\, expr.regCount \mid expr \in Callee \,\}$$

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

How does **callee** know which registers it might use? Because it has register counts for all expression trees in its logic.

$$RegsCalleeMightUse = set(allocatableRegs[1, \ldots, neededRegs])$$

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

`localSz` is space consumed by local variables stored in the frame of the **callee**. This list is accumulated with the symbol table's visitor function and can be stored in the **callee**'s root AST node for preservation during compilation. Local variables are typically accessed in generated code for **callee** with `@offset(FP)` immediate notation (namely in TREECG load and store operations).

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

This construction is specific to FPs with positive immediate offsets for local variable access. How do these steps change if we wish to use FPs with **negative** immediate offsets? Is there a more natural construction for *procedure level* vs *block level* frame allocation?

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot		
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
	...	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
		MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	15	

Step 15 in callee. Why can't we simply use SP as our base register for local variable offsets in the function body instructions?

Why is step 14 necessary? Is it *always* necessary?

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot		
3	`if` $n > a$ `treeCG(`p_n`,`$regList$`);PUSH` $head(regList)$	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
4	`if` $n-1 > a$ `treeCG(`p_{n-1}`,`$regList$`);PUSH` $head(regList)$	`PUSH RA` store the **callee's** RA on stack (optional?)	11
5	`if` $n-2 > a$ `treeCG(`p_{n-2}`,`$regList$`);PUSH` $head(regList)$	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
	...	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
6	`if` $a > 0$ `treeCG(`p_a`,` $[A_a] + regList$`))`	`MV FP,SP` Set the **callee's** FP	14
7	`if` $a-1 > 0$ `treeCG(`p_{a-1}`,` $[A_{a-1}] + regList$`))`	function body logic	15
	...	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
8	`if` $a \geq 1$ `treeCG(`p_1`,` $[A_1] + regList$`))`	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
		`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
9	`CALL @function` $\Rightarrow$	`POP RA` restore the RA from stack (optional?)	19
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
23	`POP` $head(regList)$	$\Leftarrow$ `RETURN`	21
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Store the return value for the caller (presumably located at `@??(FP)`, some local variable in the function definition). We might need to shuffle it through a working register from a local variable location to the space reserved by the caller. In `zlang` (ZOBOS, CZAR projects) the return variable could simply use the storage space at `@#returnOffset(FP)`.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	`PUSH` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	`PUSH FP` if `FP` $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ `treeCG(`p_n`,` *regList*`);` `PUSH` *head*(*regList*)	`PUSH RA` store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ `treeCG(`p_{n-1}`,` *regList*`);` `PUSH` *head*(*regList*)	`PUSH` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ `treeCG(`p_{n-2}`,` *regList*`);` `PUSH` *head*(*regList*)	`SUBTRACT SP,SP,#localSz` "Push" SP for local vars	13
	...	`MV FP,SP` Set the **callee's** FP	14
6	if $a > 0$ `treeCG(`p_a`,` $[A_a]$ + *regList*`))`	function body logic	15
7	if $a-1 > 0$ `treeCG(`p_{a-1}`,` $[A_{a-1}]$ + *regList*`))`	`LOAD W0,@??(FP); STORE @returnOffset(FP),W0`	16
	...	`ADD SP,SP,#localSz` "Pop" off the callee's locals	17
8	if $a \geq 1$ `treeCG(`p_1`,` $[A_1]$ + *regList*`))`	`POP` R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		`POP RA` restore the RA from stack (optional?)	19
9	`CALL @function` $\Rightarrow$	`POP FP` if `FP` $\in \mathcal{E}$ restore the **caller's** FP	20
22	`ADD SP,SP,`$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ `RETURN`	21
23	`POP` *head*(*regList*)		
24	`POP` R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	17	

$$returnOffset = \overbrace{paramSz}^{3\text{–}5} + \overbrace{2 \cdot wordSz}^{\text{(maybe, 10–11)}} + \overbrace{Sz(\mathcal{E} \cap RegsCalleeMightUse)}^{12} + \overbrace{localSz}^{13}$$

How much pushing happened for steps 3–13?

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	18	

Step 17 in the callee.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	20	

Step 19 in the callee.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	21	

Step 20 in the callee.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	22	

Step 21 in the callee $\Rightarrow$ step 22 in caller.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP, $\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**	23	

Oh yeah! We're in TREECG still! Pop the function result from the stack to *head*(*regList*).

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Finally, we restore our saved registers and continue generating instructions in `treeCG`.

CALLCG(*functNode*, *regList*, *paramRegs*)

Seq	Caller	Callee	Seq
1	PUSH R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$		
2	Push SP for return value slot	PUSH FP if FP $\in \mathcal{E}$ store the **caller's** FP	10
3	if $n > a$ treeCG(p_n, *regList*); PUSH *head*(*regList*)	PUSH RA store the **callee's** RA on stack (optional?)	11
4	if $n-1 > a$ treeCG(p_{n-1}, *regList*); PUSH *head*(*regList*)	PUSH R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$	12
5	if $n-2 > a$ treeCG(p_{n-2}, *regList*); PUSH *head*(*regList*)	SUBTRACT SP,SP,#localSz "Push" SP for local vars	13
	...	MV FP,SP Set the **callee's** FP	14
6	if $a > 0$ treeCG(p_a, $[A_a]$ + *regList*))	function body logic	15
7	if $a-1 > 0$ treeCG(p_{a-1}, $[A_{a-1}]$ + *regList*))	LOAD W0,@??(FP); STORE @returnOffset(FP),W0	16
	...	ADD SP,SP,#localSz "Pop" off the callee's locals	17
8	if $a \geq 1$ treeCG(p_1, $[A_1]$ + *regList*))	POP R_x for all $R_x \in (\mathcal{E} \cap RegsCalleeMightUse)$ **Reverse!**	18
		POP RA restore the RA from stack (optional?)	19
9	CALL @function $\Rightarrow$	POP FP if FP $\in \mathcal{E}$ restore the **caller's** FP	20
22	ADD SP,SP,$\sum_{a+1}^{p_n} Sz(p_i)$ "Pop" the stack args off	$\Leftarrow$ RETURN	21
23	POP *head*(*regList*)		
24	POP R_x for all $R_x \in (\mathcal{R} \cap CallerRegsInUse)$ **Reverse!**		

Caller steps 1–8: *function call* ***prologue***, 22–24 *function call* ***epilogue***.
Callee steps 10–14: *function* ***prologue***, 16–20 *function* ***epilogue***.