COMPILER CONSTRUCTION · WEEK 7 · STATION III

Procedures

Calls, the calling convention, the call stack, recursion — and where the halting problem lives in a compiler.

TWO PERSPECTIVES

A procedure is an abstraction for the programmer and an allocation for the machine.

Abstraction. Code gets a name and can be reused from anywhere, with parameters. That is structured programming, and it is the reason twelve thousand lines are readable.

Allocation. Every call allocates memory for its parameters, its return address, its local variables — on a stack, dynamically, and frees it on return. Procedures are also the most efficient memory management technology ever invented, because deallocation is in reverse order of allocation and costs one addition.

```
procedure = ( type | "void" ) identifier "(" [ variable
( ";" | "{" { variable ";" } { statement } "}" ) .
call = identifier "(" [ expression { "," expression } ]
return = "return" [ expression ] .
```

CONTROL FLOW AND DATA FLOW

Caller and callee agree on a convention, or nothing works.

Control. The caller jumps to the callee with `jal ra,…`, which saves the return address in `ra`. The callee returns with `jalr zero,0(ra)`. Between them, the callee may call others, so `ra` must be saved somewhere — the stack.

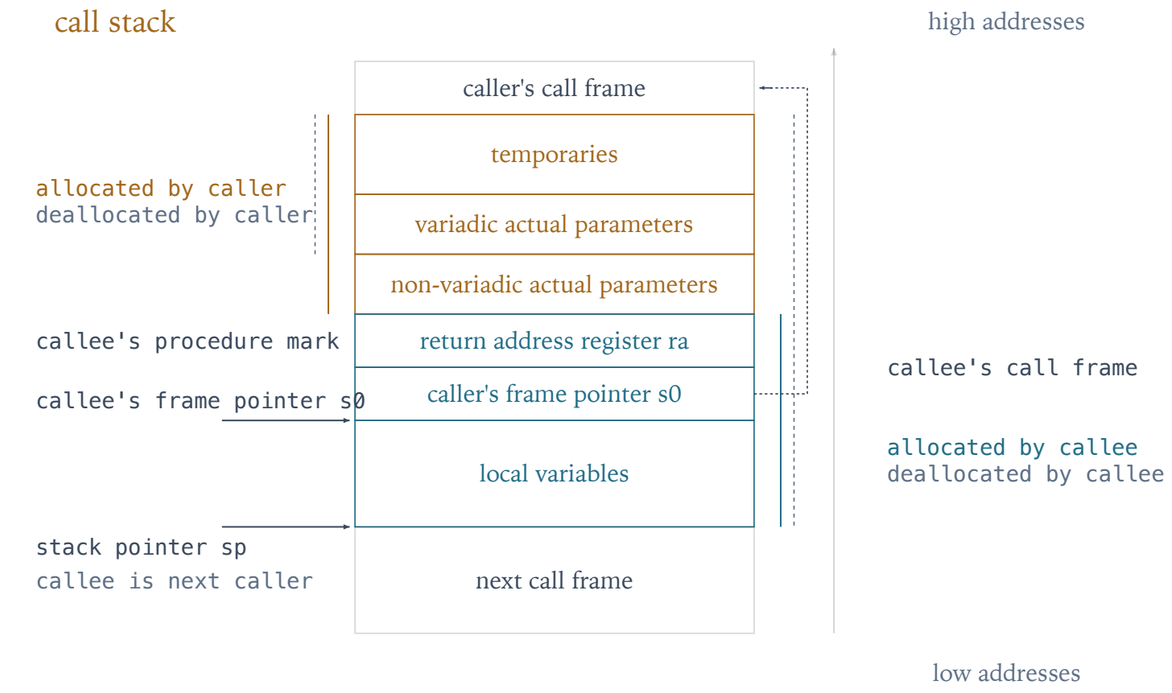
Data. The caller evaluates the actual parameters and pushes them; the callee reads them relative to its frame pointer `s0`. The return value comes back in `a0`. Who pops what is part of the convention: the caller pops the parameters it pushed, the callee pops its own locals.

WHY A CONVENTION

The caller and the callee may be compiled in different files, by different compilers, years apart. Nothing in the language says where a parameter goes. The convention does, and it is why selfie's binaries and gcc's can call each other's code — and why getting one detail wrong corrupts memory silently.

THE FRAME

What the caller allocates, what the callee allocates, and the two pointers that delimit it.



a call frame on the stack

```
// procedure_prologue
addi sp,sp,-8 ; sd ra,0(sp)
addi sp,sp,-8 ; sd s0,0(sp)
addi s0,sp,0
addi sp,sp,-locals
// procedure_epilogue: the reverse, then
jalr zero,0(ra)
```

Prologue: save the return address, save the caller's frame pointer, set the callee's, make room for locals. Epilogue: undo in reverse and return. Parameters sit above the frame, locals below it, both addressed as offsets from `s0` — which is why the frame pointer exists: `sp` moves during expression evaluation, `s0` does not.

RECURSION

A procedure that calls itself needs nothing new. The stack was already there.

```
uint64_t factorial(uint64_t n) {
if (n <= 1)
return 1;
else
return n * factorial(n - 1);
}
```

Each call gets its own frame with its own `n`; the frames stack up until `n` is 1 and unwind multiplying on the way back. The code is emitted once. The frames exist only at runtime, as many as the input demands.

And that is the second way a C* program fails to halt. Last week's while loop; this week's `factorial(n + 1)`. Unbounded recursion is unbounded looping with memory attached — and the machine's memory is finite, so it ends in a stack overflow, which the compiler cannot predict either.

Assignments, loops, procedures: the whole of C* is now on the table. The parser proved every text; the code generator constructed every meaning; and the meaning is now something no analysis of the text fully determines.

IN SELFIE

Bootstrapping the builtins: `malloc`, `exit`, `read`, `write`, `open`.

Selfie uses no library. Five procedures are declared but never defined in `selfie.c`. When gcc compiles selfie, gcc supplies them as builtins. When selfie compiles selfie, `starc` supplies them as system call wrappers: a procedure whose body is an `ecall`, obeying the calling convention on one side and the system call convention on the other.

The wrapper is emitted by `emit_exit` and unwrapped by `implement_exit`, which follows it in the source. The boundary between a program and its operating system, written twice in one file, next to each other.

BOOT LEVELS

Boot level 0: selfie compiled by gcc, running on hardware.
Level 1: selfie compiled by selfie, running on mipster.
Level 2: selfie compiled by that, on mipster on mipster.
The number of > signs in every output line says which.
Next week is about what changes between the levels, and what does not.

ASSIGNMENTS

for-loop and lazy-evaluation.

FOR-LOOP

Grammar: C's `for ( init ; condition ; update ) body`, each part optional. Code: the update is compiled *after* the body but written before it — so the fixups of week 6, once more, with a jump over the update on entry.

LAZY-EVALUATION

Make && and || short-circuit: if the left operand decides, the right is not evaluated. A branch after the left operand, fixed up after the right; the value normalised to 0 or 1 either way. Watch the temporaries: both paths must leave exactly one.

1. Grammar first, in grammar.md; note that the for's parts are statements and expressions you already compile.
2. Draw the emitted code for a for loop with all four parts before writing `compile_for`.
3. For lazy evaluation, draw the two paths and mark where the temporary is allocated and freed on each.
4. `./grader/self.py for-loop lazy-evaluation self-compile`. Then use a for loop in selfie.c and self-compile again.