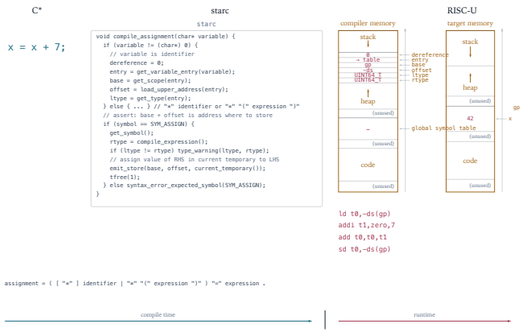
COMPILER CONSTRUCTION · WEEK 6 · STATION III

Statements

Assignments, loops, conditionals — and why while plus assignment is already universal.

ASSIGNMENTS

An rvalue becomes an lvalue: compute, then store.



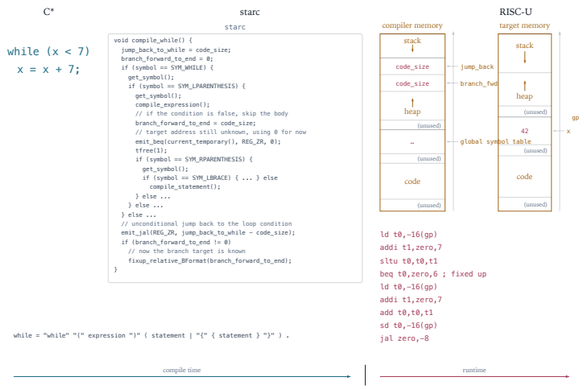
`x = x + 7` · `ld`, `addi`, `add`, `sd`

The only right-associative binary operator in C*: `x = x + 7` is `x = (x + 7)`. Semantics: compute the right-hand side's value, a temporary rvalue; then store it at the left-hand side's address, a persistent lvalue. Before and after — assignment is the reason a program has a *state*.

`compile_assignment` gets the variable already parsed, because of the lookahead that distinguishes assignments from calls. It finds the address from the table, compiles the expression into the current temporary, checks the types, and emits one `sd`. Then frees the temporary: assertion, zero registers allocated.

LOOPS

A branch forward over the body, a jump back to the condition — and a fixup.



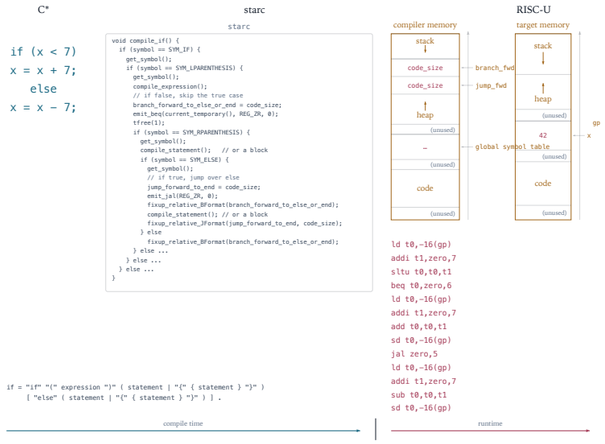

```
while (x < 7) x = x + 7;
```

Remember where the condition starts. Compile it into a temporary. Emit a `beq` that skips the body if the temporary is zero — but the body has not been compiled yet, so the branch distance is unknown. Emit it with 0 and remember its address.

Compile the body. Emit a `jal` back to the condition, whose distance is known. Then *fix up* the branch: overwrite its immediate with the distance to here. Straight code generation's one exception, and it is a write into code that was already emitted — code as data, at compile time.

CONDITIONALS

Branch to else on false, jump over else on true, fix up both.




```
if (x < 7) x = x + 7; else x = x - 7;
```

Same shape with one more fixup: the `beq` over the true branch is fixed when the else begins, and the `jal` over the else branch is fixed when the else ends. Without an else, only the first.

Ten times more ifs than whiles in selfie's source — the profile says 956 to 92 — and yet the if is redundant: every conditional can be written as a loop, not the other way round.

COMPOSITIONALITY

The grammar says what may be composed. The code generator makes the composition mean what its parts mean.

Any expression may be a loop condition; any statement, or block of them, a body. That is composability, and the grammar decides it.

Compositionality is more: the meaning of `while (x < 7) x = x + 7;` is determined by the meaning of `x < 7` and of `x = x + 7` and by the rule for while, and by nothing else. The code for the condition and the body is exactly the code they would get on their own, sandwiched between a branch and a jump.

WHY IT MATTERS

Compositionality is what makes a compiler four thousand lines instead of four million: each `compile_X` knows only its rule and the invariants — a value in the current temporary, zero temporaries at a statement boundary — and trusts the rest. It is also what a proof about a program is built on.

THEOREM

Assignment and while are universal. Nothing else is needed.

With assignments and while loops over unbounded integers you can write anything any programming language can express. Conditionals, procedures, everything else is convenience. That is what Turing-complete means for a language, and C* crossed the line on this deck.

Which is also the moment the compiler lost the ability to know what its output does. Up to expressions, everything terminated by construction. With one while loop, the halting problem is in the language — and Rice's theorem applies to every property of behaviour from here on.

```
uint64_t c;
uint64_t main() {
c = 1;
while (c != 0) c = c + 2;
return c;
}
```

The compiler compiles this happily and correctly. It cannot know that it never stops — you can, by parity and wrap-around. The gift and the limit arrived in the same week.

ASSIGNMENTS

bitwise-and-or-not and logical-and-or-not.

BITWISE-AND-OR-NOT

Grammar: &, |, ~ with C's precedence. ISA: `and`, `or`, `xori` in riscu.md, encoded, decoded, emulated. Code generation: the binary operators as in week 5; not as `xori` with −1. Only grammar.md, riscu.md and selfie.c.

LOGICAL-AND-OR-NOT

Grammar: &&, ||, ! with C's precedence, below comparisons. Code generation with what RISC-U has: `sltu` against zero normalises any value to 0 or 1, and then and, or, not are arithmetic. Both operands evaluated — lazy evaluation is next week's assignment, and it needs this week's fixups.

1. Grammar first, both times: where in the precedence ladder does each operator go?
2. For the bitwise ones, the ISA before the emulator before the code generator, as in week 5.
3. For the logical ones, write the RISC-U sequence for `a && b` on paper before coding, and count temporaries.
4. `./grader/self.py bitwise-and-or-not`, `logical-and-or-not`, and `self-compile`.