COMPILER CONSTRUCTION · WEEK 4 · STATION III

Symbols and Types

The symbol table, what a type is — and why type checking is proof checking while almost everything else about meaning is not.

VARIABLES

A declaration is an entry in a table and a word in memory.

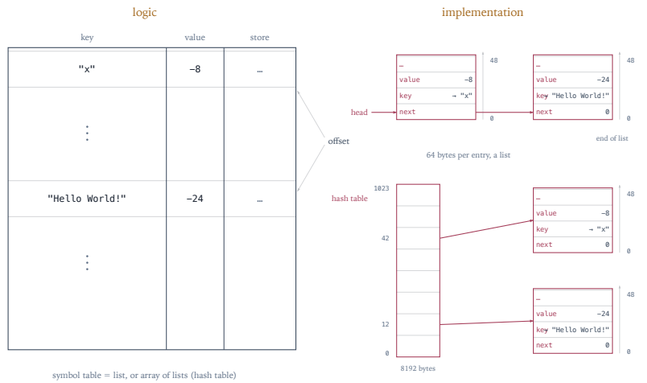

```
uint64_t x = 42;
```

`compile_cstar` sees a type and an identifier not followed by a parenthesis: a global variable. It creates a symbol table entry with the name, the type, the line number, and an address — the next free word in the data segment, negative relative to `gp` — and emits the initial value as a data word.

No code is generated. A declaration is a promise about the future: whenever `x` appears again, this is where it lives and this is what it is. The table is where the promise is kept.

THE SYMBOL TABLE

A key-value store. Logic first, implementation second.



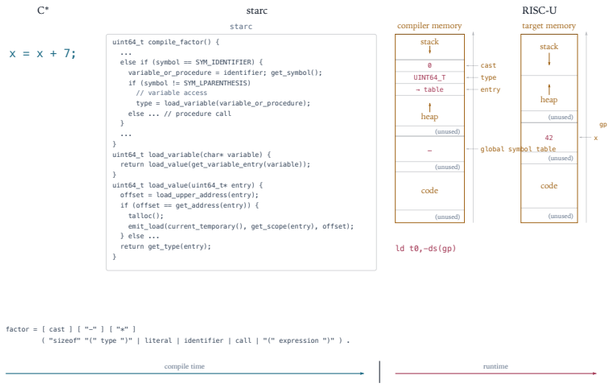
a list of entries · or an array of lists, by a hash of the key

Logic. Two operations: create an entry for a symbol with its attributes, and look a symbol up. That is the API, and it is all the parser needs to know. What happens on a duplicate, and whether to search before creating, are the subtle parts, and they are logic too.

Implementation. A linked list of entries, each a few words on the heap with a pointer to the next — or, for speed, an array of such lists indexed by a hash of the name. Selfie has both. Lists versus arrays is the only choice there ever is: point explicitly, or lay out contiguously.

USE

Using a variable: look it up, and load it relative to gp.



x in an expression · ld t0,–ds(gp)

`compile_factor` sees an identifier not followed by a parenthesis and calls `load_variable`, which looks the name up, takes the address from the entry, and emits one load: the value into a fresh temporary. The type in the entry is returned as the type of the factor.

If the lookup fails, that is a syntax error: *undeclared identifier*. Notice that this is decided — the table is finite and the name is a string — and it is the first thing the compiler knows that the parser alone did not.

TYPES

A type is a label the compiler attaches to notation. Checking labels is decidable.

C* has two: `uint64_t` and `uint64_t*`. Every literal, variable, procedure and expression gets one, as a grammar attribute computed bottom-up: a literal's from the scanner, a variable's from the table, a term's from its factors.

Where two meet — an assignment, an operator, a call — the compiler compares the labels. In C* a mismatch is a warning, because C* is permissive and casts exist; in stricter languages it is an error. Either way the compiler *decided* it, in finite time, by looking at labels.

TYPE CHECKING IS PROOF CHECKING

A typing derivation is a proof, in a small formal system, that the program will not add a pointer to a pointer. Sound: if it type checks, that mistake cannot happen. Incomplete: many fine programs are rejected. That trade is chosen, and it is Rice's theorem answered for one property.

```
$ ./selfie -c warn.c
./selfie: warning in warn.c in line 1: type mismatch, uint64_t* expected but uint64_t found
```

RICE'S LINE, INSIDE THE COMPILER

What the compiler decides. What it only warns about. What it cannot say.

DECIDED

Does it parse. Is every identifier declared. Do the types match. How many procedures, how many while loops. All about the text, all answered, always.

APPROXIMATED

Will this pointer arithmetic go wrong. Is this cast meaningful. Is this variable ever used. The compiler says something sound and incomplete, or stays silent, by design.

NEVER

Does it halt. Does it divide by zero. Is it equivalent to the previous version. Is it correct. Rice, 1953: no compiler, no tool, no method, for all programs.

Every assignment in this class extends the first column. Week 9 shows how optimisations live in the second. Week 12 shows the best anyone can do in the third, within a bound.

ASSIGNMENT

`bitwise-shift-compilation`: << and >> in the grammar and the parser.

1. Specify. Extend `grammar.md` with the shift operators, with C's precedence: below + and −, above the comparisons. That is a new rule between `arithmetic` and `expression`.
2. Scan. Two new symbols, each two characters, each with a lookahead of one after < and > — which already need one for <= and >=.
3. Parse. One new `compile_X` for the new rule, called from where `compile_arithmetic` used to be called. Types: both operands `uint64_t`; warn otherwise.
4. Check. `./grader/self.py bitwise-shift-compilation`. Code generation is next week's assignment; for now the parser must accept and type the operators.

RULES

Only grammar.md and selfie.c. No warnings. Self-compile must still pass, which it will, since selfie.c does not use the operators yet. Submit something by the deadline.