COMPILER CONSTRUCTION · WEEK 2 · STATIONS I AND II

The Scanner

Regular languages, finite state machines, and characters into symbols — a finite machine deciding a countable set.

METHOD

Specification. Modelling. Implementation. In that order, every time.

1 · SPECIFICATION

Which sequences of characters denote an integer literal? A regular expression in EBNF: `integer = digit { digit } .` That is the whole specification, and it is a piece of notation.

2 · MODELLING

How does a machine check it? A finite state machine: circles, arrows labelled with characters, a start, and accepting states. The model is what the code will be measured against.

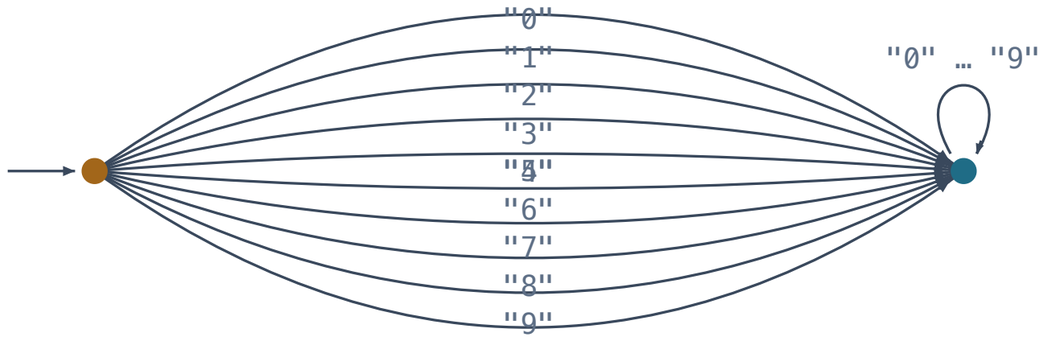
3 · IMPLEMENTATION

A C* procedure that walks the machine and, on the way, computes the value: 85 from the characters '8' and '5'. Efficient, and provably the same as the model.

The theme repeats for character literals, strings, identifiers, and then for every statement and expression in the language, just less explicitly. Learn it once here.

SPECIFICATION AND MODEL

A regular expression is a grammar in one rule. A finite state machine decides it.




```
integer = digit { digit } .
digit   = "0" | … | "9" .
```

integer = digit { digit } .

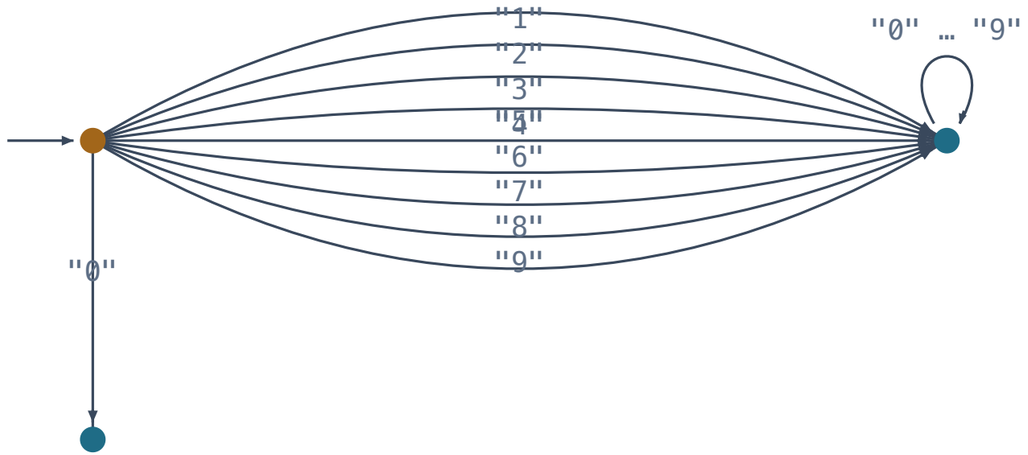
Substitute `digit` into `integer` and one rule remains: that is what regular means. Every regular expression has a finite state machine, and every finite state machine has a regular expression. Kleene, 1956.

The machine reads one character at a time and remembers only which circle it is in. Start on a digit, then loop on digits, accepting all the while. No stack, no counter, no memory of what came before.

And the machine *decides*: for every finite input it stops and says yes or no. Everything the scanner does is on the decidable side of Rice's line, because it is about the text and nothing else.

A BUG IN THE SPECIFICATION

007 is not an integer literal. The grammar said it was.




```
integer        = "0" | non_zero_digit { digit } .
non_zero_digit = "1" | … | "9" .
digit          = "0" | non_zero_digit .
```

```
integer = "0" | non_zero_digit { digit } .
```

C forbids leading zeros in decimal literals. The first grammar allowed them. So the specification had a bug, found by the model — and fixed in the specification, not in the code.

Two accepting states now: one for the single zero, one for everything else. Still regular, still one machine.

This is the class in miniature: a mistake in notation is a mistake, and it is caught where notation is exact. Selfie's scanner implements this machine.

IMPLEMENTATION

Walking the machine, and computing the value on the way.

scanning integer literals

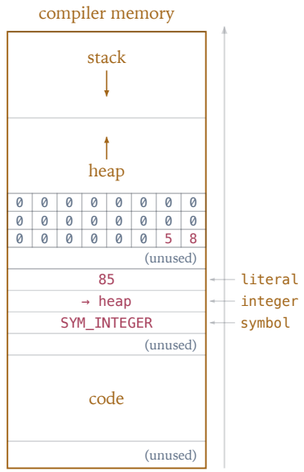
example:

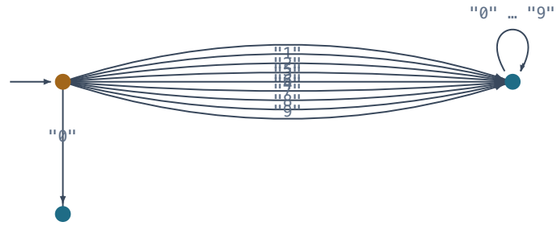
85

two digits, then a NULL

starc

```
} else if (is_digit(character)) {
  if (character == '0') {
    get_character();
    literal = 0;
  } else {
    integer = string_alloc(MAX_INTEGER_LENGTH);
    i = 0;
    while (is_digit(character)) {
      store_character(integer, i, character);
      i = i + 1;
      get_character();
    }
    store_character(integer, i, 0); // null-terminated
    literal = atoi(integer);
  }
  symbol = SYM_INTEGER;
```






```
integer       = "0" | non_zero_digit { digit } .
non_zero_digit = "1" | … | "9" .
digit         = "0" | non_zero_digit .
```

get_symbol, the case for digits · what lands in compiler memory

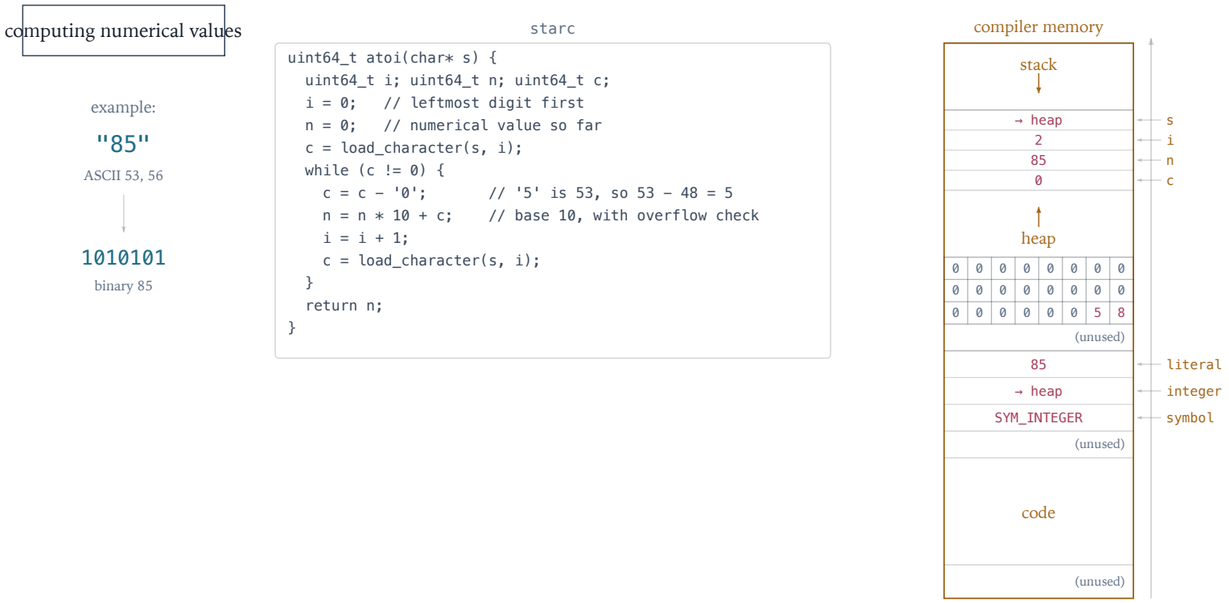
The digit branch of `get_symbol`: if the character is '0', it is the literal 0 and we are done. Otherwise allocate a string, store digits while there are digits, terminate with a NULL, convert with `atoi`, and set the symbol to `SYM_INTEGER`.

On the right, what it leaves behind: the characters '8' and '5' on the heap, the value 85 in the global `literal`, and the symbol in `symbol`. The parser reads those three globals and never sees a character.

The while loop is the self-loop of the machine. The if is the '0' branch. The code is the model, transcribed.

VALUES

atoi: from the characters of a number to the number.



"85" → 85, leftmost digit first

The character '5' is the byte 53. Subtract 48, the code of '0', and it is the digit 5. Multiply what you have so far by ten and add it. Leftmost digit first, so 8 becomes 80 becomes 85.

And check for wrap-around: a literal with twenty digits does not fit in 64 bits, and the machine will not tell you. The scanner must.

Notation to meaning, in a loop: the string is syntax, the value is what it denotes, and this procedure is the semantics of integer literals.

THE REST OF THE ALPHABET

Characters, strings, identifiers: the same machine, three more times.

scanning string literals

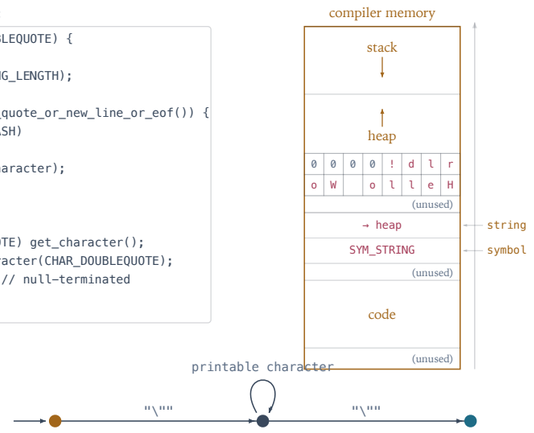
example:

"Hello World!"

twelve characters, then a NULL

starc

```
} else if (character == CHAR_DOUBLEQUOTE) {
  get_character();
  string = string_alloc(MAX_STRING_LENGTH);
  i = 0;
  while (is_character_not_double_quote_or_new_line_or_eof()) {
    if (character == CHAR_BACKSLASH)
      handle_escape_sequence();
    store_character(string, i, character);
    i = i + 1;
    get_character();
  }
  if (character == CHAR_DOUBLEQUOTE) get_character();
  else syntax_error_expected_character(CHAR_DOUBLEQUOTE);
  store_character(string, i, 0); // null-terminated
  symbol = SYM_STRING;
```




```
string = "\"" { printable_character } "\"" .
```

" { printable_character } "

scanning identifiers

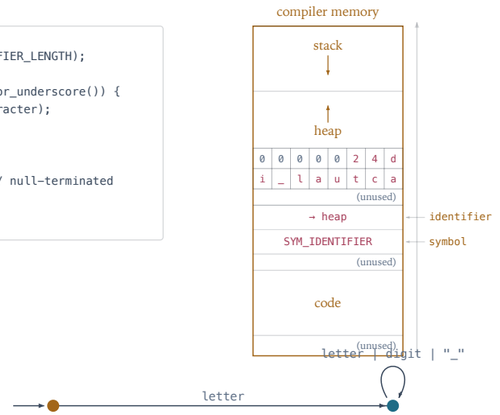
example:

actual_id42

a letter, then letters, digits, underscores

starc

```
if (is_letter(character)) {
  identifier = string_alloc(MAX_IDENTIFIER_LENGTH);
  i = 0;
  while (is_character_letter_or_digit_or_underscore()) {
    store_character(identifier, i, character);
    i = i + 1;
    get_character();
  }
  store_character(identifier, i, 0); // null-terminated
  symbol = identifier_or_keyword();
} else if (is_digit(character)) {
```

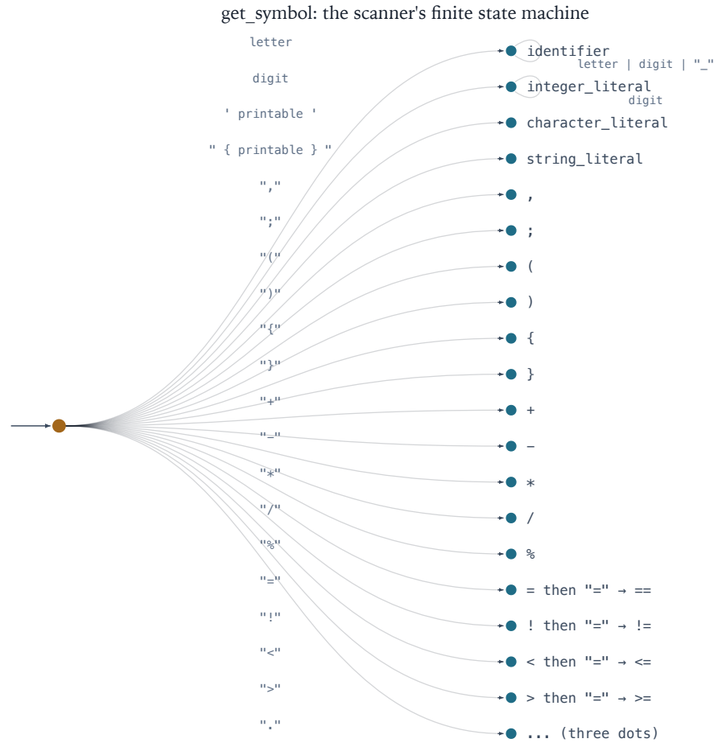



```
identifier = letter { letter | digit | "_" } .
letter     = "a" | … | "z" | "A" | … | "Z" .
digit      = "0" | … | "9" .
```

letter { letter | digit | "_" }

THE WHOLE SCANNER

22 symbols. One start state. Lookahead of one, at most.



get_symbol: one transition per class of symbol

The language of all C* symbols is regular: gather them into one EBNF rule and substitute until only terminals remain. It terminates, so it is regular, so one finite state machine scans all of C*.

Almost every symbol begins with a unique character: a digit, a quote, a letter, a bracket. Only = and ==, < and <=, > and >=, != need to see one more character. That is a lookahead of one, and it is designed in, not found.

Which is why scanning is fast for the machine — and, not by accident, for the human reading the code.

STATION I AND II

Where the scanner sits on the axis.

Countable. The set of all C* symbol sequences is a set of finite strings: listable, numbered. The scanner is a decider for it, and a decider always answers.

Finite. The scanner has no memory but its state, so it cannot count. It cannot match parentheses, and it does not try. That is the parser's job, next week, and it is exactly the difference between a finite state machine and a machine with a stack.

TESTING THE SCANNER

Selfie's own source is 365,784 characters and 51,329 symbols, and it is scanned by the scanner it contains. That is a test with one very large input — and the introduction class said what one input shows. The grader adds a few hundred more. Neither shows absence.

ASSIGNMENT

`hex-literal`: hexadecimal integer literals.

1. Specify. Extend the C* grammar in `grammar.md` with hexadecimal integer literals: `0x` followed by hex digits, both cases. Write the regular expression first.
2. Model. Draw the finite state machine. Where does it branch off the integer machine? What does '0' followed by 'x' need to do that the current '0' branch does not?
3. Implement. Extend `get_symbol` and the value computation. The value of `0x55` is 85, and `0xFFFFFFFFFFFFFFFF` must not wrap silently.
4. Check. `./grader/self.py hex-literal`, and `self-compile`. Then use a hex literal somewhere in selfie.c itself and self-compile again.

```
$ ./grader/self.py hex-literal
…
grade for hex-literal: 2
```

RULES

Do not modify any files other than grammar.md and selfie.c. No compiler warnings. Submit by the deadline even if unfinished — a submission with self-grade 5 beats no submission.