Lexical analysis

Christopher League[*]

27 January 2016

- Lexical analyzer = lexer = scanner = tokenizer. First phase of compilation.
- It converts a stream of characters into a stream of 'tokens'.
- A token (= lexeme) is an atomic unit of syntax. It includes things like identifiers, numbers, keywords, punctuation, and strings.
- Here's an example of how a bit of C/C++/Java code would be tokenized:

```
for(int i = 0; i < source.length(); i++) {
```

  - `FOR` (each keyword is usually a distinct token)
  - `LPAREN`
  - `ID(int)` (built-in types are usually not distinct tokens, we just record this as an identifier)
  - `ID(i)` (also an identifier; parser will distinguish between the type and variable name in a declaration)
  - `EQ`
  - `NUM(0)`
  - `SEMI`
  - `ID(i)`
  - `LT` or maybe `OP(<)` (sometimes it's more convenient to group together all kinds of operators under one token type, `OP`)
  - `ID(source)`
  - `DOT`
  - `ID(length)`
  - `LPAREN`
  - `RPAREN`
  - `SEMI`
  - `ID(i)`
  - `OP(++)`
  - `RPAREN`
  - `LBRACE`
- The lexer **doesn't** care about making sure all the parts of the `for` loop are present, that parentheses and braces match, or that variables are declared. Those are the job of the parser and type-checker (syntactic and semantic analysis).
- Tokens may also record the position (line and column numbers) in the source where they were found. This helps pinpoint error messages later in compilation.

[*]Copyright 2016, some rights reserved (CC by-sa)

- Lexers for most languages discard white-space right away, so there is no token for spaces. (Exceptions might be languages like Python or Haskell where indentation is significant to the syntax.)
- Lexers for most languages also discard comments, because they're not needed any further by the compiler. (Exceptions might be languages which type-check or evaluate expressions within the documentation.)
- Sometimes we have to "look ahead" further into the source before we know what token to emit. A very common way this happens is when some tokens are prefixes of others, such as the keyword `for` and the identifier `form`. While looking at the `f` we don't know yet whether this will produce `FOR` or `ID(form)`. Similarly with operators that overlap like < and <=.
- Lexical rules can be categorized by the number of characters we may need to "look ahead" in order to resolve such ambiguities.
  - A language that is `LL(1)` only needs to consider one character ahead.
  - A language that has to disambiguate identifiers and keywords might be `LL(6)` because, for example, the longest keyword is 6 characters.
  - We can use `LL(k)` to refer to languages that need a *constant* look-ahead, without having to specify what it is.
  - Finally `LL(*)` means an *unbounded* look-ahead: we might have to look arbitrarily far into the source.
- Pattern 2 in the book, *Language Implementation Patterns,* is about implementing a so-called **recursive-descent** lexer for an `LL(1)` language.
- Following this technique, I implemented a `lexdemo` project to tokenize expressions in a "list language" — see the `README.md` for examples. In the `src` directory, the classes `Token` and `ListLexer` are both well-documented, so you can learn more about the technique by reading the code and comments.
- As lexers and parsers get more complicated, we use tools called lexer generators to produce them. The generator takes a specification (often using regular expressions) of the lexical structure of the language, and outputs the code to tokenize it. Examples of lexer generators are Lex, Flex, and ANTLR.
- We'll learn to use ANTLR a little later, but for now we'll hand-code an `LL(1)` lexer in Assignment 2.