Compilers and computer architecture

3. From strings to ASTs: lexing and regular expressions

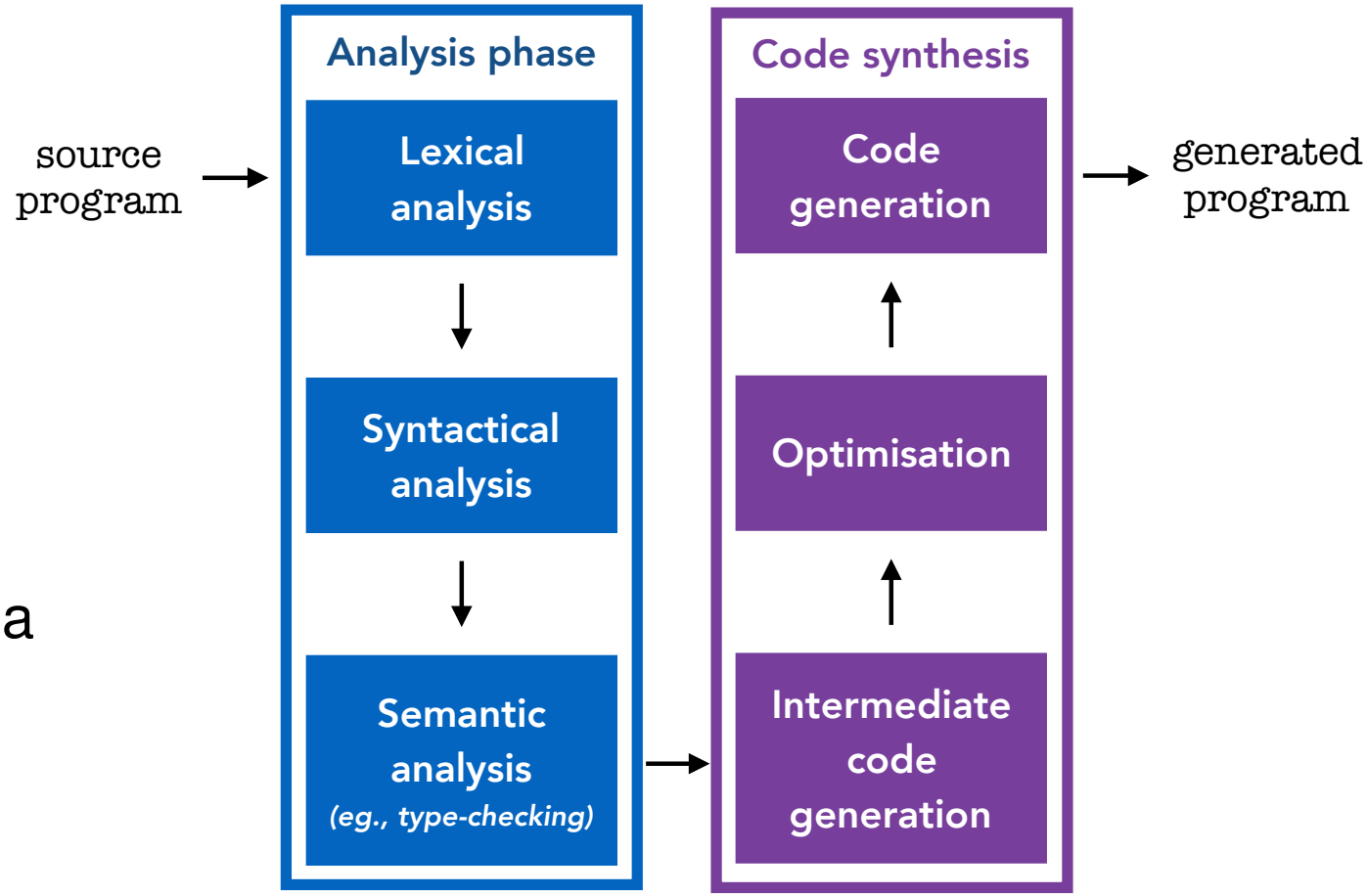
Ian Mackie

Autumn 2022

Plan for the next 9 weeks

Remember the shape of compilers:

We're studying the first phase: **lexing** — in which we divide up a text stream into units of a compute program.



Summary of last lecture

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g., whitespace, syntax of keywords, ...)
- Give a rough classification (simplification) of the input to simplify the next stage (parsing). In detail:
  - Identify the lexical structure and tokens of the language.
  - Partition the input string into small units (tokens) used by the parser.

Lexing does a left-to-right scan of the input string, sometimes with lookahead.

Tasks for the lexical stage

The point of the lexical phase is:

1. **Description** of the lexical structure of the language (determine token classes).

   We use **regular expressions** for this purpose.

2. From the description in (1) above, derive a scanning **algorithm**, called a **lexer**, that determines the token class of each lexical unit.

   We use FSAs (**finite state automata**) for this.

Regular expressions

Regular expressions (REs) are a way of precisely describing the set of “lexemes” (strings) associated with each token.

Informally, with REs we can say something like this:

An integer is a non-empty string of digits.

But we want to be more precise than using natural language.

Preparation for regular expressions

An **alphabet** is a set of characters. Examples include $\{1, 4, 7\}$, as well as $\{a, b, c, ..., z\}$.

A **string** over an alphabet A is a finite sequence elements from A. Examples with alphabet $\{a, b, c, ..., z\}$:

> "welcometocompilers"
>
> ""

Question: Is "welcome to compilers" a string over the same alphabet?

Preparation for regular expressions

A **language** over an alphabet A is a set of strings over A.
Examples over $\{a, b, c, ..., z\}$:

- $\emptyset$, the empty language.
- $\{\text{“”}\}$.
- {“welcometocompilers”}.
- {“welcometocompilers”, “dontforgettodotheexercises”}.
- {“welcometocompilers”, “dontforgettodotheexercises”’, “”}.
- The set of all strings of even length over $\{a, b, c, ..., z\}$.
- The set of all strings over $\{x, y\}$ with as many ‘x’s as ‘y’s.

Do you see what’s special about the last two examples? The languages are infinite!

Specifying languages

Finite languages (= consisting of a finite number of strings) can be given by listing all strings.

This is not possible for infinite languages — e.g. the set of all integers (as a language over $\{0, \ldots, 9\}$).

Regular expressions are a mechanism to specify finite and infinite languages.

This is the real point of regular expressions (and other formal accounts of languages like context free languages that we see later): to enable a **precise** description of languages that are too **large** (typically infinite) to enumerate.

The set of all (lexically/syntactically valid) Java/C/Python/Rust ... programs is infinite.

Regular expressions

We describe REs precisely but informally now.

Let A be an alphabet, i.e. a set of characters. We now define two things in parallel:

- The **regular expressions** over A.
- The **language** of each regular expression over A. We denote the language of r.e. R by $\text{lang}(R)$.

Regular expressions

We have 7 (basic) kinds of regular expressions over alphabet A:

- $\emptyset$ (the empty set).
- ε (the empty string).
- '$\mathtt{c}$', for any $\mathtt{c} \in A$ (any single 'letter').
- $R \mid S$, for any regular expressions R and S over A.
- RS, for any regular expressions R and S over A.
- R^*, for any regular expression R over A.
- (R), for any regular expression R over A.

Each specifies a **language**.

Regular expressions

$\emptyset$ and ε: the empty set and the empty string

- $\emptyset$ is a regular expression, denoting the empty set $\{\}$.
  We write lang$(\emptyset) = \{\}$.
- ε is a regular expression, denoting the set $\{""\}$.
  We write lang$(\varepsilon) = \{""\}$.

It's important to remember that $\emptyset$ and ε are different regular expressions, denoting different languages (as we see here).

Regular expressions
Alphabet characters

For each element `c` of an alphabet A, '`c`' is a regular expression over A, denoting the language

$$\text{lang}(\text{`}\texttt{c}\text{'}) = \{\text{``}\texttt{c}\text{''}\}.$$

This is a singleton set consisting of a single 'character'.

Regular expressions
Alternatives

If R and S are regular expressions over a common alphabet A, then $R \mid S$ is also regular expression over A, denoting the language

$$\text{lang}(R \mid S) = \text{lang}(R) \cup \text{lang}(S).$$

You can think of $R \mid S$ as R **or** S, where you can select a string from either one of R or S.

Regular expressions
Concatenation

If R and S are regular expressions over a common alphabet A, then RS (pronounced R **concatenated** with S, or R **then** S) is a regular expression, denoting the language

$$\text{lang}(RS) = \left\{ rs \,\middle|\, r \in \text{lang}(R), s \in \text{lang}(S) \right\}.$$

Here rs is the concatenation of the strings r and s.

Example: if $r =$ “hello” and $s =$ “world”, then rs is “helloworld”.

Regular expressions

Star

The regular expressions presented so far do not, on their own, allow as to define **infinite** languages. **Why?**

The next operator changes this. It can be seen as a simple kind of 'recursion' or 'loop' construct for languages.

If R is a regular expression over an alphabet A, then R^* (pronounced R**-star**) denotes the language

$$\text{lang}(R^*) = \text{lang}(\varepsilon) \cup \text{lang}(R) \cup \text{lang}(RR) \cup \text{lang}(RRR) \cup \cdots$$

In other words:

$$\text{lang}(R^*) = \bigcup_{n \geqslant 0} \text{lang}(\underbrace{RRR \ldots R}_{n})$$

So a string w is in $\text{lang}(R^*)$ exactly when $w = r_1 r_2 \ldots r_n$ for some number n, and strings $r_1, r_2, \ldots \in R$.

Regular expressions
Star

Example: Let $A = \{0, 1, 2, ..., 9\}$, then the language of

$$(\text{`0'} \mid \text{`1'})^*$$

is . . . the set of all binary numbers (allowing leading zeroes).

Whoops — where do these brackets come from in $(\text{`0'} \mid \text{`1'})^*$?

Can we drop them and write `0'} \mid \text{`1'^* ? **No!**

Regular expressions

Brackets / parentheses

If R is a regular expression, then (R) is a regular expression with the same meaning as R, i.e.

$$\text{lang}((R)) = \text{lang}(R).$$

Question. What's the point of this?

Hint.
Does '0' | '1' * denote $\{\text{'0'}, \varepsilon, \text{'1'}, \text{'11'}, \text{'111'}, \text{'1111'}, \text{'11111'}, \ldots\}$,

or $\{\varepsilon, \text{'0'}, \text{'1'}, \text{'00'}, \text{'01'}, \text{'10'}, \text{'11'}, \text{'000'}, \ldots\}$?

There is an "order of operations" in the construction of regular expressions (a priority for tokens!) and parentheses help us to group things at a lower place in the order.

Does this remind you of anything . . . ?

Regular expressions
Precedence rules

To reduce the number of brackets in regular expressions, and to reduce ambiguity, we use the following precedence rules.

- R^* binds more highly than RS —
  i.e., RS^* is equivalent to $R(S^*)$.
- RS binds more highly than $R \mid S$ —
  i.e., $RS \mid T$ is equivalent to $(RS) \mid T$.

So, $QR^*S \mid T$ is equivalent to $(QR^*S) \mid T$,
which is equivalent to $((Q(R^*))S) \mid T$.

Regular expressions: summary

Summary: the regular expressions over an alphabet A are

- $\emptyset$, ε, or 'c' for any c $\in A$.
- $R \mid S$, provided R and S are regular expressions.
- RS, provided R and S are regular expressions.
- R^*, provided R is a regular expression.
- (R), provided R is a regular expression.

Examples of REs

Let $A = \{0, 1\}$. What languages are the following?

- ‘1’ *
  — “any number of 1s”
- (‘0’ | ‘1’)* | ‘1’
  — either any number of (either 0 or 1), or a single 1
- (‘0’ | ‘1’)* ‘1’
  — any number of (either 0 or 1), followed by a single 1
- ‘0’ * | ‘1’ *
  — either any number of 0s, or any number of 1s
- ‘0’ | ‘1’ *
  — either a single zero, or any number of 1s

(where “*any number of* [x]” can include zero [x]-es)

Abbreviations in REs

One often finds abbreviations such as the following.

- We often write 1 instead of ‘1’, a instead of ‘a’, and so on for all elements of the alphabet A. Thus, instead of

  ‘s’ ‘t’ ‘r’ ‘i’ ‘n’ ‘g’

  we write ’string’ (or “string”).
- R^+ stands for RR^* (“one or more of R”).
- $R?$ stands for $R \mid \varepsilon$ (“possibly an R”).
- If there’s a well-understood order on the alphabet, we may specify ranges.
  - For instance, we may write [a–z] instead of a|b|c|...|y|z or [0–9] instead of 0|1|2|3|4|5|6|7|8|9, etc.
  - We may concatenate ranges to take their union: for instance, [a–zA–Z] instead of [a–z] | [A–Z].

Examples of REs

Let $A = \{0, 1\}$.

- What language does $(0 \mid 1)^* \, 00 \, (0 \mid 1)^*$ represent?
  - The set of all binary strings containing 00 as a substring.
- What regular expression over A has exactly the strings of length 4 as language?
  - $(0 \mid 1)(0 \mid 1)(0 \mid 1)(0 \mid 1)$.
- What regular expression has only strings with at most one 0 as language?
  - $1^* \, 0? \, 1^*$.

Now, let $A = \{-, 0, 1, ..., 9\}$.

- What regular expression has only strings representing positive and negative integers as language?
  - $-? \, [1\text{–}9] \, [0\text{–}9]^*$.

Lexical specification using REs

Let us now give a lexical specification of a simple programming language using REs.

- Keywords: `keyw = if | else | for | while`,
  where, e.g., `if` is a shorthand for '`i`' '`f`'.
- Digits: `digit = 0|1|2|3|4|5|6|7|8|9`,
  or `digit = [0–9]` for short.
- Integers: `integer = -?digit+`
  (if we allow leading zeroes in out integers)
- Whitespace: `ws = ' ' | '\n' | '\t'`

  (Question: why would we need to define this in the lexical specification of a programming language?)

Lexical specification using REs

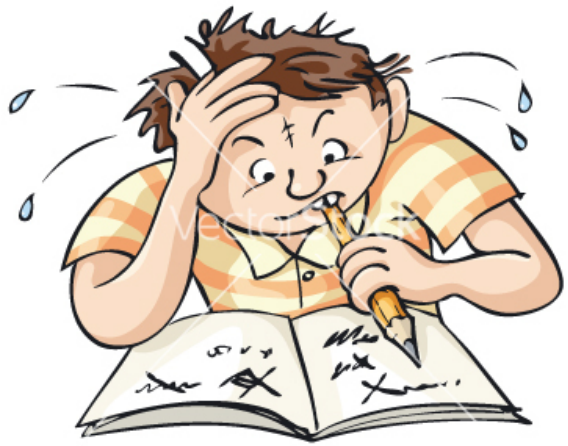
Here is a more complex example: specification of non-negative numbers, such as 234, 3.14159, or 6.2E−14, in a realistic programming language (writing **eps** for ε.)

```
        digit = [0-9]
       digits = digit+
  optFraction = ('.'  digits) | eps
  optExponent = ('E' ('-' | eps) digits) | eps
          num = digits optFraction optExponent
```

Real languages

Java's (JDK 13) lexical spec:
`docs.oracle.com/javase/specs/jls/se13/html/jls-3.html`

Python 3's lexical spec:
`docs.python.org/3/reference/lexical_analysis.html`



Lexical specification using REs

REs are useful and can specify the lexical level of programming languages. We end with two questions:

1. Can all languages be specified with REs? For example, can all **syntactically** valid Java programs, or all arithmetic expressions that have **balanced** parentheses?

   No! We need more general languages. e.g. context free and context sensitive languages. More about this soon.

2. How to decide, given a string s and a regular expression R, if $s \in \text{lang}(R)$?

   Finite State Automata (FSA), as **algorithms** to **decide** the language defined by REs.

The material in the textbooks

- Dragon Book: Chapter 2.6 and Chapter 3.
- Appel, Palsberg: Chapter 2.1 and 2.2
- "Engineering a compiler": Chapter 2.1 and 2.2