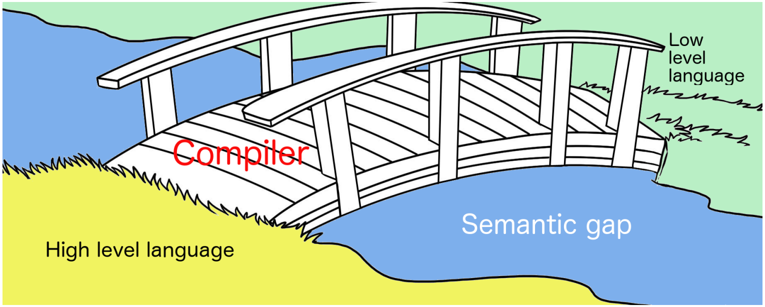
Compilers and computer architecture: From strings to ASTs (1): lexing

Martin Berger [1]

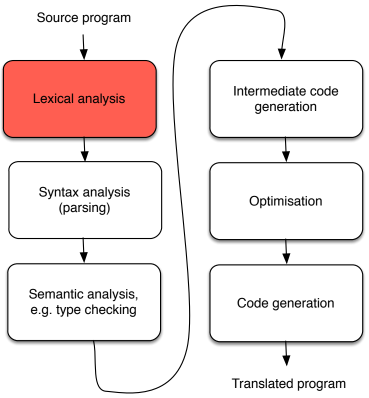
October 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Plan for the next 9 weeks


Source program
Lexical analysis
Syntax analysis (parsing)
Semantic analysis, e.g. type checking
Intermediate code generation
Optimisation
Code generation
Translated program

Remember the shape of compilers?

For the next 9 weeks or so, we will explore this pipeline step-by-step, starting with lexing.

From strings to ASTs

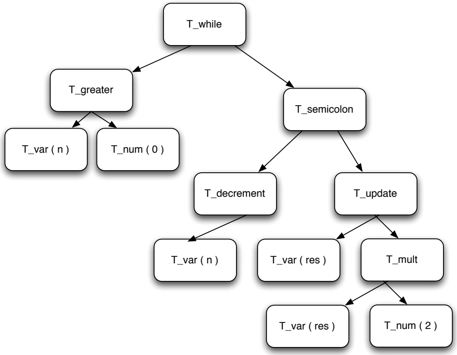
The purposes of the lexing and parsing phase is twofold.

- To convert the input from strings (a representation that is convenient for humans) to an abstract syntax tree (AST), a representation that is convenient for (type-checking and) code generation.
- Check syntactic correctness.

From strings to ASTs

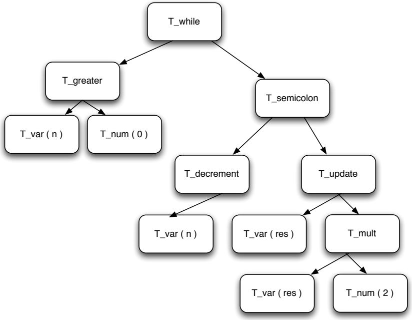
We will later define in details what ASTs are, but for now, a picture says more than 1000 words. We want to go from the representation on the left to that on the right:

```
while( n > 0 ){
     n--;
     res *= 2; }
```



From strings to ASTs

```
while( n > 0 ){
     n--;
     res *= 2; }
```



Why? To make the backend and type-checking much easier and faster: type-checking and code generation need acces to components of the program. E.g. to generate code for `if C then Q else R` we need to generate code for `C`. `Q` and `R`. This is **difficult directly from strings**. The AST is a data structure optimised for making this simple. ASTs use pointers (references) to point to program components. CPUs can maninpulate pointers very efficiently.

From strings to ASTs

It is possible to go in from strings to ASTs in one step (called scannerless parsing). But that is complicated. Most compilers use two steps.

- Lexical analysis (lexing).
- Syntactical analysis (parsing).

Another example of divide-and-conquer.

This will involve a bit of language theory and automata, some of which you have already seen last year in one of the introductory courses. You will also learn about tools for generating lexers and parsers.

Knowing about this, (e.g. regular expressions, context-free languages) is one of the most useful things to know as a programmer, even if you'll never write a compiler: just about every application you'll ever encounter will involve reading or creating formal languages.

Lexical analysis (lexing)

The purpose of lexing is to prepare the input for the parser. This involves several related jobs.

- Removing irrelevant detail (e.g. whitespace, tab-stops, new-lines).
- Split the string into basic parts called tokens. You can think of them as ’atoms’.

Why is using tokens useful?

Lexical analysis (lexing)

Why is using tokens useful?

Because for syntax analysis it's **not** important if a variable is called *n* or *numberOfUsers*, and likewise for *x* vs *y* 3 vs 4 etc.

In other words, the program

```
if n == 3 then x := 0 else x := 1
```

is **syntactically** correct exactly when

```
if numberOfUsers == 666 then y := 0 else z := 171717
```

is syntactically correct.

The precise choice of variable name is important later (for type-checking and for code generation) so we keep the name of the identifier for later phases. But syntactic correctness is decided independently of this detail.

Lexical analysis (lexing)

Another reason why using tokens is useful?

Information hiding / modularisation: for syntax analysis it's **not** important if the syntax for conditionals is `if then else` or `IF THENDO ELSEDO` etc. In other words, the reason why we deem

```
if n == 3 then x := 0 else x := 1
```

to be **syntactically** correct (in one language) is exactly the same as the reason why we deem

```
IF n == 3 THENDO x := 0 ELSEDO x := 1
```

to be **syntactically** correct (in another language).

The precise choice of keywords name is irrelevant in later stages, so let's abstract it by way of a **token** representing concrete keywords. That makes it much easier to change language keywords later, only the lexer needs adaptation.

Lexical analysis (lexing)

Let's look at an analogy. You can classify natural languages sentences according to syntactic function. For example

Synchronised dancing is great!

This can be simplified as follows

$$\overbrace{\text{Synchronised}}^{\textit{Adjective}}\ \underbrace{\text{dancing}}_{\textit{Noun}}\ \overbrace{\text{is}}^{\textit{Verb}}\ \underbrace{\text{great}}_{\textit{Adverb}}\ \overbrace{\text{!}}^{\textit{Punctuation}}$$

Roughly: sentences of the form ...

Adjective Noun Verb Adverb Punctuation

... are **syntactically** correct in English.

Lexical analysis (lexing)

The process of going from string “Synchronised dancing is great!” to list “Adjective Noun Verb Adverb Punctuation” is **lexing**. Let’s break it down.

To work out if an English sentence is syntactically correct, we can do the following:

- Classify every word in terms of its function (tokens), i.e. perform lexical analysis.
- Check if the ordering of tokens is acceptable according to the grammar of English. (This is parsing aka sytnax analysis.)

Question: How do we know what the words are? Answer: We have a concept of **boundary**, typically whitespace.

We do the same with computer languages.

Breaking down this process even further

Lexing/tokenisation can be seen as two separate steps:

- Start with String
- Split string on boundaries into List(String)
- Convert/classify the List(String) into List(Token)

Example:

- String: “Synchronised dancing is   great!”
- List(String): [“Synchronised”, “dancing”, “is”, “great”, “!”]
- List(Token): [Adjective, Noun, Verb, Adverb, Punctuation]

In practise, those steps are often excuted in one go for efficiency.

Question: why bother with this?

Recall that our goal is twofold:

- Produce an AST form the input string, because code generation works with ASTs.
- Check syntactic correctness.

Both are **much** easier using a list of tokens, rather than a string. So tokenisation is a form of simplification (information hiding): it shields the next stage (parsing) from having to deal with irrelevant information.

Question: why bother with this?

In summary, lexing has two beneficial effects.

- It simplifies the next stage, parsing (which checks syntactic correctness and constructs the AST).
- It abstracts the rest of the compilers from the lexical detail of the source language.

Lexical analysis (lexing)

When designing a programming language, or writing a compiler, we need to decide what the basic constituents of programs are, and what the grammatical structure of the language is. There are many different ways of doing this. Let's look at one.

- Keywords (IF, ELSE, WHILE, ...), but not GIRAFFE
- Identifiers (x, i, username, ...), but not _+++17
- Integers (0, 1, 17, -3, ...), but not ...
- Floating point numbers (2.0, 3.1415, -16.993, ...)
- Binary operators (+, *, &&, ||, =, ==, !=, :=, ...)
- Left bracket
- Right bracket
- Token boundaries (whitespace, tab-stops, newlines, semicolon, ...)
- ...

So the tokens are: Keywords, Identifiers, Integers, Floating point numbers, Binary operators, Left bracket, Right bracket, ...

Example

Let’s look a the raw string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

This gives rise to the following token list:

- Keyword: “IF”
- LeftBracket
- Ident: “x”
- Binop: “>”
- Int: “1”
- RightBracket
- Keyword: “THEN”
- Identifier: “z”
- Binop: “:=”
- Unop: “-”
- Float “3.01”

Lexical analysis (lexing): ingredients

Hence, to lex strings we need to decide on the following ingredients.

- Tokens (representing sets of strings).
- Boundaries between tokens.
- Inhabitation of tokens, i.e. decide which strings are classified by what tokens.
- An algorithm that inputs a strings and outputs a token list.
- What to do if we encounter an input that isn't well-formed, i.e. a string that cannot be broken down into a list of tokens.

Example: Tokens and their inhabitation

For example the token class for Identifiers is inhabited by: e.g. non-empty strings of letters, digits, and “_” (underscore), starting with a letter. (This is the language designer’s choice.)

Token class Integers is inhabited by: e.g. non-empty strings of digits.

Keywords: “ELSE”, “IF”, “NEW”, ... each is its own token class.

Whitespace: a non-empty string of blanks, newlines and tab-stops. **Not** inhabiting any token classes, since irrelevant to rest of compilation.

Can you see a problem with this?

Tokens and their inhabitation

Some strings can be described by more than one token, e.g. “THEN” is an identifier and a keyword.

We must have a mechanism to deal with this: e.g. have priorities like: if a string is both a keyword and an identifier, it should be classified as a keyword. In other words, keywords have **priority** over identifiers.

Tokens and their inhabitation

Aside: Some old programming languages let keywords be identifiers, for example in PL/1 the following is valid (or something close to it):

```
if if = then ( if, else ) then else = 1 else else = 3
```

Allowing identifiers also to be keywords is rarely useful, so most modern programming languages prohibit it.

Tokens and their inhabitation

There is a second problem!

What about the string “IFTHEN”? Should it be the tokens “IF” followed by “THEN”, or a single token standing for the identifier “IFTHEN”? Usual answer: use **longest match.**

Tokens and their inhabitation

However, problems still remain. For efficiency, most lexers usually scan their input from left to right, but only once! A keyword might be a proper prefix of an identifier:

```
int formulaLength = 17; for i = 0 to ...
```

So the lexer can only classify strings by **looking ahead**.

How does the lexer know from looking ahead that the “for” in “formulaLength” isn’t a keyword? Answer: because we know what word boundaries are (such as whitespace, semicolon).

Tokens and their inhabitation

Question: how far does the lexer have to look ahead (in Java-like languages) before it can decide whether `for...` is the keyword for or the beginning of an identifier?

The more lookahead required, the less efficient the lexing process. Some old language can require unbounded lookahead. Modern languages require little lookahead (typically 1 character). Bear that in mind when designing a language.

Summary

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g. whitespace, tab stops, 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

Let's rephrase what we've just said in a slightly different language. 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) derive a scanning **algorithm**, called lexer, that determines the token class of each lexical unit. We use FSAs (**finite state automata**) for this.

Regular expressions

Regular expressions are a way of formally (= 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 terse and precise than using natural language.

Aside: Invented in the 1950s to study neurons / neural nets! (Then called “nerve nets”.)

Preparation for regular expressions

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

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

“hellomynameismartin”

""

Question: Is “hello my name is martin” a string over the same alphabet?

Question: what are the strings that you can form over the empty set as 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.
- $\{""\}$.
- $\{"hellomynameismartin"\}$.
- $\{"hellomynameismartin", "hellomynameistom"\}$.
- $\{"hellomynameismartin", "hellomynameistom", ""\}$.
- The set of all strings of even length over $\{a, b, c, ..., z\}$.
- The set of all strings of even length over $\{a, b\}$.
- What is the language of all strings over the alphabet $\{0, 1\}$?

Do you see what's special about the last three 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 language of all integers as a language over $\{0, ..., 9\}$).

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 **terse** 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

Regular expressions are a tool for specifying languages. You can think of them as a “domain specific language” or an ’API’ to specify languages. We will describe them precisely but informally now.

Regular expressions

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

- $\emptyset$.
- ϵ.
- $'c'$ for all $c \in A$.
- $R|R'$.
- RR'.
- R^*.
- (R).

Each specifies a **language**.

Regular expressions (1): $\emptyset$

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

$\emptyset$ is a regular expression, denoting the empty set $\{\}$. Now lang($\emptyset$) $= \{\}$.

Regular expressions (2): ϵ

Let *A* be an alphabet, i.e. a set of characters. The **regular expressions** over *A* are given by the following rules.

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

It's important to realise that $\emptyset$ and ϵ are different regular expressions, denoting different languages.

Regular expressions (3): alphabet characters

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

For each character c from A, $'c'$ is a regular expression, denoting the language

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

Regular expressions (4): alternatives

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

If R and S are regular expression, then $R|S$ is a regular expression, denoting the language

$$\mathrm{lang}(R) \cup \mathrm{lang}(S).$$

You can think of $R|S$ as R **or** S.

Regular expressions (5): concatenation

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

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

$$\{rs|r \in \mathrm{lang}(R), s \in \mathrm{lang}(S)\}.$$

Here rs is the concatenation of the strings r and s. Example: if $r = "hello"$ and $s = "world"$, then rs is $"helloworld"$.

Regular expressions (6): 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.

Regular expressions (6): star

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

If R is a regular expression, then R^* (pronounced R**-star**) is a regular expression, denoting the language

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

In other words

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

So a string w is in $\text{lang}(R^*)$ exactly when some number n exists with $w \in \text{lang}(\underbrace{RRR...R}_{n \text{ times}})$.

Regular expressions (6): star

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

$$('0'|'1')^*$$

is ... the set of all binary numbers.

Whoops, where do these brackets come from in $('0'|'1')^*$?

Can we drop them and write

$$'0'|'1'^*$$

No!

Regular expressions (7): brackets

Let A be an alphabet, i.e. a set of characters. The **regular expressions** over A are given by the following rules.

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

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

Regular expressions: summary

Summary: the regular expressions over an alphabet A are

- $\emptyset$.
- ϵ.
- $'c'$ for all $c \in A$.
- $R|R'$, provided R and R' are regular expressions.
- RR', provided R and R' are regular expressions.
- R^*, provided R is a regular expressions.
- (R), provided R is a regular expressions.

Regular expressions precedence rules

To reduce the number of brackets in regular expressions with assume the following precedence rules.

- R^* binds more highly than RR', i.e. AB^* is short for $A(B^*)$.
- RR' binds more highly than $R|R'$, i.e. $AB|\mathrm{C}$ is short for $(AB)|C$.

So $AB^*C|D$ should be read as $((A(B^*))C)|D$.

Examples of REs

- Alphabet $A = \{0, 1\}$. What language is $'1'^*$?
- Alphabet $A = \{0, 1\}$. What language is $('0'|'1')^*|1$?
- Alphabet $A = \{0, 1\}$. What language is $('0'|'1')^*1$?
- Alphabet $A = \{0, 1\}$. What language is $'0'^*|'1'^*$?
- Alphabet $A = \{0, 1\}$. What language is $'0'|'1'^*$?

Examples of REs

Alphabet $A = \{0, 1\}$. What language is

$$('0'|'1')^{*}'0''0'('0'|'1')^{*}$$

Answer: the set of all binary strings containing 00 as a substring.

Examples of REs

Alphabet $A = \{0, 1\}$. What regular expression over A has exactly the strings of length 4 as language?

Answer: $('0'|'1')('0'|'1')('0'|'1')('0'|'1')$.

Examples of REs

Alphabet $A = \{0, 1\}$. What regular expression has only strings with at most one 0 as language?

Answer: $'1'^*('0'|\epsilon)'1'^*$

Examples of REs

Alphabet $A = \{-, 0, 1, ..., 9\}$. What regular expression has only strings representing positive and negative integers as language?

$$('-'|\epsilon)('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')('0'|...|'9')^*$$

Note that "..." is not part of regular expressions, I was just too lazy to type out all the relevant characters.

Abbreviations in REs

One often finds the following abbreviations.

- We often write 1 instead of $'1'$, a instead of $'a'$ and so on for all elements of the alphabet A. With this convention it makes sense to write e.g. A^*.
- Instead of $'a'' ''s''t''r''i''n''g'$ we write $'a\ string'$ or $''a\ string''$.
- Sometimes $R + S$ is written for $R|S$.
- R^+ stands for RR^*. (Note R^+S is different from $R + S$)
- If there's a 'natural' order on the alphabet we often specify ranges, e.g. $[a - z]$ instead of $a|b|c|...|y|z$ or $[0 - 9]$ instead of $0|1|2|3|4|5|6|7|8|9$, or $[2 - 5]$ for $2|3|4|5$. Etc.
- We write $[a - zA - Z]$ instead of $[a - z]|[A - Z]$
- $R?$ for $R|\epsilon$

Lexical specification using REs

Let us now give a lexical specification of a simple programming language using REs. We start with keywords.

```
"if" | "else" | "for" | "while"
```

Recall that `"if"` is a shorthand for `'i''f'`.

Lexical specification using REs

Next are digits and integers. Digits are:

```
0|1|2|3|4|5|6|7|8|9
```

or simpler

```
[0-9]
```

We want to refer to this RE later, so we name it.

```
digit = [0-9]
```

Lexical specification using REs

Now integers can be written as follows

```
integer = -?digit+
```

This is short for

```
integer =
    ('-' | epsilon)
      ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
      ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')*
```

Abbreviations are helpful for readability (writing `epsilon` for ϵ).

Lexical specification using REs

When specifying the lexical level of programming languages we also need to specify whitespace (why?). Often the following is used.

```
    whitespace = (' ' | '\n' | '\t')+
```

What does '\n' and '\t' mean?

Lexical specification using REs

Here is a more realistic example: specification of numbers in a real programming language. Examples 234, 3.141526 or 6.2E-14

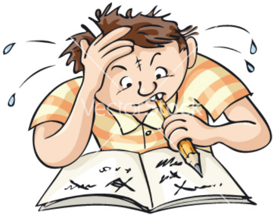
```
digit         = [0-9]
digits        = digit+
optFraction   = ('.'digits) | eps
optExponent   = ( 'E'( '-' | eps ) digits )
              | eps
num           = digits optFraction optExponent
```

(Writing `eps` for `epsilon`.)

Real languages

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

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



Lexical specification using REs

I hope you see that REs are useful and can specify the lexical level of programming languages. Two main questions are left open.

1. Can all languages be specified with REs? E.g. all **syntactically** valid Java programs, or all arithmetic expressions that have **balanced** parentheses? Answer: **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 $s \in lang(R)$? Answer: 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