Handout 5

Whenever you want to design a new programming language or implement a compiler for an existing language, the first task is to fix the basic “words” of the language. For example what are the keywords, or reserved words, of the language, what are permitted identifiers, numbers, expressions and so on. One convenient way to do this is, of course, by using regular expressions.

In this course we want to take a closer look at the WHILE programming language. This is a simple imperative programming language consisting of arithmetic expressions, assignments, if-statements and loops. For example the Fibonacci program can be written in this language as follows:

```
write "Input a number ";
read n;
x := 0;
y := 1;
while n > 0 do {
  temp := y;
  y := x + y;
  x := temp;
  n := n - 1
};
write "Result ";
write y
```

The keywords in this language will be

`while`, `if`, `then`, `else`, `write`, `read`

In addition we will have some common operators, such as `<`, `>`, `:=` and so on, as well as numbers and strings (which we however ignore for the moment). We also need to specify what the “whitespace” is in our programming language and what comments should look like. As a first try, we might specify the regular expressions for our language roughly as follows

$$
\begin{array}{lcl}
\mathit{LETTER} & := & \texttt{a} + \texttt{A} + \texttt{b} + \texttt{B} + \ldots \\
\mathit{DIGIT} & := & \texttt{0} + \texttt{1} + \texttt{2} + \ldots \\
\mathit{KEYWORD} & := & \texttt{while} + \texttt{if} + \texttt{then} + \texttt{else} + \ldots \\
\mathit{IDENT} & := & \mathit{LETTER} \cdot (\mathit{LETTER} + \mathit{DIGIT} + \_)^* \\
\mathit{OP} & := & \texttt{:=} + \texttt{<} + \ldots \\
\mathit{NUM} & := & \mathit{DIGIT}^+ \\
\mathit{WHITESPACE} & := & (\text{"~~"} + \backslash\texttt{n})^+
\end{array}
$$

Having the regular expressions in place, the problem we have to solve is: given a string of our programming language, which regular expression matches which part of the string. By solving this problem, we can split up a string of our language into components. For example given the input string `i f ␣ t r u e ␣ t h e n ␣ x + 2 ␣ e l s e ␣ x + 3`

we expect it will be split up as follows

`if` `␣` `true` `␣` `then` `␣` `x` `+` `2` `␣` `else` `␣` `x` `+` `3`

This process of splitting an input string into components is often called *lexing* or *scanning*. It is usually the first phase of a compiler. Note that the separation into words cannot, in general, be done by just looking at whitespaces: while `if` and `true` are separated by a whitespace, this is not always the case. As can be seen the three components in `x+2` are not separated by any whitespace. Another reason for recognising whitespaces explicitly is that in some languages, for example Python, whitespace matters. However in our small language we will eventually just filter out all whitespaces and also all comments.

Lexing will not just separate a string into its components, but also classify the components, that is explicitly record that `if` is a keyword, ␣ a whitespace, `true` an identifier and so on. For the moment, though, we will only focus on the simpler problem of just splitting a string into components.

There are a few subtleties we need to consider first. For example, say the string is

`i f f o o ␣ …`

then there are two possibilities for how it can be split up: either we regard the input as the keyword `if` followed by the identifier `foo` (both regular expressions match) or we regard `iffoo` as a single identifier. The choice that is often made in lexers is to look for the longest possible match. This leaves `iffoo` as the only match (since it is longer than `if`).

Unfortunately, the convention about the longest match does not yet make the whole process of lexing completely deterministic. Consider the string

`t h e n ␣ …`

Clearly, this string should be identified as a keyword. The problem is that also the regular expression *IDENT* for identifiers matches this string. To overcome this ambiguity we need to rank our regular expressions. In our running example we just use the ranking

$$\mathit{KEYWORD} < \mathit{IDENT} < \mathit{OP} < \ldots$$

So even if both regular expressions match in the example above, we give preference to the regular expression for keywords.

Let us see how our algorithm for lexing works in detail. The regular expressions and their ranking are shown above. For our algorithm it will be helpful to have a look at the function *zeroable* defined as follows:

$$
\begin{array}{lcl}
zeroable(\varnothing) & \stackrel{\text{def}}{=} & true \\
zeroable(\epsilon) & \stackrel{\text{def}}{=} & false \\
zeroable(c) & \stackrel{\text{def}}{=} & false \\
zeroable(r_1 + r_2) & \stackrel{\text{def}}{=} & zeroable(r_1) \wedge zeroable(r_2) \\
zeroable(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & zeroable(r_1) \vee zeroable(r_2) \\
zeroable(r^*) & \stackrel{\text{def}}{=} & false
\end{array}
$$

In contrast to the function $nullable(r)$, which test whether a regular expression can match the empty string, the *zeroable* function identifies whether a regular expression cannot match anything at all. The mathematical way of stating this property is

$$zeroable(r) \text{ if and only if } L(r) = \varnothing$$

Let us fix a set of regular expressions rs. The crucial idea of the algorithm working with rs and the string, say

$$\boxed{c_1}\boxed{c_2}\boxed{c_3}\boxed{c_4}\boxed{\ldots}$$

is to build the derivative of all regular expressions in rs with respect to the first character c_1. Then we take the results and continue with building the derivatives with respect to c_2 until we have either exhausted our input string or all of the regular expressions are “zeroable”.