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:

```
/* Fibonacci Program
   input: n */

write "Fib";
read n;    // n := 19;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

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

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

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

`if␣true␣then␣x+2␣else␣x+3`

we expect it is split up as follows

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

This process of splitting up 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 in the example above, 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, whitespaces matters, that is carry meaning. However in our small language we will eventually just filter out all whitespaces and also all comments.

Lexing not just separates a string into its components, but also classifies the components, meaning it explicitly records 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 up a string into components.

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

`iffoo␣` ...

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 in this case (since it is longer than `if`).

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

`then␣` ...

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

$$KEYWORD < IDENT < OP < \ldots$$

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. In addition to the functions *nullable* and *der*, it will use the function *zeroable* defined as follows:

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

Recall that the function *nullable*(r) tests whether a regular expression can match the empty string. The function *zeroable*, on the other hand, tests whether a regular expression cannot match anything at all. The mathematical way of stating this property is

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

For what follows let us fix a set of regular expressions rs as being

$$KEYWORD,\ IDENT,\ WHITESPACE$$

specifying the "words" of our programming language. The algorithm takes as input the rs and a string, say

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

and tries to chop off one word from the beginning of the string. If none of the regular expression in rs matches, we will just return the empty string.

The crucial idea in the algorithm is to build the derivatives 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". Suppose the input string is

$$\boxed{\texttt{i}}\boxed{\texttt{f}}\boxed{\texttt{2}}\boxed{\texttt{␣}} \ldots$$

then building the derivatives with respect to `i` gives

	zeroable
der `i` (*KEYWORD*)	no
der `i` (*IDENT*)	no
der `i` (*WHITESPACE*)	yes

We can eliminate *WHITESPACE* as a potential candidate, because no derivative can go from *zeroable* = yes to no. That leaves the other two regular expressions as potential candidate and we have to consider the next character, `f`, from the input string

	zeroable
der `f` (*der* `i` (*KEYWORD*))	no
der `f` (*der* `i` (*IDENT*))	no

Since both are ‘no’, we have to continue with `2` from the input string

	zeroable
$\mathit{der}\ \texttt{2}\ (\mathit{der}\ \texttt{f}\ (\mathit{der}\ \texttt{i}\ (\mathit{KEYWORD})))$	yes
$\mathit{der}\ \texttt{2}\ (\mathit{der}\ \texttt{f}\ (\mathit{der}\ \texttt{i}\ (\mathit{IDENT})))$	no

Although we now know that the beginning is definitely an *IDENT*, we do not yet know how much of the input string should be considered as an *IDENT*. So we still have to continue and consider the next derivative.

	zeroable
$\mathit{der}\ \text{␣}\ (\mathit{der}\ \texttt{2}\ (\mathit{der}\ \texttt{f}\ (\mathit{der}\ \texttt{i}\ (\mathit{IDENT}))))$	yes

Since the answer is now ‘yes’ also in this case, we can stop: once all derivatives are zeroable, we know the regular expressions cannot match any more letters from the input. In this case we only have to go back to the derivative that is nullable. In this case it is

$$\mathit{der}\ \texttt{2}\ (\mathit{der}\ \texttt{f}\ (\mathit{der}\ \texttt{i}\ (\mathit{IDENT})))$$

which means we recognised an identifier. In case where there is a choice of more than one regular expressions that are nullable, then we choose the one with the highest precedence. You can try out such a case with the input string

$$\boxed{\texttt{t}}\boxed{\texttt{h}}\boxed{\texttt{e}}\boxed{\texttt{n}}\boxed{\text{␣}}\ \ldots$$

which can both be recognised as a keyword, but also an identifier.

While in the example above the last nullable derivative is the one directly before the derivative turns zeroable, this is not always the case. Imagine, identifiers can be letters, as permuted by the regular expression *IDENT*, but must end with an undercore.

$$\mathit{NEWIDENT} \quad := \quad \mathit{LETTER} \cdot (\mathit{LETTER} + \mathit{DIGIT} + \_)^* \cdot \_$$

If we use *NEWIDENT* with the input string

$$\boxed{\texttt{i}}\boxed{\texttt{f}}\boxed{\texttt{f}}\boxed{\texttt{o}}\boxed{\texttt{o}}\boxed{\text{␣}}\ \ldots$$

then it will only become *zeroable* after the ␣ has been analysed. In this case we have to go back to the first `f` because only

$$\mathit{der}\ \texttt{f}\ (\mathit{der}\ \texttt{i}\ (\mathit{KEYWORD}))$$

is nullable. As a result we recognise successfully the keyword `if` and the remaining string needs to be consumed by other regular expressions or lead to a lexing error.