

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 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-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:

```
1  write "Input a number ";
2  read n;
3  x := 0;
4  y := 1;
5  while n > 0 do {
6    temp := y;
7    y := x + y;
8    x := temp;
9    n := n - 1
10 };
11 write "Result ";
12 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 “white space” is in our programming language and what comments should look like. As a first try, we specify the regular expressions for our language roughly as follows

<i>KEYWORD</i>	$:=$	while + if + then + else + ...
<i>IDENT</i>	$:=$	<i>LETTER</i> · (<i>LETTER</i> + <i>DIGIT</i> + $_$)*
<i>OP</i>	$:=$:= + < + ...
<i>NUM</i>	$:=$	<i>DIGIT</i> ⁺
<i>WHITESPACE</i>	$:=$	" " + \n

with the usual meanings for the regular expressions *LETTER* and *DIGIT*.

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. In this way we can split up a string into components. For example we expect that the input string

```
i f _ t r u e _ t h e n _ x + 2 _ e l s e _ x + 3
```

is 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 looking at whitespaces: while **if** and **true** are separated by a whitespace, the components in **x+2** are not. 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 comments.

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

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

```
iffoo_...
```

then there are two possibilities 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 of the longest match does not yet make the whole process completely deterministic. Consider the input string

```
then_...
```

Clearly, this string should clearly be identified as a keyword. The problem is that also the regular expression *IDENT* for identifiers would also match this string. To solve this ambiguity we need to rank our regular expressions.