Problem Sheet 1

Lexical and Syntax Analysis

1.1 In the lecture, we wrote a regular expression to describe decimal and hexadecimal constants in C, and derived an NV and a DFA from it. But there was a white lie, because C forbids decimal constants with a leading zero, and allows unsigned octal constants that start with a zero and continue with an arbitrary string of octal digits 0 to 7, a convention beloved of those ancients who programmed the PDP–11. Thus the string 0293 is not an integer constant, because the non-octal digit 9 is inconsistent with the leading zero. Modify the regular expression to reflect this rule, and show what changes result in the NFA and DFA.

Answer: Consider the fact that the number being octal or not is a disjoint subset. Therefore, we will consider the regex form to be a union over octal numbers and non-octal numbers.

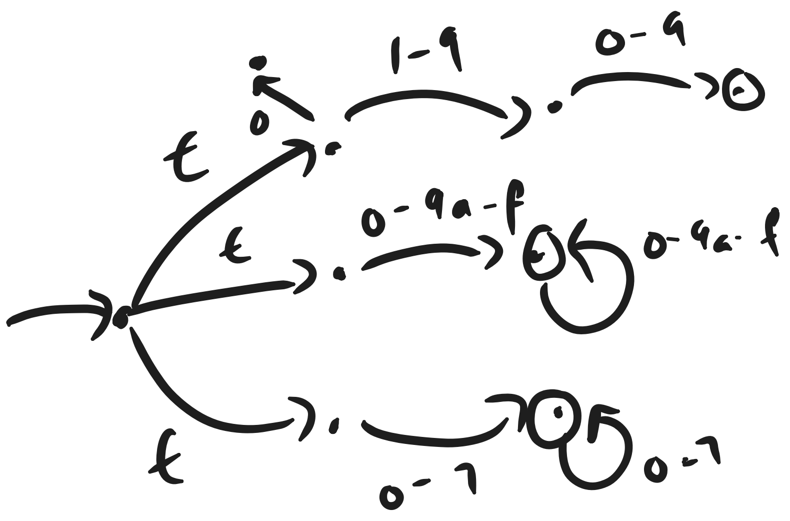
Here's the regex for an octal string: `0o[0-7]+`

And here is the rest, accommodating that change where decimals can't begin with a zero: `-([1-9]+)[0-9]+|0x[0-9a-f]+`

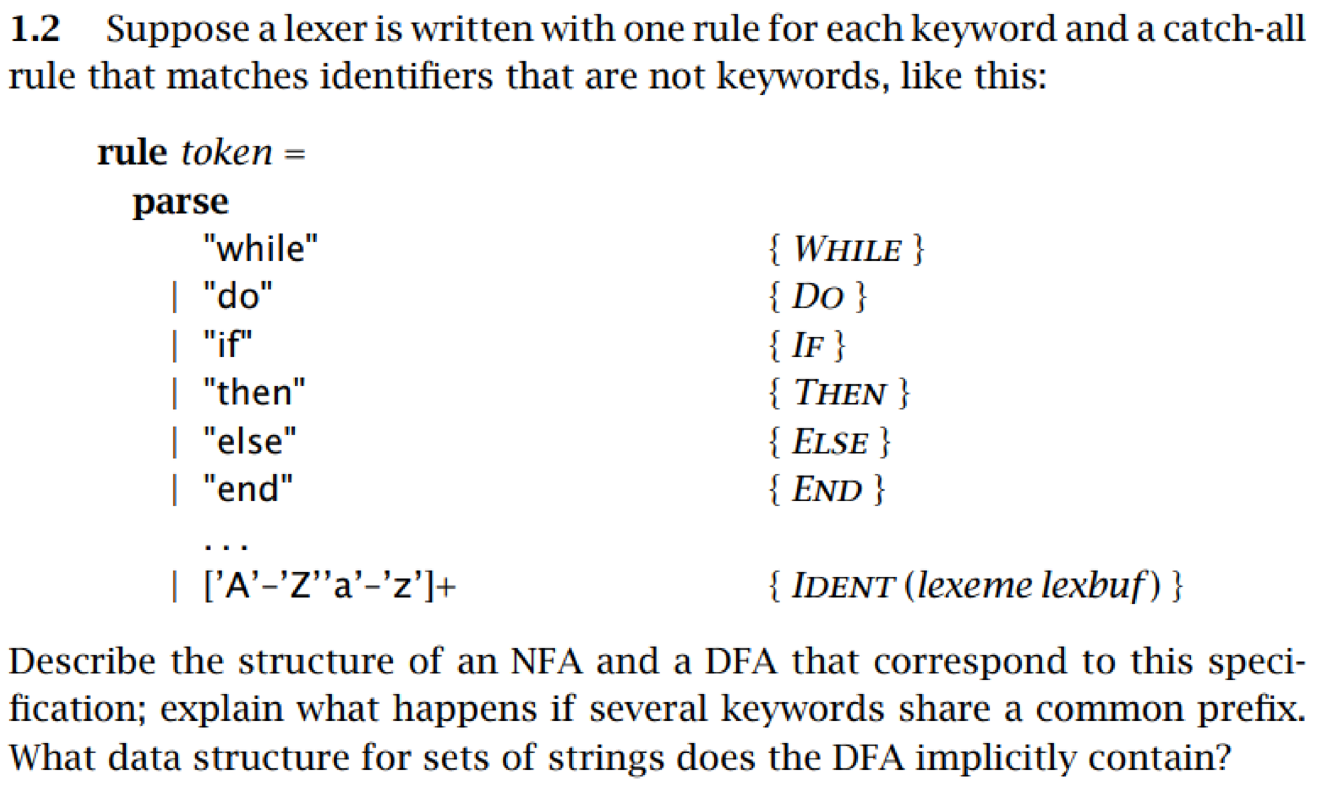
And now, here is the full form:

```
-([1-9]+)[0-9]+|0x[0-9a-f]+|0o[0-7]+
```

Here is the updated NFA.



1.2:



1.2 Suppose a lexer is written with one rule for each keyword and a catch-all rule that matches identifiers that are not keywords, like this:

```
rule token =
  parse
      "while"                    { WHILE }
    | "do"                       { DO }
    | "if"                       { IF }
    | "then"                     { THEN }
    | "else"                     { ELSE }
    | "end"                      { END }
      . . .
    | ['A'-'Z''a'-'z']+          { IDENT (lexeme lexbuf) }
```

Describe the structure of an NFA and a DFA that correspond to this specification; explain what happens if several keywords share a common prefix. What data structure for sets of strings does the DFA implicitly contain?

Answer: Constructing an NFA from this would involve setting ϵ-transitions from the initial state to n different states where n is the number of keywords. From there, each state will have its NFA that corresponds to a specific token, let's say `WHILE` for now.

The NFA will accept this string and this string only:

$$q' \overset{W}{\Longrightarrow} q_1 \overset{H}{\Longrightarrow} q_2 \overset{I}{\Rightarrow} q_3 \overset{L}{\Rightarrow} q_4 \overset{E}{\Rightarrow} q_5$$

and each and every other keyword will have their own path exactly like this one. If two keywords are to have the same prefix, then it is possible that either this specific model for the NFA breaks. This is how you would fix it. Suppose `ELSE` and `END` are our words here, and both begin with an E. So, we will have a concrete transition $q' \overset{E}{\Rightarrow} q^*$, where q^* has a further two extra ϵ-transitions that consider the strings `ND` and `LSE`.

1.3: Lex has the conventions that the longest match wins, and that earlier rules have higher priority than later ones in the script. These conventions are exploited in the lexer shown in Exercise 1.2 that recognises both keywords and identifiers. Would it be possible to describe the set of identifiers that are not keywords by a regular expression, without relying on these rules? If so, would this be a practical way of building a lexical analyser?

Answer: Given that both keywords and identifiers use the same 26 alphabets, it is tricky, but it could be possible. However, without the rules above, specifically that the earlier rules have higher priority could mean that an identifier called `WHILE_THIS` could be mistakenly interpreted as a keyword instead of an identifier.

Even if we enforce this constraint and try to go about not relying on the rules defined above, the NFA required to make this regex interpreter becomes larger regardless, meaning that the actual DFA implementation grows exponentially along with it. Although possible, it is not a practical way of building a lexical analyser.

1.4: In C, a comment begins with '/*' and extends up to the next occurrence of '*/'. Write a regular expression that matches any comment. What would be the practical advantages and disadvantages of using this regular expression in a lexical analyser for C?

Answer: Here is the regex for a comment in C: `/*[^]+?*/

Note this isn't what was shown in lectures, because C treats `*` as a special character in regex. Therefore, the need for the backslash in order to match the literal `\*`.

Also, C uses `^` as the same character as `.`, but with `\n` also accepted, meaning this comment can span multiple lines.

1.5: In Pascal, comments can be nested, so that a comment beginning with (* and ending with *) can have other comments inside it. What is an advantage of this convention? Show how Pascal comments can be handled in a lexical analyser written according to the conventions of `ocamllex` by using either recursion or an explicit counter.

Answer: An advantage of this convention is that you can explain the purpose of a given purpose inside a comment block where its nature is previously specified, basically making it easier to document code in the script itself. What I mean by this is that for example, let's say you have a factorial function and the comment above looks like:

```
(*
   This function calculates the factorial of a given integer n, n > 0.
   (* the factorial function relates recursively with the equation
      F(n) = n * F(n-1)

*)
```

In `ocamellex`, we can define a rule for nested comments with the following token Type constructor:

```
rule handle_comment depth = parse
| "(*" { handle_comment (depth + 1) lexbuf } (* Nested comment *)
| "*)" { if depth = 1 then tokenize lexbuf else handle_comment (depth - 1)
lexbuf }
| _ { handle_comment depth lexbuf } (* Skip over all other characters *)
```

We have used a buffer and a kind of "stack" system that manages how many open comments we have made, and how many of those we close. It is actually somewhat analogous to Catalan numbers/sequences, where an open bracket represents a +1, and a close bracket represents a -1. The sequence therefore needs to end at zero and always remain positive.

1.6:The following productions for if statements appeared in the original definition of Algol 60:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textit{basic-stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt}. \end{array}$$

Show that these productions lead to an ambiguity in the grammar. Suggest an unambiguous grammar that corresponds to the interpretation that associates each else with the closest possible if. Now consider the ambiguous grammar: because it is ambiguous, a shift– reduce parser must have a state where both shifting and reducing lead to a successful conclusion, or a state where it is possible to reduce by two different productions. Find a string with two parse trees according to the grammar, and show the parser state where two actions are possible. Describe the results of shifting and of reducing in this state.do you know the definition of the if statement in the original definition of algol 60?

Answer: The problem with defining the structure of the if statement in this manner is that if there are many different if statements, its ambiguous to know which if the else clause should be attached to. As a matter of fact, it is referred to as the "dangling else" problem.

A solution to this problem is introducing this new idea of a "closed" or an "open"-statement. We can define it to be as such:

Let S be a statement.

Then S is *closed* if "S **else** S_1" is not a statement for any S_1.

And, S is *open* if "S **else** S_1" is a statement for some S_1.

This allows us to define which statement is going to have the else clause appended to it, and which one is left opened. Here is the specification for that:

```
stmt -> basic-stmt | opn-stmt | cls-stmt
      |  if expr then stmt
          | (opn-stmt then stmt) | (if opn-stmt then stmt)+
              | cls-stmt else cls-stmt
              | if opn-stmt then stmt if cls-stmt then stmt else stmt
```

It would be nice if we could go over this, since I didn't quite understand how this would be transcribed in this specific format: https://dl.acm.org/doi/pdf/10.1145/365813.365821#:~:text=The%20dangling%20else%20problem%20consists,and%20%242%20are%20basic%20statements.

Imagine we have the following if statement:

```
if expr_1 then
        pass
if expr_2 then
        pass
else:
        pass
```

Suppose we reduce the first if statement, then we get the following:

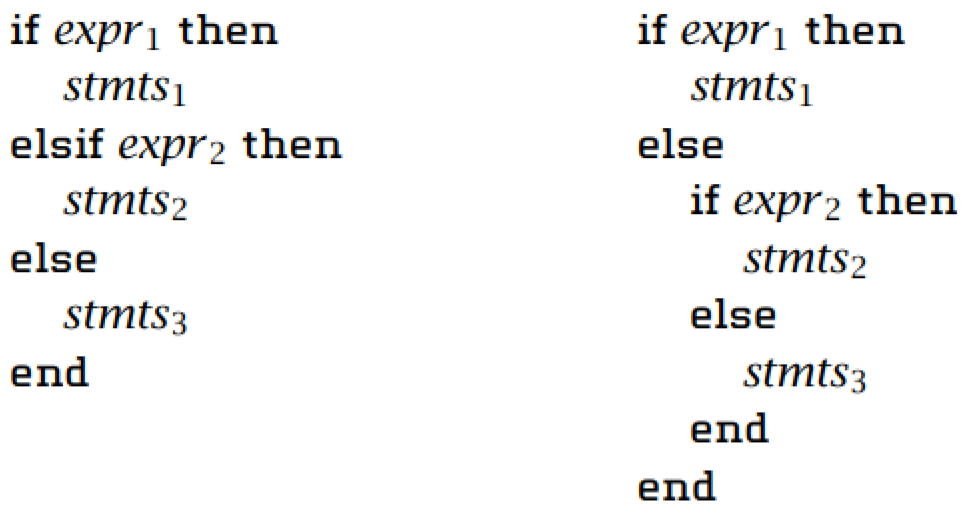
```
if expr_2 then
        pass
else:
        pass
```

or, if we reduce the second if statement, we get the following:

```
if expr_1 then
        pass
else:
        pass
```

You can imagine an NFA with two ϵ-transitions that both end with an accepting state, but two completely different logical expressions.

1.7: In the language of Lab 1, `if` statements have an explicit terminator end that removes the ambiguity discussed in the preceding exercise. However, this makes it cumbersome to write a chain of if tests, since the end keyword must be repeated once for each `if`. Show how to change the parser from Lab 1 to allow the syntax shown on the left in Figure 1 as an abbreviation for the language of Lab 1, `if` statements have an explicit terminator end that removes the ambiguity discussed in the preceding exercise. However, this makes it cumbersome to write a chain of `if` tests, since the end keyword must be repeated once for each `if`. Show how to change the parser from Lab 1 to allow the syntax shown on the left in Figure 1 as an abbreviation for the syntax on the right. An arbitrarily long chain of tests written with the keyword `elsif` can have a single end; the else part remains optional. Arrange for the parser to build the same abstract syntax tree for the abbreviated program as it would for its equivalent written without `elsif`. Here is the figure which was referenced:



```
if expr1 then                  if expr1 then
    stmts1                         stmts1
elsif expr2 then               else
    stmts2                         if expr2 then
else                                   stmts2
    stmts3                         else
end                                    stmts3
                                   end
                               end
```

Goal: The goal is to find a suitable expression that not only removes the need for an "end" keyword, but also deals with the "hanging else" problem.

```
stmt :
    IF expr THEN stmt %prec IF          { IfStmt($2, $4, Skip) }
  | IF expr THEN stmt ELSE stmt         { IfStmt($2, $4, $6) }
  | other_statements ;
```

The `%prec IF` is a precedent rule that ensures the `else` associated with the nearest preceding `if`. What we have done here is that the closest unmatched if statement is the one that receives the else clause.

Although not a complete solution, it is one where some level of freedom is taken away from the programmer in order to make the program more consistent and unambiguous.

1.8: One grammar for lists of identifiers contains the productions,

```
idlist ->    id
        |    idlist "," id
```

(we call this left recursive), and another (right recursive) contains the productions,

```
idlist ->    id
        |    id "," idlist
```

In parsing a list of 100 identifiers, how much space on the parser stack is needed by shift–reduce parsers based on these two grammars? Which grammar is more convenient if we want to build an abstract syntax tree that is a list built with cons?

Proposition: Using right recursive shifts is far better than using left recursive lists.

The reason behind this is that when using left recursive shifts, each and every one of the 100 identifiers need to be on the stack, (assuming left is evaluated first). This would mean that more storage would be required in the beginning of parsing in every identifier, unlike right recursive shifts. You can deal with identifiers one at a time, saving space.

It's somewhat akin to Haskell's `foldl` and `foldr` functions, where `foldr` is typically used over `foldl` due to that exact reason.