Individual Exercises - Lecture 6

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 3 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/3/studio.shtml
   A virtual machine is available on Moodle with the necessary tools pre installed.

2. Do Fischer et al exercise 1, 2, 9, 22 on pages 106-112 (exercise 3, 4, 14, 23 on pages 134-141 in GE)

   1. Assume the following text is presented to a C scanner:

```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d  Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
    }
}
```

   What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

   ID(main), LPAREN, RPAREN, LBRACE

   CONST, FLOATDCL, ID(payment), ASSIGN, FLOATNUM(384.00), SEMICOLON

   FLOATDCL, ID(bal), SEMICOLON

   INTDCL, ID(month) ASSIGN INTNUM(0) SEMICOLON

   Etc…

   2. How many lexical errors (if any) appear in the following C program? How should each error be handled by the scanner?

```
main(){
        if(1<2.)a=1.0else a=1.0e-n;
        subr('aa',"aaaaa (newline in string not allowed without \ )
                aaaaa");
        /* That's all  (*/ missing)
}
```

   The blue letters will cause errors. The first one needs a decimal after the decimal point. The scanner could then give an error message saying that this is the case. The second one needs space between 0 and e, otherwise the scanner cannot create a token from it. This is a bit harder since C actually supports scientific numbers like

1.5321E10, so the error message might suggest either creating a scientific number. It cannot suggest that it would expect a space followed by else, since if statements in C do not require an else clause. The scanner will also have problems with the last comment, since it is not closed. The error message for this could be quite difficult, since an unclosed multiline comment simply in effect erases the rest of the file. Now the parser must see it while creating the function, since it is no longer closed in curly brackets.

9. Most compilers can produce a source listing of the program being compiled. This listing is usually just a copy of the source file, perhaps embellished with line numbers and page breaks. Assume you are to produce a prettyprinted listing.

(a) How would you modify a Lex scanner specification to produce a prettyprinted listing (that is, a listing with text properly indented, if-else pairs aligned, and so on)?
The lex scanner should keep track of and add information about each token's line and its number of indentation.

(b) How are compiler diagnostics and line numbering complicated when a prettyprinted listing is produced?
One token could represent text from multiple lines.

22. Write Lex regular expressions (using character classes if you wish) that match the following sets of strings:

(a) The set of all unprintable ASCII characters (those before the blank and the very last character)
[^ -~]+
Explanation: In the ASCII table the printable characters start with white space and ends with tilde. Using the ^ says, that we want anything that is not in this range, i.e. all the non-printable characters.
(b) The string ["""] (that is, a left bracket, three double quotes, and a right bracket)
\[\"\"\"\]
(c) The string x^12,345 (your solution should be far less than 12,345 characters in length)
X{12345}
This accepts exactly 12345 X's in a row.

3. *Write a regular expression definition for unsigned integer literals excluding those that contain unnecessary leading zeroes. Thus 0, 1, 123, 10000 are included, but 00, 0123 and 0010000 are excluded.*
([1-9][0-9]*)|0
Either we have 0 or we have one non-zero number followed by zero or more numbers from 0 to 9.

4. *You have scanned an integer literal into a character buffer (e.g. using yytext). You now want to convert the string representation to a numeric (int) form. However, the string may represent a value too large for the int form. Explain how to convert a string representation of an integer literal into numeric form with full overflow checking.*
   An inefficient way could be to use existing libraries and convert the string to a 32-bit int and then back to a string which can be compared with the input string, if the two values are the same the number could be represented by a 32-bit int. To do it probably a method like the one used by FreeBSD in their version of LIBC should probably be used (like below approach):

   https://github.com/freebsd/freebsd/blob/master/lib/libc/stdlib/atoi.c

   https://github.com/freebsd/freebsd/blob/master/lib/libc/stdlib/strtol.c

   Another option is to also store the max value for a 32 bit integer in a char buffer and compare it to the integer literal one char at a time, starting from the left going right.

5. *(optional) Modify the studio associated with Crafting a Compiler Chapter 3 to produce a lexer for the ac language from Lecture 3 using the lexical grammar for ac in figure 2.3 on page 36 in (64 in GE edition) of Fischer et. al. Take a look at the Yylex source file in the autogen package and compare the code with the hand written ScannerCode class from Studio 2*
   You can use the virtual machine available on Moodle for this exercise.

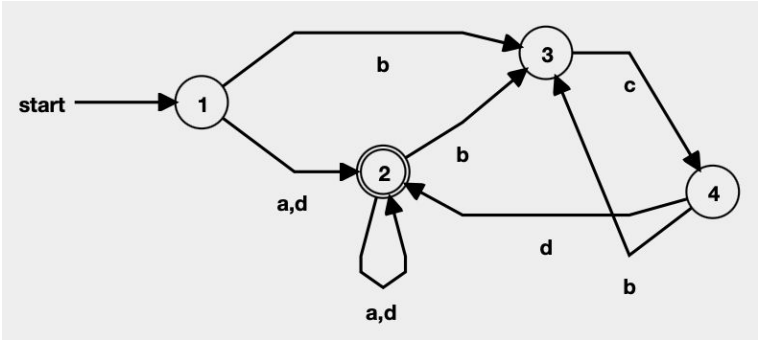
Group Exercises - Lecture 6

The following exercises are best done as group discussions:

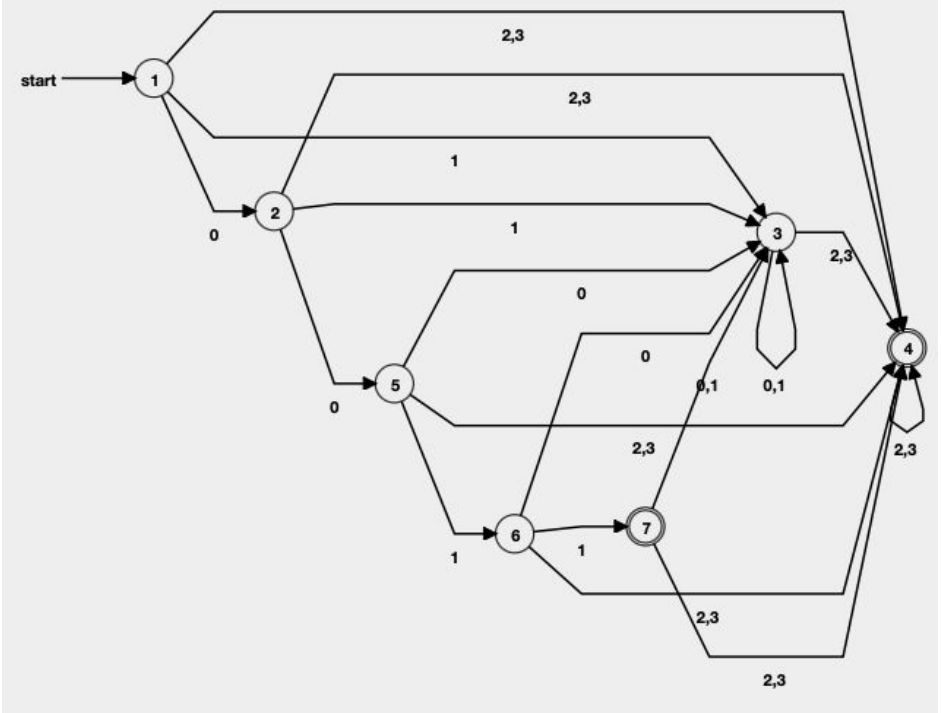
1. Discuss the outcome of the individual exercises
   Did you all agree on the exercise?

2. Do Fischer et al exercises 4, 6, 7, 8 on pages 106-112 (exercises 5, 6, 7, 9, 12, 13 on pages 134-141 in GE)

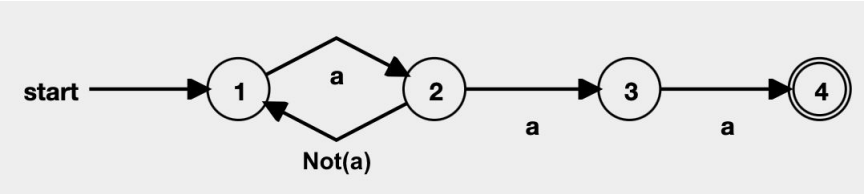
   4. Write DFAs that recognize the tokens defined by the following regular expressions:
   Note that in the following DFAs there are implicit transitions (ie. not shown on the figure) from each state to an error state, whenever an illegal input is read.
   (a)
   (a | (bc)*d)+



3. (b)
   ((0 | 1)*(2 | 3)+) | 0011



(c)
(a Not(a))*aaa



6. Write a regular expression that defines a C-like comment delimited by /* and */. Individual *'s and /'s may appear in the comment body, but the pair */ may not.
(\/*)(.|\n)*?(*\/)
We start with a /* with both characters having to be escaped with backslash. This may be followed by any number of characters and lastly followed by a */ (again escaped). The ? makes it stop the first time, it finds a */ in the end.

7. Define a token class AlmostReserved to be those identifiers that are not reserved words but that would be if a single character were changed. Why is it useful to know that an identifier is "almost" a reserved word? How would you generalize a scanner to recognize AlmostReserved tokens as well as ordinary reserved words and identifiers?
AlmostReserved tokens could be used to identify possible syntax errors caused by misspelling reserved keywords. This can be used to provide a warning for possible (off-by-one) syntax errors, and make it easy for the compiler to provide a "did you mean x?" error message to the user.

The grammar used to drive the parser must be rewritten to treat the token classes AlmostReserved and Identifier as interchangeable until an error is recognized. If the parsing error occurred in a state that would accept a reserved word and the input token was AlmostReserved, then the value of the token would be checked to see if it was a variant of the expected reserved word. If so, the reserved word could be assumed to be the intended input token and parsing could be restarted from the state it was in before the error.

AlmostReserved tokens would have to be recognized by using an extension of one of the reserved word lookup techniques described at the end of Section 3.7.1 on page 79. Because of the large number of identifiers that are one character change removed from reserved words, it would cause a huge increase in the number of states in the scanner tables to recognize them by generating regular expressions to recognize each of the possibilities.

8. When a compiler is first designed and implemented, it is wise to concentrate on correctness and simplicity of design. After the compiler is fully implemented and tested, you may need to increase compilation speed. How would you determine whether the scanner component of a compiler is a significant performance bottleneck? If it is, what might you do to improve performance (without affecting compiler correctness)?
To test the performance one could test the entire compiler with a profiler that checks how much time is spent in each function. This would reveal if some scanner functions were slow. Alternatively one could collect start time and end time for each compiler phase to reveal if one phase (for example the parsing phase) is much slower than the others.

To improve the scanning speed one could use a scanner generator like Flex or GLA, which are designed to generate very fast scanners.

If your scanner is hand-coded, avoid reading single characters, which can be very expensive. Instead bulk load into buffers, perhaps even double buffers. Performance can also be increased by copying as few characters as possible from the input buffers to the tokens. For example only copy the values, if they are needed for later analysis, for example literal values.

4. (optional) Read the Lab associated with Crafting a Compiler Chapter 3 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/3/lab.shtml and take a look at the "official" solution in the file FischerLab356solutions.zip in the General Course material directory https://www.moodle.aau.dk/pluginfile.php/154451/mod_folder/content/0/FischerLab356solutions.zip?forcedownload=1 (You may of course Follow the Lab without looking at the solution, but be warned that this is a tough one)
For this a virtual machine will be provided with the necessary tools pre installed.