

lab session 2: compiler construction

Automata and LL(1) parsing

November 28, 2017

The second lab session of the course *Compiler Construction* consists of 7 exercises. The first five exercises concern finite automata (which are internally used by Flex). The last two exercises concern parsing arithmetic expressions using a hand-written Recursive Descent Parser.

Finite State Automata

On Nestor you can find the tar-file `cclab2.tar` which contains the files `automaton.c`, `Makefile`, `intset.c`, `intset.h`, and a few example files describing NFAs (filenames `example1.nfa`, `example2.nfa` and `example3.nfa`).

The format of the `*.nfa` files is as follows. The first line contains the number of states n . The states are numbered 0, 1, ..., $n - 1$. The next line contains the number of the starting state. The 3rd line contains a set of final (accepting) states. The remaining lines describe the transitions. These line start with a state number s , followed by a character x (or the word `eps` for ϵ). Note that a character is placed between a single quote, or is denoted as `#` followed by its ASCII table index. The 3rd component on the line is the set of states that can be reached from s on accepting the input x . As an example, the content of the file `example1.nfa` is:

```
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
```

The files `intset.*` implement an Abstract Data Type (ADT) for sets of unsigned integers (see for the API the file `intset.h`). In this lab exercise, these sets are used for representing sets of states in an automaton, where the states are simply integer numbers. Have a look at the file `intset.h` which operations are available. You are not allowed to change the implementation of the data type `intSet`.

The file `automaton.c` contains definitions of the datatypes `nfa` and `dfa`, and some basic operations on them. Study this code before you make the following exercises.

Exercise 1: Epsilon closure of a state in an NFA

Implement the body of the function `intSet epsilonClosureState(nfa automaton, State s)`. The function should return the set of states reachable from `s` without accepting any input. Make a program that reads from the input an NFA (using the format described above). The specification of the NFA is terminated by a line containing the `#` character. Next, a line is given containing a state number s . Print on the output the ϵ -closure of s .

Example 1:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
0
output:
{0,1,2}

```

Example 2:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
2
output:
{1,2}

```

Example 3:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
3
output:
{3}

```

Exercise 2: Epsilon closure of a state set in an NFA

Implement the function `intSet epsilonClosureStateSet(nfa automaton, intSet s)`. This function should return the set of states reachable from all states from the set `s` without accepting any input.

Next, make a program that reads from the input an NFA (using the format described above). The specification of the NFA is terminated by a line containing the `#` character. Next, a line is given containing a set of states `s`. Print on the output the ε -closure of the set `s`.

Example 1:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
{0}
output:
{0,1,2}

```

Example 2:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
{0,2}
output:
{0,1,2}

```

Example 3:

```

input:
4
0
{2,3}
0 eps {2}
0 '0' {1}
1 '1' {1,3}
2 eps {1}
2 '0' {3}
3 #48 {2}
#
{0,3}
output:
{0,1,2,3}

```

Exercise 3: NFA to DFA conversion

Implement the body of the function `dfa nfa2dfa(nfa n)` which returns the DFA that is obtained using the powerset construction algorithm that converts an NFA into a DFA. This conversion process has been discussed in the lectures, and can be found in the lecture slides. Moreover, you are referred to the following wikipedia page https://en.wikipedia.org/wiki/Powerset_construction or youtube movie <https://www.youtube.com/watch?v=taClnxU-nao>.

Next, write a program that reads from the input an NFA, and outputs the corresponding DFA that is obtained by applying the powerset construction algorithm. Note that the output format of a DFA is very similar to the format of an NFA. The only difference is that transitions go from state to state (and not from state to set of states), so the curly braces are not printed.

Example:

input:

2
0
{1}
0 'a' {0}
0 'b' {0,1}

output:

3
0
{2}
0 'a' 1
0 'b' 2
1 'a' 1
1 'b' 2
2 'a' 1
2 'b' 2

Exercise 4: Minimization using Brzozowski's algorithm

Implement the body of the function `dfa nfa2minimalDFA(nfa n)` which returns the DFA that is obtained using Brzozowski's algorithm that converts an NFA into a minimal DFA.

Next, write a program that reads from the input an NFA, and outputs the corresponding minimal DFA (using the same format as in the previous exercise).

Example:

input:

2
0
{1}
0 'a' {0}
0 'b' {0,1}

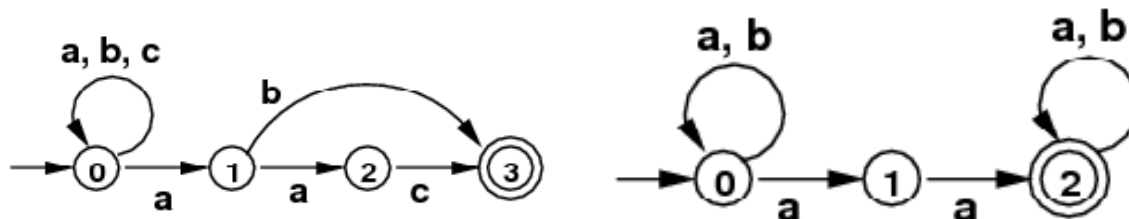
output:

2
0
{1}
0 'a' 0
0 'b' 1
1 'a' 0
1 'b' 1

Exercise 5: Merging NFA's and transformation into a minimal DFA

Write a program that accepts on its standard input a series of NFAs. The input consists of an integer n , followed by n specifications of NFAs. The specification of each automaton is terminated by the character '#'. These NFAs must be merged into a large 'super' NFA. This 'super' automaton is converted into a minimal 'super' DFA (using the code of exercises 1-3). Once, this super DFA has been constructed, the program should read strings from the input, and answer for each string whether it is accepted by the automaton or not. Note that the automaton must accept the longest possible string.

As an example, consider the following two NFAs:



The specification of these two automata can be found in the files `example2.nfa` and `example3.nfa`. The left automaton accepts strings over the alphabet $\{a, b, c\}$ that end in "ab" or "aac", i.e. the regular expression $(a|b|c)^*(ab|aac)$. The right automaton accepts strings over the alphabet $\{a, b\}$ that contain the substring "aa", i.e. the regular expression $(a|b)^*aa(a|b)^*$. Of course, the merged 'super' automaton accepts the regular expression $((a|b|c)^*(ab|aac)) \mid ((a|b)^*aa(a|b)^*)$. The alphabet of this super automaton is the union of the original alphabets, i.e. $\{a, b, c\}$. If we feed this 'super' automaton with the input string "aabababbabaaabbbab",

then it should accept the entire string, since it ends in "ab", Note that the prefix "aab" also ends in "ab", but the automaton matches the longest possible string. Also note that this string is actually accepted by both automata.

Now consider the input string "aabcaacaacbabbabaaabbbabb". The fused automaton should accept the longest prefix "aabcaacaacbabbabaaabbbab", and leave the remaining input "b".

Example input:

```

2
4
0
{3}
0 'a' {0,1}
0 'b' {0}
0 'c' {0}
1 'a' {2}
1 'b' {3}
2 'c' {3}
#
3
0
{2}
0 'a' {0,1}
0 'b' {0}
1 'a' {2}
2 'a' {2}
2 'b' {2}
#
aabababbabaaabbbab
aabcaacaacbabbabaaabbbabb

```

Example output:

```

Accepted "aabababbabaaabbbab"
Remaining ""
Accepted "aabcaacaacbabbabaaabbbab"
Remaining "b"

```

Parsing arithmetic expressions

Consider the following BNF context free grammar for arithmetic expressions.

```
<E> ::= number
<E> ::= ( <E> )
<E> ::= <E> + <E>
<E> ::= <E> - <E>
<E> ::= <E> * <E>
<E> ::= <E> / <E>
<E> ::= <E> ^ <E>
```

Operator precedence (high to low): \wedge , $*$, $/$, $+$, $-$

Associativity for binary operators:

```
 $\wedge$  is right associative
+, -, *, / are left associative
```

The grammar is clearly ambiguous. Moreover, it does not take into account the precedence of the operators and their associativities. Note that the exponentiation operator (\wedge) is right associative (so 2^3^4 means $2^{(3^4)}$), while all other operators are left associative.

Exercise 6: Making a recursive descent parser by hand

Convert the grammar into an equivalent LL(1) grammar that takes into account the right operator precedence and the right associativities. Use flex to make a suitable scanner for this grammar. Next, write by hand a recursive descent LL(1) parser for this grammar. Expressions that are accepted by the parser must be echoed to the screen in RPN (Reverse Polish Notation). For expressions containing a syntax error, the program should print "SYNTAX ERROR".

Example 1:

```
input:
3-4*5
output:
3 4 5 * -
```

Example 2:

```
input:
(3-4)*5
output:
3 4 - 5 *
```

Example 3:

```
input:
3 + 4 5
output:
SYNTAX ERROR
```

Exercise 7: Making a recursive descent parser using LLnextgen

Repeat the previous exercise using LLnextgen to generate a parser.