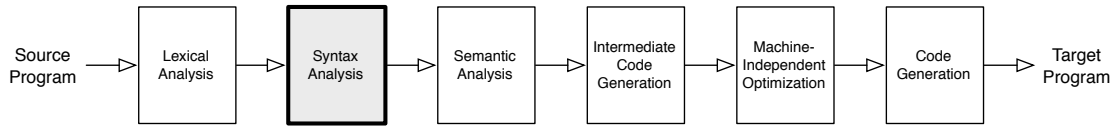
CS 301 Spring 2019
Tutorial Assignment
12 February

Lexers, Grammars, Top-Down Parsing

1 Plan



This week we wrap up lexical analysis and begin syntax analysis:

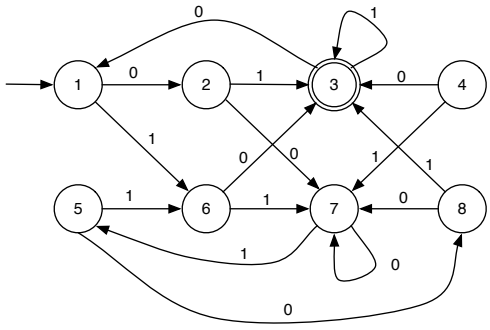
- We apply our foundations for regular expressions and finite automata to see how a lexical analyzer can be constructed automatically from a specification of token patterns (exercises 1 and 2). The first project stage applies these ideas in developing a full lexical specification as input to a tool for automatic lexer generation.
- The reading on syntax analysis covers grammars and a first general parsing technique. The problems explore properties of grammars that make them suitable for describing programming languages and automatic parsing. Next week, we continue with more parsing.

2 Readings

- Dragon 3.8, 3.9.6, 3.10
  Alternative: EC 2.4.4-2.4.5, 2.5.1
- Refer to last week's readings (including Russ Cox's article) for discussion of the conversion of regular expression to NFA to DFA (Thompson's construction and subset construction).
- Dragon 4.1 – 4.4 (focus on 4.1.1-4.1.2, 4.2-4.2.5, 4.2.7, 4.3-4.3.4, 4.4-4.4.4)
  Alternative: EC 3.1 – 3.3
- **Optional:** If you are curious about the general algorithm behind the ad hoc method we used to transform our C-comment DFA into a regular expression, see Kleene's construction in EC 2.6.1.

3 Exercises

1. Minimize the states of the following DFA. Label each state in the minimized DFA with the set of states from the original DFA to which it corresponds.



2. Consider the following lexical analysis specification:

```
(aba)+   { return Tok1; }
(a(b)+a) { return Tok2; }
(a|b)    { return Tok3; }
```

In case of tokens with the same length, the token whose pattern occurs first in the above list is returned.

(a) Build an NFA that accepts strings matching one of the above three patterns using Thompson's construction.

(b) Transform your NFA into a DFA using the subset construction. Label the DFA states with the set of NFA states to which they correspond. Indicate the final/accept states in the DFA and label each of these states with the (unique) token being returned in that state.

(c) Show the steps in the functioning of the lexer for the input string `abaabbaba`. Indicate what tokens the lexer returns for successive calls to `getToken()`. For each of these calls indicate the DFA states being traversed in the automaton.

3. Dragon 4.2.1

4. Dragon 4.2.3 (a) — (e)

5. Dragon 4.3.1

6. Consider the following grammar:

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}\ S \mid \mathbf{b}\ S\ \mathbf{a}\ S \mid \epsilon$$

(a) Show that the grammar is ambiguous by constructing two different rightmost derivations for some string.

(b) Construct the corresponding parse trees for this string.

(c) **Optional:** Write an unambiguous grammar that describes the same language. (There is no algorithm to remove ambiguiuty from a grammar. I suggest first trying to understand the language for the original grammar and then constructing a new unambiguous CFG from scratch that accepts the same language.)

7. Consider the following grammar:

$$\begin{aligned} S &\rightarrow B\ C\ \mathbf{z} \\ B &\rightarrow \mathbf{x}\ B \mid D \\ C &\rightarrow \mathbf{u}\ \mathbf{v} \mid \mathbf{u} \\ D &\rightarrow \mathbf{y}\ D \mid \epsilon \end{aligned}$$

(a) Is this grammar LL(1)? Explain why (not). If not, modify the grammar to be LL(1) before proceeding.

(b) Compute the FIRST and FOLLOW sets for the (possibly modified) grammar.

(c) Construct the LL(1) parsing table.
NOTE: There is a typo in the Dragon book in the description of how to construct the parsing table. On page 224, step 1 of Algorithm 4.31 should refer to FIRST(α), and *not* FIRST(A). This has been fixed in some printings (international paperback?) but not all.

(d) Show the steps taken to parse **xxyuz** with your table. (Use Fig. 4.21 as an example of how to show the parser's progress.)

8. Consider the following grammar for statements:

$$
\begin{array}{rcl}
\textit{Stmt} & \rightarrow & \texttt{if E then } \textit{Stmt}\ \textit{StmtTail} \\
& | & \texttt{while E } \textit{Stmt} \\
& | & \{\ \textit{List}\ \} \\
& | & \texttt{S} \\
\\
\textit{StmtTail} & \rightarrow & \texttt{else } \textit{Stmt} \\
& | & \epsilon \\
\\
\textit{List} & \rightarrow & \textit{Stmt}\ \textit{ListTail} \\
\\
\textit{ListTail} & \rightarrow & ;\ \textit{List} \\
& | & \epsilon
\end{array}
$$

   Unlike Java (and like ML), semicolons separate consecutive statements. You can assume `E` and `S` are terminals that represent other expression and statement forms that we do not currently care about. If we resolve the typical conflict regarding expansion of the optional `else` part of an `if` statement by preferring to consume an `else` from the input whenever we see one, we can build a predictive parser for this grammar.

   (a) Build the LL(1) predictive parser table for this grammar.

   (b) Using Figure 4.21 in the Dragon book as a model, show the steps taken by your parser on input:

   ```
   if E then S else while E { S }
   ```

9. **Optional:** Use the techniques outlined in Dragon 4.4.5 to add error-correcting rules to your table.

10. **Optional:** Describe the behavior of your parser on the following two inputs:

    - `if E then S ; if E then S }`
    - `while E { S ; if E S ; }`