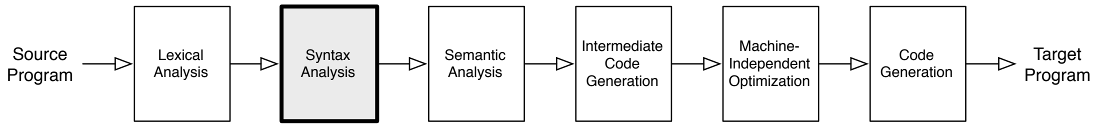
HW 2: Lexical Analyzers, Grammars, and Top-Down Parsing

CSCI 434T
Spring, 2019

Overview



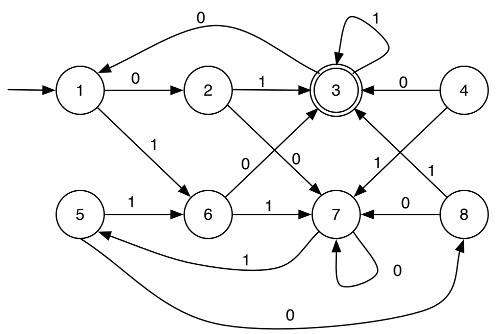
- We'll finish our exploration of Lexical Analysis this week by 1) looking more deeply at the automatic construction of a lexical analyzer from a specification of token patterns (problems 1 and 2), and 2) implementing the lexical analyzer for IC. The second part is described in a separate handout on PA 1.
- We'll also begin to look at the next phase of the compiler: Syntax Analysis. The reading on this topic covers grammars and the first of two general parsing techniques. The problems explore properties of grammars that make them suitable for describing programming languages and automatic parsing.

Readings

- Dragon 3.8 and 3.9.6
- Dragon 4.1 – 4.4

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. Label the DFA states with the set of NFA states to which they correspond. Indicate the final 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 does the lexer return 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:

$$S \rightarrow a\,S\,b\,S \mid b\,S\,a\,S \mid \epsilon$$

(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.

7. Consider the following grammar:

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

(a) Is this grammar LL(1)? If it is not, explain why, and then 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 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).**

(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:

$$\begin{aligned} \mathit{Stmt} &\rightarrow \texttt{if E then}\ \mathit{Stmt}\ \mathit{StmtTail} \\ &\mid \texttt{while E}\ \mathit{Stmt} \\ &\mid \{\ \mathit{List}\ \} \\ &\mid \texttt{S} \\ \\ \mathit{StmtTail} &\rightarrow \texttt{else}\ \mathit{Stmt} \\ &\mid \epsilon \\ \\ \mathit{List} &\rightarrow \mathit{Stmt}\ \mathit{ListTail} \\ \\ \mathit{ListTail} &\rightarrow\ ;\ \mathit{List} \\ &\mid \epsilon \end{aligned}$$

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 }
```

(c) Use the techniques outlined in Dragon 4.4.5 to add error-correcting rules to your table.

(d) 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 ; }`