CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- ~~**Syntax Analysis (Ch 4)**~~
  - ~~**Introduction (Ch 4.1)**~~
    - ~~**The Role of the Parser (4.1.1)**~~
    - ~~**Representative Grammars (4.1.2)**~~
    - ~~**Syntax Error Handling (4.1.3)**~~
    - ~~**Error-Recovery Strategies (4.1.4)**~~
  - **Context-Free Grammars (4.2)**
    - ~~**The Formal Definition of a Context-Free Grammar (4.2.1)**~~
    - ~~**Notational Conventions (4.2.2)**~~
    - ~~**Derivations (4.2.3)**~~
    - ~~**Parse Trees and Derivations (4.2.4)**~~
    - ~~**Ambiguity (4.2.5)**~~
    - **Verifying the Language Generated by a Grammar (4.2.6)**

- Context-Free Grammars Versus Regular Expressions (4.2.7)
- Writing a Grammar (4.3)
  - Lexical Versus Syntactic Analysis (4.3.1)
  - Eliminating Ambiguity (4.3.2)

Verifying the Language Generated by a Grammar (4.2.6)

- A proof that a grammar G generates a language L has two parts.
- We need to show every string generated by G is in L,
- Conversely, every string in L can indeed be generated by G.
- Two directions. It contains "basis" part and "induction" part
- We are not going through this part because it is not interesting to all of you ^_^

Context-Free Grammars (CFG) Versus Regular Expressions (4.2.7)

- CFG is actually a more powerful tool than RegExp
- Every construct that can be described by a RegExp can be described by a CFG, but is not vice-versa
- Consider the following RegExp: (a|b)*abb, and
- The grammar:

$$
\begin{array}{lcl}
A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \rightarrow & bA_2 \\
A_2 & \rightarrow & bA_3 \\
A_3 & \rightarrow & \epsilon
\end{array}
$$

- They can describe the same language:
  - A0→aA1→abA2→abbA3→abb epsilon → abb

Context-Free Grammars (CFG) Versus Regular Expressions (4.2.7)

- We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA).
- There is a high level algorithm to construct a grammar, to recognize the same language as NFA.
  - The purpose of a grammar is to recognize a language
  - This is what we are going to construct, but how to do that?

$$
\begin{array}{lcl}
A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \rightarrow & bA_2 \\
A_2 & \rightarrow & bA_3 \\
A_3 & \rightarrow & \epsilon
\end{array}
$$

- You know (remember) the NFA, it has states (start, middle, accepting), and transitions,

Context-Free Grammars (CFG) Versus Regular Expressions (4.2.7)

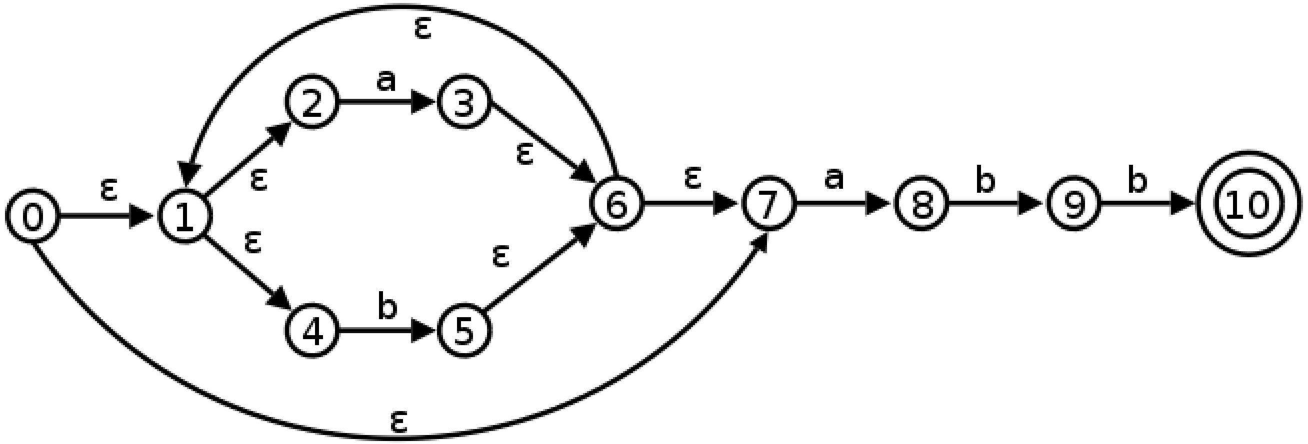
- **The Algorithm (very high level description, a general guideline)**

1. For each state i of the NFA, create a nonterminal A_i.

2. If state i has a transition to state j on input a, add the production $A_i \rightarrow aA_j$. If state i goes to state j on input ϵ, add the production $A_i \rightarrow A_j$.

3. If i is an accepting state, add $A_i \rightarrow \epsilon$.

4. If i is the start state, make A_i be the start symbol of the grammar.



$$
\begin{array}{lcl}
A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \rightarrow & bA_2 \\
A_2 & \rightarrow & bA_3 \\
A_3 & \rightarrow & \epsilon
\end{array}
$$

Lexical Versus Syntactic Analysis (4.3.1)

- Why have a separate lexer and parser?
- Since the lexer deals with REs / Regular Languages, and the parser deals with the more powerful Context Free Grammars (CFGs) / Context Free Languages (CFLs), everything a lexer can do, a parser could do as well. (parser is more powerful)
- The reasons for separating the lexer and parser are from **software engineering considerations.**

Lexical Versus Syntactic Analysis (4.3.1)

- REs are easier than CFGs to understand.
- More efficient algorithms/tools exist for automating the RE-based lexer than for the CFG-based parser. (We got more helps from the previous stages)
- Only the lexer need to deal with the external environment.
  - Because it takes in the inputs

Eliminating Ambiguity (4.3.2)

- We are going to introduce a very famous programming language bug, in the early development history of the programming language --- the "Dangling else"
- The 1st one looks ok but the second one?
  - if a then s
  - if b then s1 else s2 ← human readable?
- In C language, we have the following which is more understandable
  - If a then s1, else if b then s2 (high level, not the real C language)
  - Or just use "{" and "}" directly!

Eliminating Ambiguity (4.3.2)

- Another example (Very ambiguous, isn’t it?)
  - “If a then if b then s else s2”
    - What is that? For me, it could be several interpretations!
    - if a then (if b then s) else s2 … (1)
    - if a then {(if b then s) else s2} … (2)
  - If a and b are true, then s will be executed
  - But we might interpret that **s2 will get executed** when a is false for (1)
  - Or, when a is true (because a is true, the “1st then” part will get executed, from { to }) and b is false. So, s2 will be executed
- Some language has such kind of “bug” and is not using by people anymore --- i.e. ALGOL 60

Eliminating Ambiguity (4.3.2)

- Is there any ways we can do to avoid ambiguities?
  - Yes! We do! A most commonly used way is to use the "**end if**"
    - i.e. ALGOL 68, Ada, VB, AppleScript (MacOS 8, 9)
  - Or just a '{' and '}'
- In the book, they just give us an example. We say it is an ambiguous grammar because it has 2 parsing trees
  - For example, in this grammar, it can go with the 1st production first or the 2nd production first

$$
\begin{array}{rcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
& | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
& | & \textbf{other}
\end{array}
\tag{4.14}
$$

Eliminating Ambiguity (4.3.2)

- Everything looks perfectly normal in this example (keep running the 2nd rule)

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\ & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\ & | & \textbf{other} \end{array} \qquad (4.14)$$

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

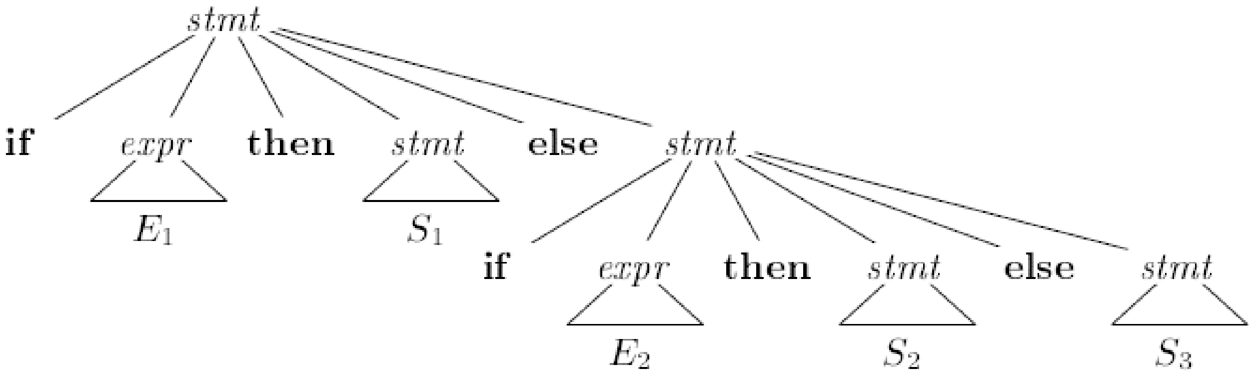


Figure 4.8: Parse tree for a conditional statement

Eliminating Ambiguity (4.3.2)

- What about this example?

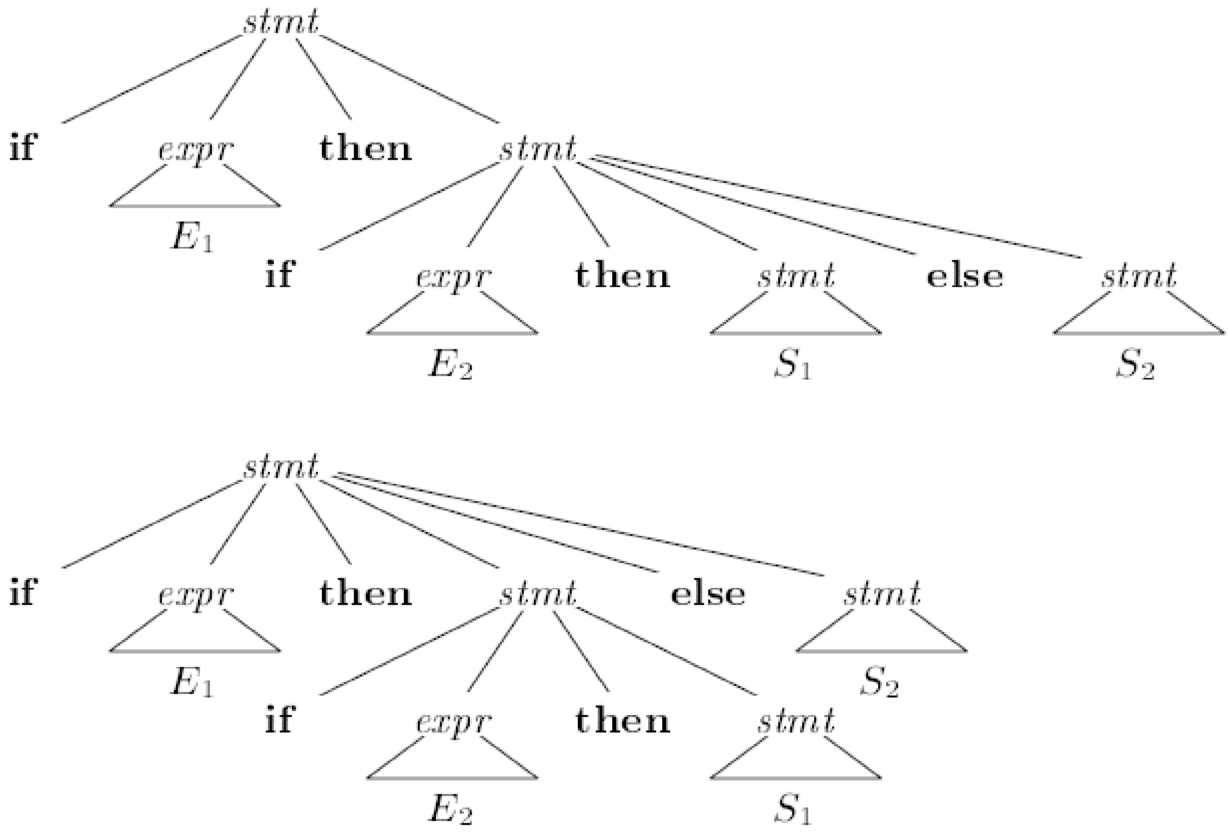
$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \qquad (4.15)$$

- Unfortunately, it has 2 parsing trees
- See the next page

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array} \qquad (4.14)$$

Eliminating Ambiguity (4.3.2)

- Because we need to make a decision in the 1st layer.
- Keep growing to the RHS or not? (decision)



$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
\qquad (4.14)
$$

Figure 4.9: Two parse trees for an ambiguous sentence