Compiler Design
Syntax Analysis
Writing a Grammar

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Lexical Versus Syntactic Analysis
- Eliminating Ambiguity
- Elimination of Left Recursion
- Left Factoring
- Non-Context-Free Language Constructs

Grammars

- describe most of the programming language syntax
- some aspects can not be described by a context-free grammar
  - identifiers must be declared before they are used
- sequence of tokens accepted by the parser forms a superset of the programming language
- Subsequent phases of the compiler will analyze the parser output to ensure compliance with supplementary rules

Next…

- How to divide the work between lexical analyzer and parser
- Transformations to make a grammar suitable for top-down parsing
  - Left recursion elimination
  - Left factoring
- Programming language constructs which cannot be described by any grammar

Lexical vs. Syntactic Analysis

- Everything that can be described by a regular expression can be described by a grammar

Why to use regular expressions to define lexical syntax of a language ?

- Separating the syntactic structure into lexical and non-lexical is a convenient way of modularizing the front end of a compiler into two components
- Lexical rules
  - are quite simple
  - do not need a powerful notation such as grammars
- Regular expressions provide a concise and easier to understand notation for tokens than grammars
- Efficient lexical analyzers can be constructed automatically from regular expressions than from grammars

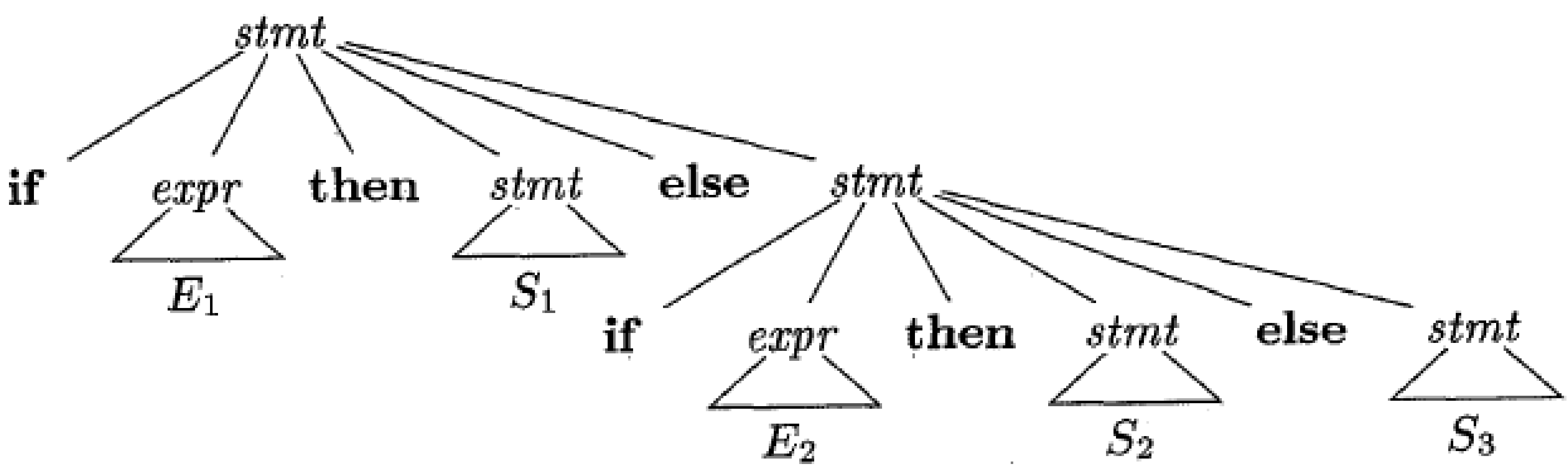
Eliminating Ambiguity

- sometimes ambiguous grammar can be rewritten to eliminate ambiguity
- stmt-> **if** expr **then** stmt

  | **if** expr **then** stmt **else** stmt

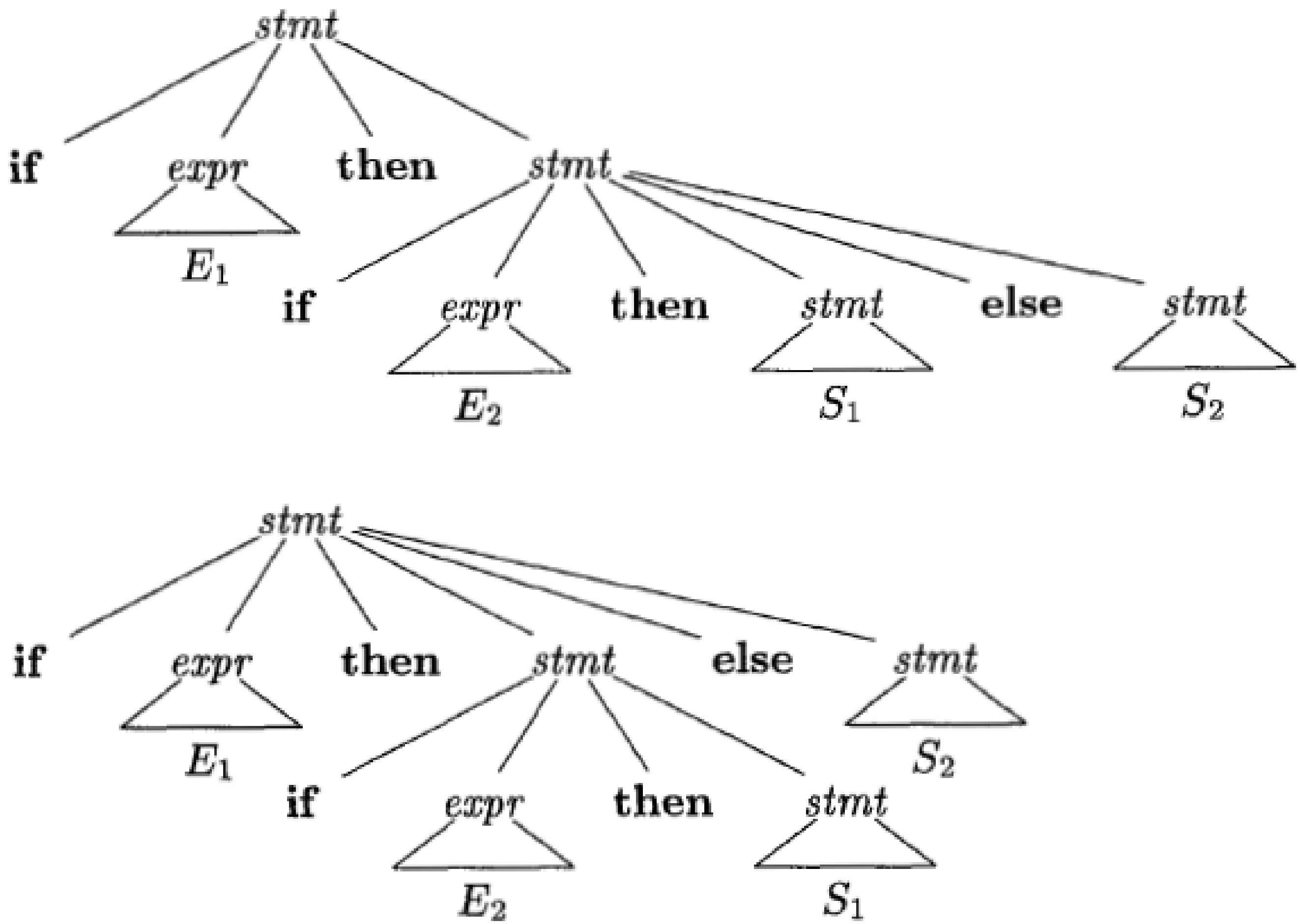
  | **other**
- **if** E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

Parse Tree for a Conditional Statement



Ambiguous Grammar Example

- **if** E_1 **then if** E_2 **then** S_1 **else** S_2



Ambiguous Grammar Example

- General rule
  - match “else” with closest unmatched “then”
  - it is the case also for C language which misses the “then” keyword but it is implied by “{“, “}”
- disambiguation should be present in the grammar
- in practice it is rarely present in the production rules

Disambiguation Solution for the Dangling Else Example

stmt ->

matched_stmt | open_stmt

matched_stmt ->

if expr **then** matched_stmt **else** matched_stmt

| other

open_stmt ->

if expr **then** stmt

| **if** expr **then** matched_stmt **else** open_stmt

Elimination of Left Recursion

- general case
  - a grammar is recursive if there is a derivation $A \stackrel{+}{=}> A\alpha$ for some string α
- particular case
  - immediate left recursion $A \to A\alpha$
  - solution
    - $A \to A\alpha | \beta$
    - $A \to \beta A'$
    - $A' \to \alpha A' | \varepsilon$

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- E->TE’
- E’->+TE’|ε
- T->FT’
- T’->*FT’|ε
- F->(E) | id

Direct Left Recursion

- $A \text{->} A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\ \beta_1|\ \beta_2|\ldots|\beta_n$
- no βi begins with A
- $A \text{->} \beta_1 A'|\ \beta_2 A'|\ldots|\beta_n A'$
- $A' \text{->}\ \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_m A'|\varepsilon$

Indirect Left Recursion Example

- S-> A a | b
- A-> A c | S d | ε

- S=>Aa=>Sda
  - not immediate left recursive

Eliminating Left Recursion

- Input
  - grammar G with no cycles or ε-productions
- Output
  - an equivalent grammar with no left recursion
- Method
  - …

Method

1. arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$
2. for (each i from 1 to n){
3. for(each j from 1 to i-1){
4. replace each production of the form Ai->Ajγ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all A_j-productions
5. }
6. eliminate the immediate left recursion among A_i-productions
7. }

Method

- iteration i=1
  - eliminates any immediate left recursion among A_1-productions
  - any remaining A_1 productions of the form $A_1 \rightarrow A_t\alpha$ must have t>1
- iteration i-1
  - all A_k where k<i are "cleaned"
  - any production $A_k \rightarrow A_t\alpha$ must have t>k

Example - revisited

- S-> A a | b
- A-> A c | S d | ε
- we order S, A
- i=1
  - no left recursion is in S
- i=2
  - we replace in A the S by the rule S->A a | b
  - A->A c | A a d | b d | ε

Example - revisited

- S->A a | b
- A-> b d A’ | A’
- A’ -> c A’ | a d A’ | ε

Left Factoring

- grammar transformation useful for producing a grammar suitable for predictive, top-down parsing
- e.g.
  - stmt -> **if** expr **then** stmt **else** stmt

    | **if** expr **then** stmt
- A->$\alpha\beta_1$ | $\alpha\beta_2$
- A->αA'
- A'->β_1 | β_2

Left Factoring a Grammar

- Input
  - grammar G
- Output
  - equivalent left-factored grammar
- Method
  - for each non-terminal A find the longest prefix α to two or more alternatives
  - replace A-productions A-> $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

Left Factoring a Grammar

- A->αA’ | γ
- A’-> β_1 | β_2 |…| β_n

Dangling-else Problem

- S -> i E t S | i E t S e S | a
- E -> b

- S -> i E t S S’ | a
- S’ -> e S | ε
- E -> b

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007