ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 5b

Syntax Analysis

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Regular Expressions vs Context-Free Grammars

- Grammar for the regular expression *(a|b)*abb*

  $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

  $A_1 \rightarrow bA_2$

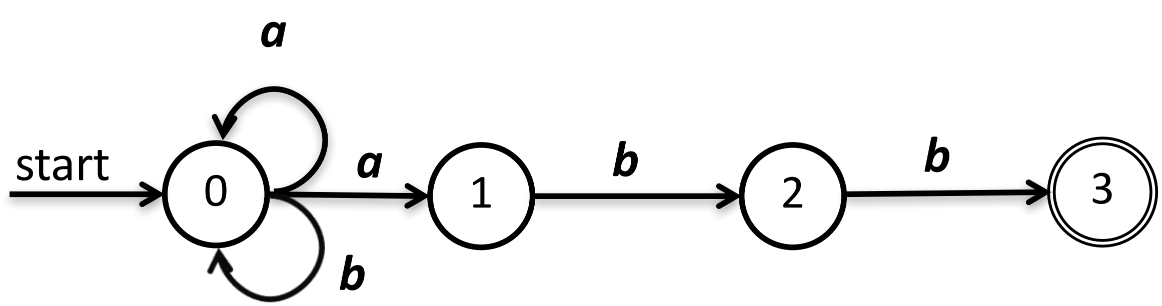
  $A_2 \rightarrow bA_3$

  $A_3 \rightarrow \varepsilon$

Construct a grammar from an NFA

- For each state i of the NFA, create a nonterminal symbol A_i
  - If state *i* has a transition to state *j* on symbol *a*, introduce the production: $A_i \rightarrow aA_j$
  - If state *i* has a transition to state *j* on symbol *ε*, introduce the production: $A_i \rightarrow A_j$
  - If state *i* is an accepting state: $A_i \rightarrow \varepsilon$
  - If *i* is the start state, then make A_i be the start symbol of the grammar

Recall the NFA version:



REs are useful

1. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
2. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.
3. More efficient lexical analyzers can be construct automatically from regular expressions than from arbitrary grammars.
4. Separating the syntactic structure of the language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two mangeable-sized components.

Eliminating Ambiguity

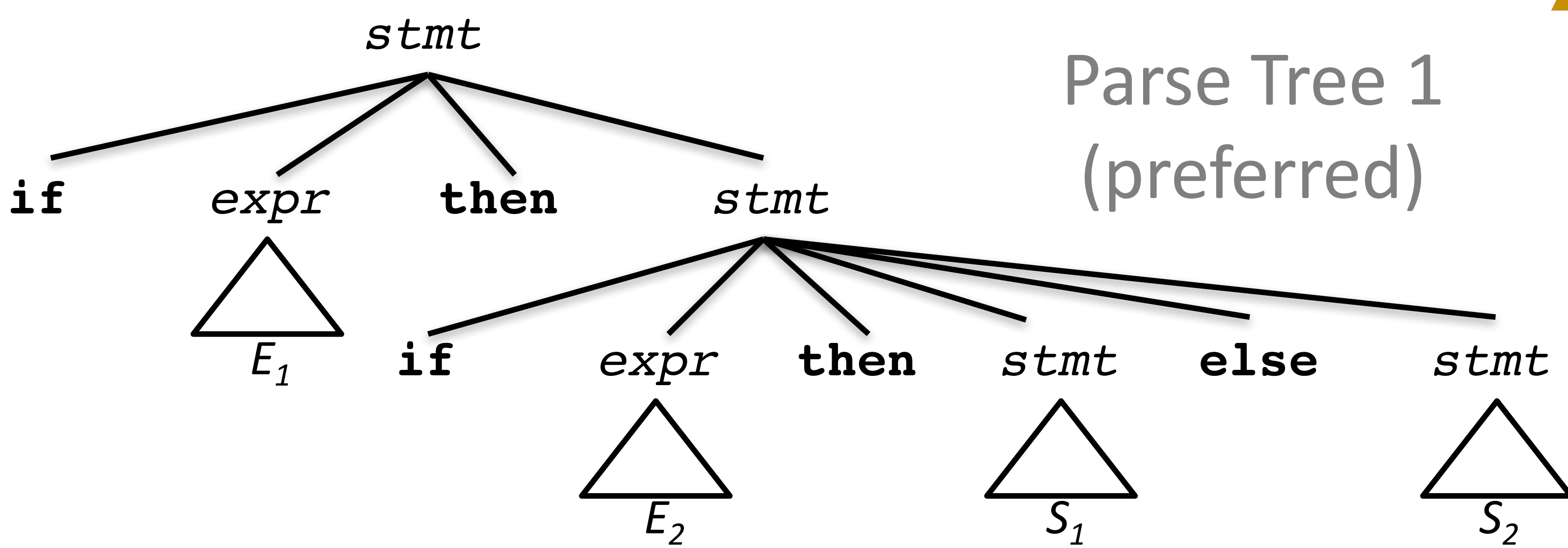
stmt → **if** *expr* **then** *stmt* |
if *expr* **then** *stmt* **else** *stmt* |
other

Valid Sentence

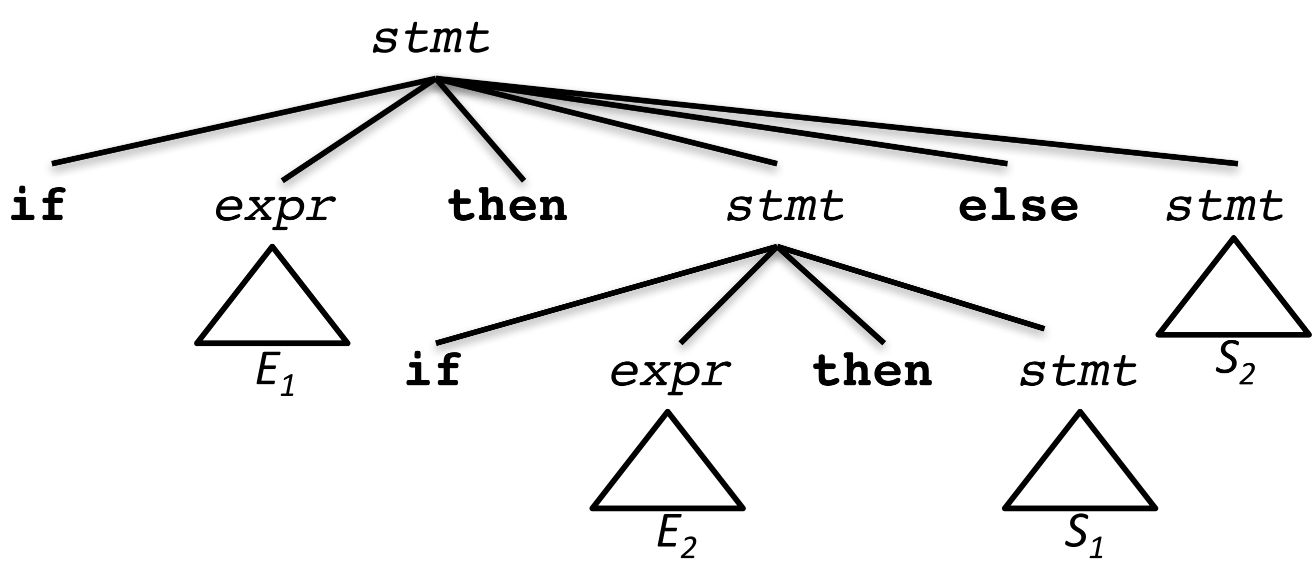
if E_1 **then** **if** E_2 ***then*** S_1 ***else*** S_2

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

Parse Tree 1 (preferred)



Parse Tree 2



Eliminating Ambiguity

- General rule
  - *Match each* **`else`** *with the closest previous unmatched* **`then`**.

Unambiguous Version

stmt → *matched_stmt* | *unmatched_stmt*
matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt*
| **other**
unmatched_stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

The idea is that a statement appearing between a **`then/else`** must be matched, i.e., it must not end with an unmatched **`then`** followed by any statement

Left Recursion

- Expressions where the leftmost symbol on the right side is the same as the nonterminal in the left side of the production are called *left recursive*
  - *expr* → *expr + term*
- These productions can cause the parser to loop forever

```
expr()
{
   expr(); match('+'); term();
}
```

Left Recursion Elimination

- A left-recursive production can be eliminated by re-writing. Consider:
  - *A* ➔ *Aa* | b, where a, b are sequences of terminals and nonterminals that do not start with A
- E.g., *expr* ➔ *expr + term* | term
  - A = expr, a = +term, b = term
- This production can be re-written as:
  - *A* ➔ *bR*
  - *R* ➔ *aR* | ε (R is right-recursive)

Left Recursion Elimination

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Generic Rule

- No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique.

(1) We group the A-productions as:

$A \rightarrow A\mathrm{a}_1 \mid A\mathrm{a}_2 \mid \ldots \mid A\mathrm{a}_m \mid \mathrm{b}_1 \mid \mathrm{b}_2 \mid \ldots \mid \mathrm{b}_m \mid$

(where no b_1 begins with an A)

(2) We replace the A-productions:

$A \rightarrow \mathrm{b}_1 A' \mid \mathrm{b}_2 A' \mid \ldots \mid \mathrm{b}_m A'$

$A' \rightarrow \mathrm{a}_1 A' \mid \mathrm{a}_2 A' \mid \ldots \mid \mathrm{a}_m A' \mid \varepsilon$

Non-immediate Left Recursion

$S \rightarrow Aa \mid \mathtt{b}$

$A \rightarrow Ac \mid S\mathtt{d} \mid \varepsilon$

The nonterminal *S* is left recursive because `S=>Aa=>Sda,` but it is not immediately recursive

Eliminating left recursion (any kind)

- *Input*
  - Grammar *G* with no cycles or ε-productions (cycle is $A \overset{+}{=>} A$, and ε-production is $A \rightarrow \varepsilon$)
- Output
  - An equivalent grammar with no left recursion

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
2. **for** i := 1 **to** n **do begin**

    for j := 1 **to** j-1 **do begin**

        replace each production of the form $A_i \rightarrow A_j\gamma$

        by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions

    end

    eliminate the immediate left recursion among the A_j-productions

end

Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

- We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during step (2) for the case $i = 1$.
- For $i = 2$, we substitute the *S*-productions in $A \rightarrow Sd$ to obtain the following *A*-productions: $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$
- The final grammar

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring

- When we have two productions

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

- on seeing the input token **if**, we cannot immediately tell which production to use to expand ***stmt***

Left Factoring

In general, if $A \rightarrow ab_1 | ab_2$ are two *A*-productions and the input begins with a nonempty string derived from a, we do not know whether to expand A to ab_1 or ab_2. However we may defer the decision by expanding A to aA'. Then after seeing the input derived from a, we expand A' to b_1 or to b_2:

$$A \rightarrow aA'$$

$$A' \rightarrow b_1 | b_2$$

Left Factoring a Grammar

- *Input*
  - Grammar *G*
- *Output*
  - An equivalent left-factored grammar
- *Method*
  - For each nonterminal A find the longest prefix a common to two or more of its alternatives. If a<>ε, i.e., there is a nontrivial common prefix, replace all the A productions $A \rightarrow ab_1 | ab_2 | \ldots | ab_n | \gamma$, where γ represents all alternatives that do not begin with a by

    $A \rightarrow aA' | \gamma$

    $A' \rightarrow b_1 | b_2 | \ldots | b_n$

Example

If expression then statement,
If expression then statement else statememt

$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

Non-Context-Free Grammars

Γραμματικές με Συμφραζόμενα

- $L_1 = \{wcw \mid \text{όπου } w \in (a|b)^*\}$
  - This language abstracts the problem of checking that identifiers are declared before their use in the program.
- $L_2 = \{a^n b^m c^n d^m \mid \text{όπου } n \geq 1, m \geq 1\}$
  - This language abstracts the problem of checking that the number of formal parameters in the declaration of a procedure agrees with the number of actual parameters in a use of the procedure
- Properties that cannot be expressed using a CFG are checked in Semantic Analysis