UNIVERSITY OF NICOSIA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΛΕΥΚΩΣΙΑΣ

COMP-421 Compiler Design

Presented by
Dr Ioanna Dionysiou

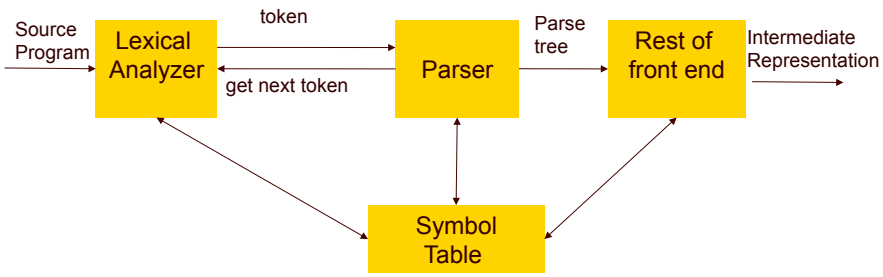
Administrative

- [ALSU07] Chapter 4 - Syntax Analysis
  - This week we will cover up to section 4.3
  - Next week we will cover top-down parsing (LL(1) parsers – section 4.4) and continue with bottom-up parsing (LR parsers – sections 4.5-4.6)

Lecture Outline

- Role of parser
- Context-free grammars
  - Review definition, derivations and parse trees
- Writing grammars
  - Ambiguity, eliminating left recursion, left factoring

Role of Parser

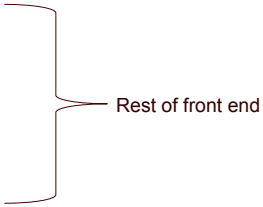


Obtains a string of tokens from the lexical analyzer and verifies that the string of tokens can be generated by the grammar for the source language

Output is some representation of the parse tree

Role of Parser (2)

- Assumption
  - Output of parser is some representation of a parse tree
- In practice
  - Other tasks are also performed
    - Collect more info about tokens
    - Type checking
    - Semantic analysis
    - Generating intermediate code



Types of Parsers

- 3 categories
  - Universal parsers
    - Can parse any grammar
    - Too inefficient to use in production compilers
  - Top-down parsers
    - Parse trees constructed from top (root) to bottom (leaves)
  - Bottom-up parsers
    - Parse trees constructed from bottom (leaves) up to the top (root)
- Most commonly used
  - Top-down and Bottom-Up
  - Input scanned from left-to-right, one symbol at a time

Syntax Error Handling

- Most programming language specifications do not describe error handling
  - It's up to the compiler designer
- Programs can contain errors at many different level:
  - Lexical
    - Misspelling an identifier
  - Syntactic (focus of error detection and recovery in a compiler)
    - Arithmetic expression with unbalanced parentheses
  - Semantic
    - Operator applied to an incompatible operand
  - Logical
    - Infinitely recursive call, use = instead of ==

Syntax Error Handling

- Goals of error handler in a parser
  - Report the presence of errors clearly and accurately
    - E.g. print the line that error found and indicate the nature of the error
  - Recover from each error quickly enough to be able to detect subsequent errors
    - Panic mode, phrase level, error productions, global correction
  - Must not slow down the processing of correct programs

Lecture Outline

- Role of parser
- Context-free grammars
  - Review definition, derivations and parse trees
- Writing grammars
  - Ambiguity, eliminating left recursion, left factoring
- Top-down Parsing
  - Recursive-descent parsing, non-recursive predictive parsing, construction of predictive parser

Context-Free Grammars

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

- Context-free grammars consists of terminals, nonterminals, a start symbol, and productions
  - A set of tokens, known as terminal symbols (if, then, else)
  - A set of nonterminals, which are variables that denote sets of strings (expr, stmt)
  - A set of productions where each production consists of
    - A nonterminal called the left side of the production
    - An arrow →
    - A sequence of terminals and/or nonterminals called the right side of the production
  - A designation of one of the nonterminals as the start symbol (stmt)

Context-Free Grammars

E→ E +E
E→ E * E
E→(E)
E→ -E
E→ **id**

Terminals?
Nonterminals?

Notational Conventions

1) These symbols are terminals:
   1) Lower-case letters early in the alphabet such as a, b, c
   2) Operator symbols such as *,+, etc
   3) Punctuation symbols such as (),!
   4) Digits 0,…,9
   5) Boldface strings such as **if, then**

Notational Conventions

2) These symbols are nonterminals:
   1) Upper-case letters early in the alphabet such as A, B, C
   2) The letter S, which when it appears, is usually the start symbol
   3) Lower-case italic names such as *expr, stmt*

Notational Conventions

3) Upper-case letters late in the alphabet, such as X, Y, Z, represent grammar symbols (either terminals or nonterminals)

4) Lower-case letters late in the alphabet, such as x, y, u, v, represent strings of terminals

5) Lower-case Greek letters represent strings of grammar symbols (this is a way to represent a generic production)

   $A \rightarrow \alpha$

6) If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …, $A \rightarrow \alpha_k$ then we may write $\mathbf{A \rightarrow \alpha_1 | \alpha_2 | \alpha_k}$

7) Unless otherwise stated, the left side of the first production is the start symbol

Notational Conventions

E→ E +E
E→ E * E
E→(E)
E→ -E
E→ **id**

E→ E +E | E * E |(E) |-E | **id**

Using conventional notations, we obtain the grammar on the right

Derivations

- There are several ways to view the process by which a grammar defines a language
  - Building a parse tree
    - graphical representation of derivation
  - Derivational view
    - precise description of the top-down construction of a parse tree
      - Idea: a production is a rewriting rule
      - Beginning with the start symbol, each rewriting step replaces the nonterminal by the right side of one of its productions

Derivations

E→ E +E | E * E |(E) |-E | **id** **Grammar G**

The expression E → -E allows us to replace any instance of E by -E

E ⇒ -E is read **'E derives -E'**

We can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements

E ⇒ -E ⇒ -(E) ⇒-(**id**)

The sequence of the above replacements is called a derivation of **–(id)** from E
This is a proof that one particular instance of an expression is the string –(**id**)

General Derivation

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_k$ **α_1 derives α_k**

Given Grammar G with start symbol S,

L(G) is the language generated by G

Strings in L(G) may only contain terminals of G

A string of terminals w is in L(G) iff $S \overset{+}{\Rightarrow} w$ (sentence)

If $S \overset{*}{\Rightarrow} \alpha$ where α may contain nonterminals, then we say that α is a sentential form of G. A sentence is a sentential form with no nonterminals.

Is –(**id+id**) a sentence of G?

E→ E +E | E * E |(E) |-E | **id** **Grammar G**

- At each step of derivation, there are 2 choices to be made:
  - Which nonterminal to replace, and
  - Having made this choice, which production to use for that nonterminal

Is –(**id+id**) a sentence of G?

E→ E +E | E * E |(E) |-E | **id** **Grammar G**

E⇒ -E ⇒ -(E)⇒ -(E+E) ⇒ -(**id**+E) ⇒ -(**id+id**)

E⇒ -E ⇒ -(E)⇒ -(E+E) ⇒ -(E**+id)** ⇒ -(**id+id**)

Order of replacement is different

Leftmost and rightmost derivations

- Leftmost derivations $\Rightarrow_{lm}$
  - Only the leftmost nonterminal in any sentential form is replaced at each step

E⇒ -E ⇒ -(E)⇒ -(E+E) ⇒ -(**id**+E) ⇒ -(**id**+**id**)

- Rightmost derivations $\Rightarrow_{rm}$
  - Only the rightmost nonterminal in any sentential form is replaced at each step

E⇒ -E ⇒ -(E)⇒ -(E+E) ⇒ -(E**+id**) ⇒ -(**id**+**id**)

Parse Trees and Derivations

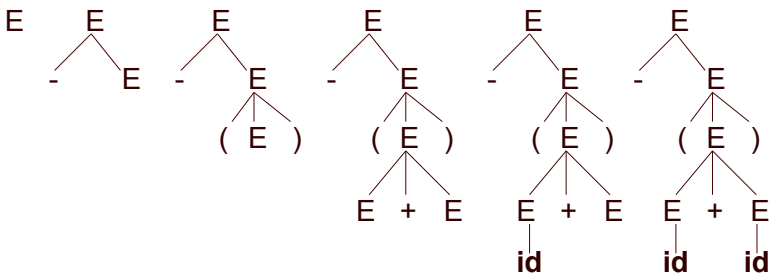
- Graphical representation of a derivation
- Filters out the choice regarding replacement order
  - Ignores variations in the order in which symbols in sentential forms are replaced
    - Many-to-one relationship between derivations and parse trees
  - Every parse tree has associated with it a unique leftmost and a unique rightmost derivation
    - We should not assume that every sentence necessarily has one parse tree only or only one leftmost or rightmost derivation!
      - AMBIGUITY

Parse Tree Definition

- Formally, given a context-free grammar, a parse tree is a tree with the following properties:
  - The root is labeled by the start symbol
  - Each leaf is labeled by a terminal or by ε (empty string)
  - Each interior node is labeled by a nonterminal
  - If A is the nonterminal labeling some interior node and X_1, X_2, …, X_n are the labels of the children of that node from left to right, then $A \rightarrow X_1 X_2 \ldots X_n$ is a production.
    - Here X_1, X_2, …, X_n stand for a symbol that is either a terminal or a nonterminal.

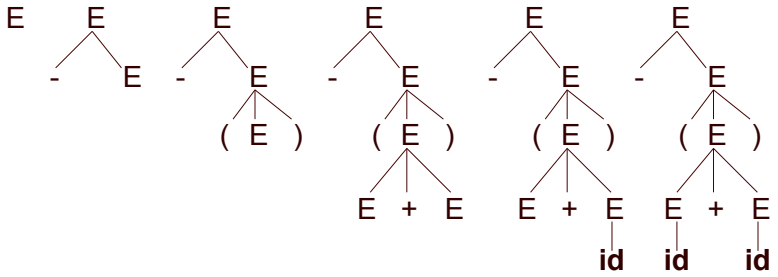
Building parse tree for derivations

E⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(**id**+E) ⇒ -(**id**+**id**)



Building parse tree for derivations

E⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(E+id) ⇒ -(**id+id**)



Parse Trees and Grammar Ambiguity

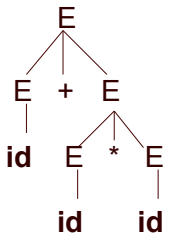
Derive the leftmost derivation of sentence **id+id*id**

E→ E +E | E * E |(E) |-E | **id**

Leftmost Derivation 1

E→ E +E | E * E |(E) |-E | **id**

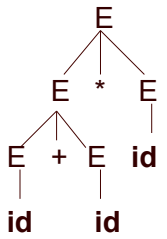
E⇒ E + E⇒ **id** + E⇒ **id** + E * E⇒ **id** + **id** * E⇒ **id+id*id**



Leftmost Derivation 2

E→ E +E | E * E |(E) |-E | **id**

E⇒ E * E⇒ E + E *E ⇒ **id** + E * E⇒ **id** + **id** * E⇒ **id+id*id**



Grammar Ambiguity

A grammar that produces more than one parse trees for some sentence it said to be ambiguous!

Summary

- How can we verify that a sentence belongs to the language generated by grammar G?
  - Parse tree
  - Derivation
  - Or formal proof

Lecture Outline

- Role of parser
- Context-free grammars
  - Review definition, derivations and parse trees
- Writing grammars
  - Ambiguity, eliminating left recursion, left factoring
- Top-down Parsing
  - Recursive-descent parsing, non-recursive predictive parsing, construction of predictive parser

Writing Grammars

- Grammars are more powerful than regular expressions
  - A construct that can be described by a regular expression can also be described by a grammar
    - Construct NFA from r.e.
    - Convert NFA into a grammar using simple construction rules
      - For each state i create a nonterminal symbol A_i
      - If state i has transition to state j on symbol a, then introduce production $A_i \rightarrow aA_j$
      - If state i has a transition to state j on input ε, then introduce production $A_i \rightarrow A_j$
      - If state i is an accepting state then introduce $A_i \rightarrow \varepsilon$
      - If state i is the start state, make A_i the start symbol of the grammar
  - But not vice versa!

Grammars and Lexical Analysis

- Why user regular expressions to define lexical syntax of a language?
  - Lexical rules are frequently simple
  - Regular expressions provide a notation that is more suitable for tokens than grammars
  - More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars
  - Separating the syntactic structure into lexical and non-lexical parts provides a convenient way of modularizing the front-end of the compiler into two manageable-sized components

Eliminating Ambiguity

- Sometimes an ambiguous grammar can be rewritten to eliminate ambiguity

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

Derive the parse tree for the following nested if statement
(answer in [ALSU07], page 211)

if E1 **then if** E2 **then** S1 **else** S2

Eliminating Ambiguity

- Rule for matching if-else
  - Match each **else** with the closest previous unmatched **then**
- Statement appearing between a then and else must be 'matched'

stmt → matched_stmt
| unmatched_stmt

matched_stmt → **if** *expr* **then** matched_*stmt* ***else*** *matched_stmt*
unmatched_stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** matched_*stmt* **else** unmatched_*stmt*

Eliminating Ambiguity

- Derive the parse tree for the following string using the unambiguous grammar
  (answer in [ALSU07],page 210)

if E1 **then if** E2 **then** S1 **else** S2

Eliminating Left Recursion

- A grammar is left recursive if
  - it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α

    $A \rightarrow A\alpha \mid \beta$
  - For instance, which productions are left recursive?
    - $E \rightarrow Ex$
    - $E \rightarrow (E)$
- Top-down parsing cannot handle left-recursive grammars
  - Need to eliminate it

Eliminating Left Recursion

General case: $A \rightarrow A\alpha \mid \beta$

Example: *expr* → *expr* + *term* | *stmt*

A is *expr*
α is + *term*
β is *stmt*

Eliminating Left Recursion

General case: $A \rightarrow A\alpha \mid \beta$

Non-left-recursive: $A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \varepsilon$

R is a new nonterminal
Right-recursive grammar

Eliminating Left Recursion

General case: $A \rightarrow A\alpha \mid \beta$

Non-left-recursive: $A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \varepsilon$

Eliminate the immediate left recursion from the grammar G

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

Eliminating Left Recursion

E → E + T | T
T → T * F | F
F → (E) | **id**

First, eliminate the immediate left recursion that occurs for E

α is +T
β is T

E → TE`
E` → +TE` | ε

New transformed grammar so far is

E → TE`
E` → +TE` | ε
T → T * F | F
F → (E) | **id**

Eliminating Left Recursion

E → TE`
E` → +TE` | ε
T → T * F | F
F → (E) | **id**

Second, eliminate the immediate left recursion that occurs for T

α is *F
β is F

T → FT`
T` → *FT`| ε

Final transformed grammar is

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

Eliminating Left Recursion

- This technique does not eliminate left recursion involving derivations of two or more steps

S → A a | b
A → Ac | Sd | ε

In this case, S is left-recursive because

S⇒Aa⇒Sda

Eliminating Left Recursion

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

  for j:= 1 to i-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_i productions

end

In-class Exercise

Order the nonterminals
S, A

S → A a | b
A → Ac | Sd | ε

i = 1
Immediate left-recursion among S productions?
No, so nothing happens
i = 2
we substitute the S in A-production A → Sd to obtain the following A-productions
A → Ac | Aad | bd | ε
eliminate the left recursion among A productions
A → bdA' | A'
A → cA' | adA' |ε

Solution

S → A a | b
A → bdA` | A`
A` → cA` | adA` | ε

Left Factoring

- It is a grammar transformation that is useful for producing a grammar suitable for predictive parsing
  - Idea:
    - Suppose that it is not clear which of two alternative productions to use to expand a nonterminal A
    - Rewrite the A-productions to defer this decision when more input is seen (be able to make a right choice then)

Left factoring

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

Suppose that we have identified token **if**

Which production for stmt to expand?

Left factoring

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

Suppose input begins with a nonempty string derived from α, then we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$

We could defer the decision by expanding A to αA`

Then after seeing the input derived from α we expand A` to β_1 or β_2

Left factoring

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$ → $A \rightarrow \alpha A`$, $A` \rightarrow \beta_1 | \beta_2$

See [ALSU07], page 214, Algorithm 4.21

Left factoring Example

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

Solution

stmt → **if** *expr* **then** *stmt* **stmt`**
| a
stmt` → **else** *stmt* | ε
expr → b