Compiler Design
Syntax Analysis
Context Free Grammars

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

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

Outline

- The Formal Definition of a Context Free Grammar
- Notational Conventions
- Derivations
- Parse Trees and Derivations
- Ambiguity
- Verifying the Language Generated by a Grammar
- Context-Free Grammars versus Regular Expressions

Grammars

- to systematically describe syntax of programming language constructs
  - expressions
  - statements
- stmt->**if** (expr) stmt **else** stmt
- notions
  - parsing
  - derivations
    - the order in which productions are applied during parsing

The Formal Definition of a Context Free Grammar

- terminals
  - basic symbols from which strings are formed
  - token name = terminal
  - output of the lexical analyzer
  - e.g.: if, else, (and)
- non-terminals
  - syntactic variables that denotes sets of strings
  - e.g.: stmt, expr
  - define the language generated by the grammar
  - impose a hierarchical structure on the language
  - key to syntax analysis and translation

The Formal Definition of a Context Free Grammar

- start symbol
  - the set of its strings denotes the language generated by the grammar
  - conventionally are listed first
- productions of a grammar
  - specify the manner in which terminals and non-terminals are combined to form strings
  - consists in:
    - a non-terminal called head or left side of the production
    - the symbol -> or ::=
    - body or right side
      - contains zero or more terminals or non-terminals
      - describe one way the strings of the non-terminal at the head can be constructed

Grammar for Simple Arithmetic Expressions Example

- terminals
  - id + - * / ( )
- non-terminals
  - expression -> expression + term
  - expression -> expression – term
  - expression -> term
  - term -> term * factor
  - term -> term / factor
  - term -> factor
  - factor -> (expression)
  - factor -> id
- start symbol
  - expression

Notational Conventions

- Terminals
  - Lowercase letters: a, b, c
  - Operator symbols: +, *, …
  - Punctuation symbols: ( ) , ;
  - Digits 0,1,2,3,..9
  - Boldface strings **id**, **if**
- Non-terminals
  - Uppercase letters early in the alphabet: A, B, C
  - Letter S is usually the start symbol
  - Lowercase, italic: *expr, stmt*
  - expressions - E, terms - T, factors - F

Notational Conventions

- Grammar symbols
  - either non-terminals or terminals
  - alphabet late uppercase letters: X, Y, Z
- Strings of terminals
  - alphabet late lowercase letters: u,v,…,z
- Strings of grammar symbols
  - Lowercase Greek letters: α,β,γ
  - A->α
    - A is the head
    - α is the body

Notational Conventions

- Set of productions
  - $A \rightarrow \alpha_1, A \rightarrow \alpha_2 \ldots, A \rightarrow \alpha_k$
  - common head A
  - A productions
  - $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$
  - $\alpha_1 | \alpha_2 | \ldots | \alpha_k$ - alternatives for A
- The head of the first production is the start symbol

Concise Grammar Example

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

- E, T, F are non-terminals
- E is start symbol
- the remaining symbols are terminals

Derivations

- productions rewriting rules
- to begin with the start symbol
- to replace a non-terminal by the body of its productions
- top-down parsing
  - derivational view
- bottom-up parsing
  - rightmost derivations
  - the right most terminal is rewritten at each step

Derivation Example

- E -> E + E | E * E | -E | (E) | id
- E -> -E
  - replacement is noted E =>-E
  - is read as E derives –E
- E -> (E)
  - E * E => (E) * E or
  - E * E => E * (E)
- E => -E => -(E) => -(**id)**
  - derivation of –(**id)** from E
  - -(**id**) is one particular instance of an expression

General Definition of Derivation

- given non-terminal A in the middle of $\alpha A\beta$
- $A \to \gamma$ (A->γ)
- $\alpha A\beta \Rightarrow \alpha\gamma\beta$
- => means derives in one step
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$
  - rewrites α_1 to α_n
  - α_1 derives α_n

General Definition of Derivation

- $\stackrel{*}{=}>$ means "derives in zero or more steps"
  - $\alpha \stackrel{*}{=}> \alpha$
  - $\alpha \stackrel{*}{=}> \beta$ and $\beta => \gamma$ then $\alpha \stackrel{*}{=}> \gamma$
- $\stackrel{+}{=}>$ means "derives in one or more steps"
- if
  - S is the starting symbol of a grammar G
  - $S \stackrel{*}{=}> \alpha$
- then
  - α is the sentential form of G

Sentential Form

- may contain terminals and non-terminals
- may be empty
- sentence of G is a sentential form with no non-terminals
- the language generated by a grammar is a set of sentences
- L(G) – the language generated by G
- a string of terminals w is in L(G) iff w is a sentence of G ($S \overset{*}{\Rightarrow} w$)

Context Free Language

- a language which can be generated by a grammar
- if two grammars generate the same language then they are equivalent
- -(id+id) is a sentence of the grammar

because of the derivation

E=>-E=>-(E+E)=>-(id+E)=>-(id+id)

$E \overset{*}{\Rightarrow} -(id+id)$

Derivation Choices

- E=>-E=>-(E+E)=>-(id+E)=>-(id+id)
- E=>-E=>-(E+E)=>-(E+id)=>-(id+id)
- the order of replacement is different
- leftmost derivations
  - the leftmost terminal in α is replaced
  - $\alpha \underset{lm}{\Rightarrow} \beta$
- rightmost derivations
  - the rightmost terminal in α is replaced
  - $\alpha \underset{rm}{\Rightarrow} \beta$

Derivation Examples

- $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$
- $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$
- every leftmost step is denoted by
  - $\omega A\gamma \rightarrow \omega\delta\gamma$
  - ω – has terminals only
  - γ - string of grammar symbols
- α derives β
  - $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$
- $S \rightarrow \alpha$
  - α – left sentential form of the grammar
- rightmost derivations = canonical derivations

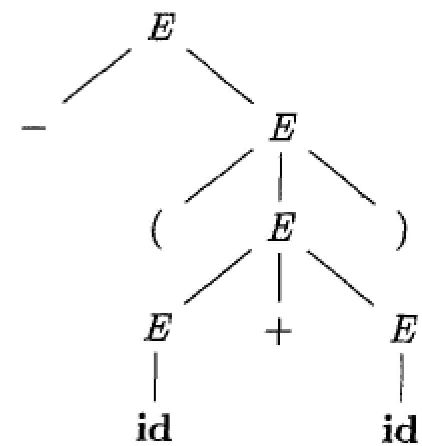
Parse Trees and Derivations

- graphical representation of a derivation
- filters out the order in which productions replace non-terminals
- each interior node of a parse tree represents the application of a production
- the interior node is labeled with the non-terminal A in the head
- the children are labeled from left to right by symbols in the body of the production by which A was replaced during derivation

Parse Trees and Derivations

- leaves of a parse tree are represented by terminals or non-terminals
- from left to right represent
  - a sentential form
  - the frontier of the tree
- $\alpha_1 => \alpha_2 => \ldots => \alpha_n$ where $\alpha_1 = A$
  - for each sentential form α_i we can construct a parse tree whose frontier is α_i

Parse Trees and Derivations



- parse tree for –(id+id)
- Induction:
- suppose we build parse tree

with yield

- $\alpha_{i-1} = X_1 X_2 \ldots X_k$
- where X_i=non-terminal of terminal
- α_i is derived from α_{i-1} by replacing X_j by
- $\beta = Y_1 Y_2 \ldots Y_m$
- $X_j \rightarrow \beta$
- $\alpha_i = X_1 X_2 \ldots X_{j-1} \beta X_{j+1} \ldots X_k$

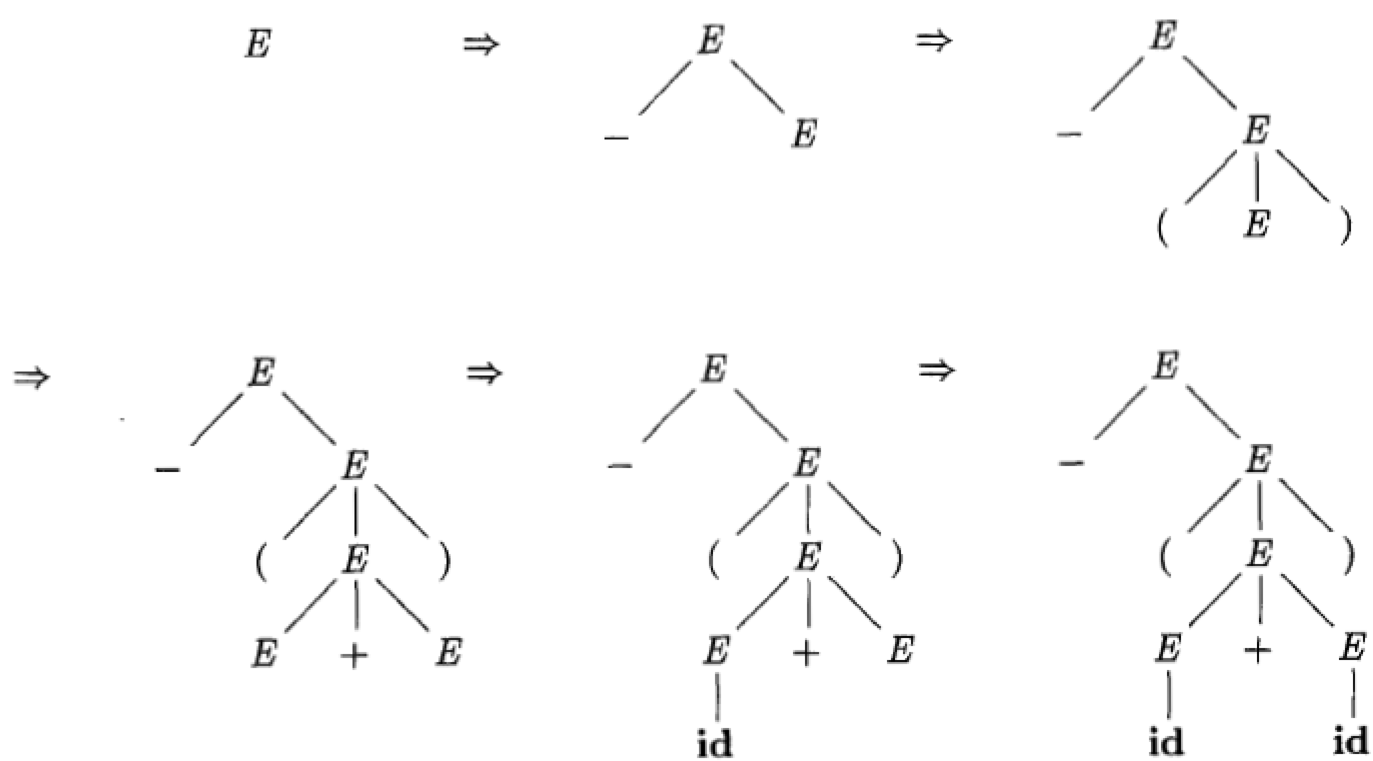
Parse Trees and Derivations

- find the j-th leaf from the left in the current parse tree (X_j)
- give this leaf m children labeled $Y_1 Y_2 \ldots Y_m$ from the left to right
- if m=0 then $\beta=\varepsilon$
  - we give one child labeled ε

Parse Trees and Derivations

- many to one relationship between
  - derivations
  - parse trees
- one to one relationship between
  - leftmost or rightmost derivations
  - parse trees
    - filter out the variations in the order

Sequence of parse trees for derivation

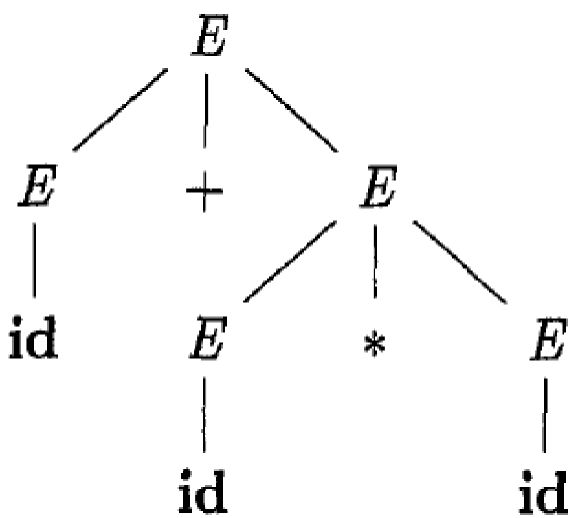


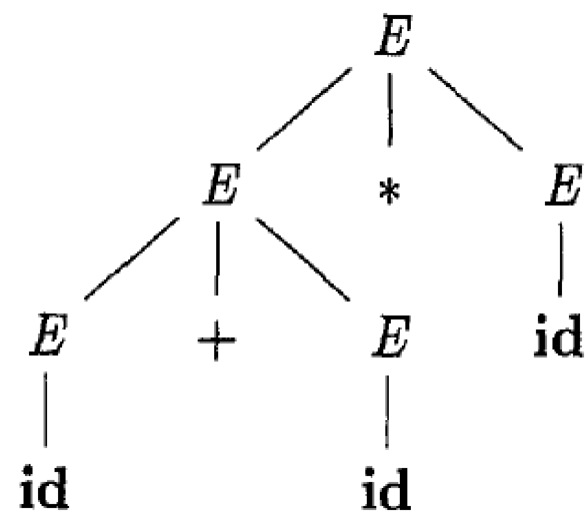
Ambiguity

- ambiguous grammar
  - grammar that produces more than one parse tree for some sentence
- id + id * id

E -> E + E -> id + E -> id + E * E -> id + id * E -> id + id * id	E -> E + E -> E + E * E -> id + E * E -> id + id * E -> id + id * id

Ambiguity





Verifying the Language Generated by a Grammar

- compiler designers rarely do so for a complete programming language grammar
- to reason whether a given set of productions generates a particular language
- troublesome constructs can be studied
  - constructing a concise abstract grammar
  - analyzing the language that it generates
- a proof for a grammar G generates a language L
  - every string generated by G is in L
  - every string in L can be generated by G

Example

- S->(S) S | ε
  - generates all strings of balanced parentheses
- to show that
  - any string derivable from S is balanced
  - every balanced string is derivable from S
- using an inductive proof on a number of steps n in a derivation

Any String Derivable from S is Balanced

- Basis
  - n=1
  - the only string of terminals derivable from S in one step is the empty string
  - the empty string is balanced

Any String Derivable from S is Balanced

- Induction
  - we assume that all derivations of fewer than n steps produce balanced sentences
  - let us consider a leftmost derivation of exactly n steps
  - $S \underset{lm}{=>} (S)S \overset{*}{\underset{lm}{=>}} (x)S \overset{*}{\underset{lm}{=>}} (x)y$
  - x, y
    - take fewer than n steps
    - are balanced – by hypothesis
  - so (x)y is balanced
    - the number of left and right parentheses are equal
    - every prefix has a no of left parentheses >= no of right parentheses

Every Balanced is String Derivable from S

- Basis
  - if the length is 0 then it must be the empty string
  - the empty string is balanced
- Induction
  - every balanced string has a length
  - we assume that any string of length less than 2n is derivable from S
  - let us consider a balanced string w of length 2n, n>=1

Every Balanced is String Derivable from S

- Induction
  - w begins with left parenthesis
  - let (x)
    - be the shortest non-empty prefix of w
    - having equal number of left and right parentheses
  - w=(x)y, where both x and y
    - are balanced
    - are of length less than 2n
    - are derivable from S
  - we can find a derivation
    - $S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$
  - proving that w=(x)y is also derivable from S

Context Free Grammars Versus Regular Expressions

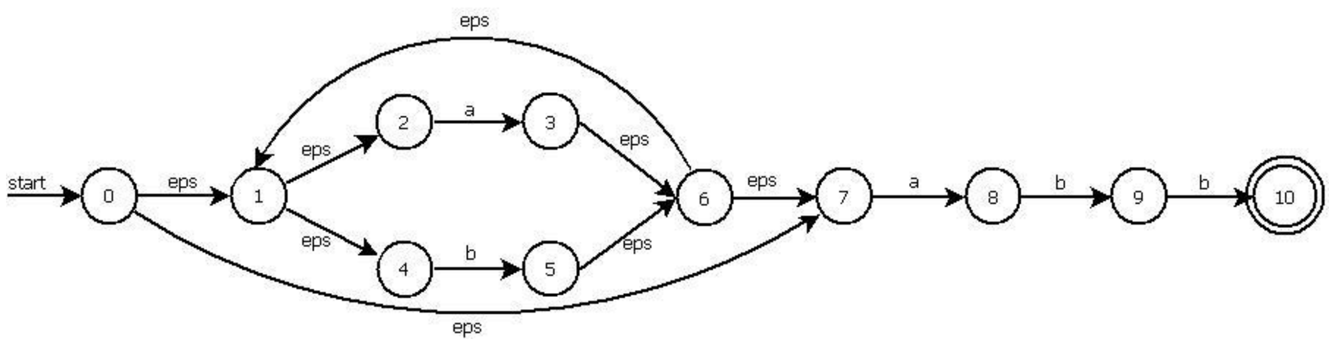
- grammars are more powerful notations than regular expressions
- any construct that can be described by a RE can be described by a grammar
- not vice-versa

NFA to Grammar

- for each state i we create a non-terminal A_i
- a transition from i to j on input a is translated as $A_i \to aA_j$
- a transition from i to j on input ε is translated as $A_i \to A_j$
- if i is an accepting state $A_i \to \varepsilon$
- if i is the start state make A_i the start symbol of the grammar

Example

- (a|b)*abb



- $A_0 \to aA_0 \mid bA_0 \mid aA_8$
- $A_8 \to bA_9$
- $A_9 \to bA_{10}$
- $A_{10} \to \varepsilon$

Example

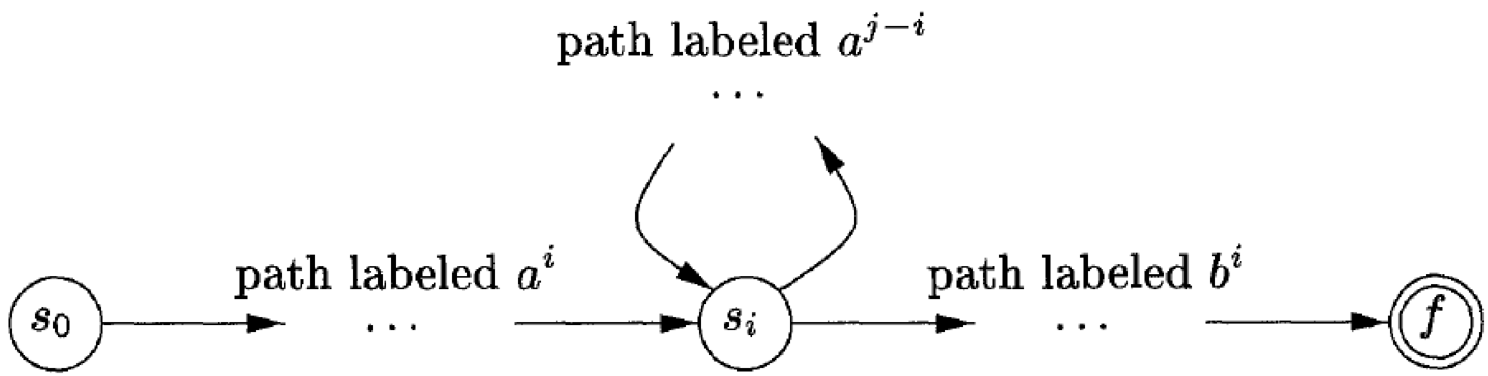
- $L=\{a^n b^n | n>=1\}$
- typical language example that
  - has an equal number of a and b's
  - can be described by a grammar
  - can not be described by a regular expression

Example

- let us suppose that L is defined by a regular expression
- we construct a DFA D with a finite number of states k to accept L
- D has only k states

Example

- for an input with more than k a's
- D must enter some state twice, say s_i
- the path from s_i to itself is labeled with a^{j-i}
- $a^i b^i$ is in the language so there must be a path labeled b^i from s_i to an accepting state f
- there is also a path from s_0 through s_i to f labeled $a^j b^i$
- so D accepts $a^j b^i$ also which is not in the language



Conclusion

- finite automata cannot count !!!
- the automata can not keep the count of a's before it sees the b's

Bibliography

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