

Compiler: Theories, Techniques and Tools

Georges Edouard KOUAMOU

National Advanced School of Engineering

Objectives

- Introduce students to the principles of compilation.
 - The ideas and techniques developed in this area are so general and fundamental that a computer scientist (and even a non-computer scientist) will use them very often during his career (data processing, search engines, etc.).
- To study the algorithms and data structures inherent in the implementation of compilers: lexical analysis, Syntax Analysis, semantic analysis, code generation.
- Understand how a compiler is written to allow students to better understand the "constraints" imposed by different languages when writing a program in a high-level language.

Contents

- Introduction
- Lexical analysis
 - Finite state machine
 - Regular expressions
 - Lexical tokens
- Parsing (syntax analysis)
 - Grammars
 - Topdown parsing
 - Bottom up parsing
- Attributed Grammars (add semantics to a context-free grammar)
- Implementation with SableCC

References

- Seth D. Bergmann. **Compiler Design: Theory, Tools, and Examples**, February 2016, Course notes
- Andrew W. Appel. **Modern Compiler implementation in Java**. 2nd Edition, Cambridge University Press, 2004
- Donald Knuth. **Semantics of Context-Free Languages**. MATHEMATICAL SYSTEMS THEORY, Vol. 2, No. 2, Published by Springer-Verlag, New York Inc.

Introduction

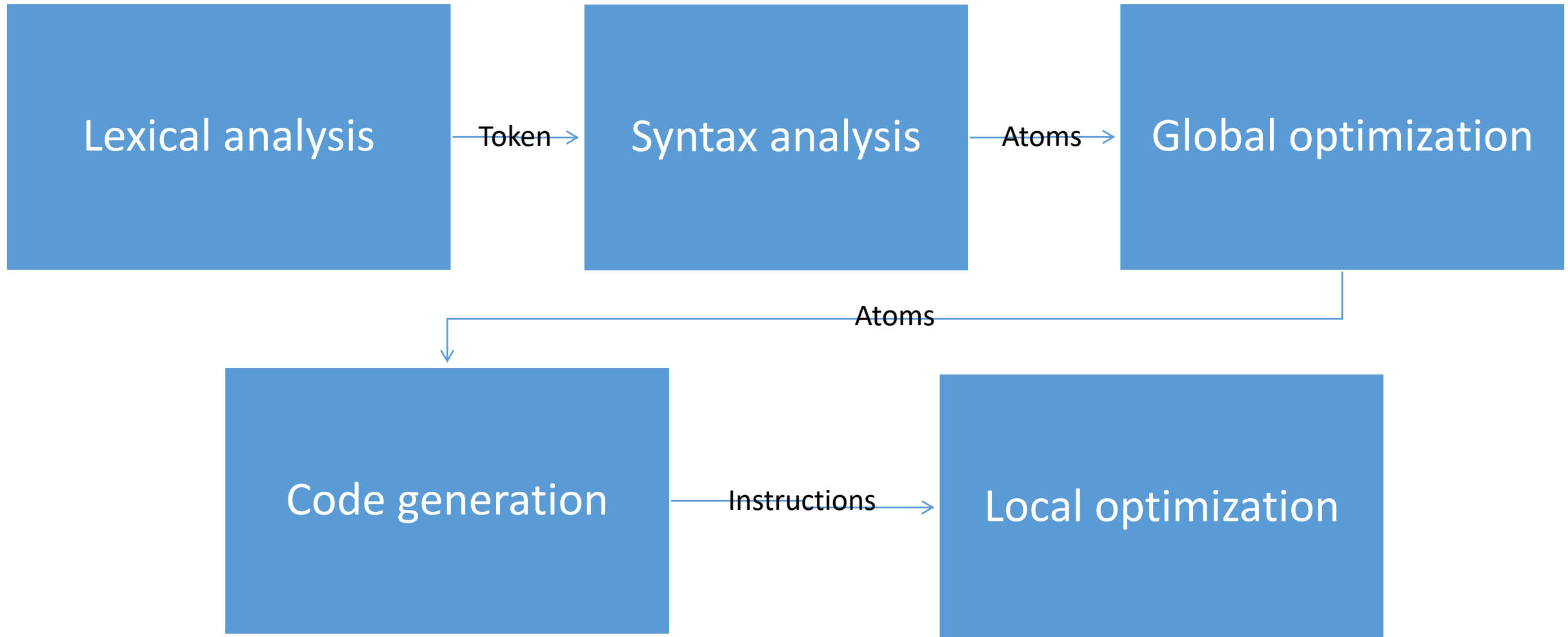
Definition (Compiler)

- The computer's CPU is capable of executing very simple, primitive operations = machine (or assembly) language
 1. add two numbers stored in memory
 2. move numbers from one location in memory to another
 3. move information between the CPU and memory
- How do the computer executes a complex instruction: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
 - Use of a software translator
- A **compiler** accepts (complex) statements and translates them into sequences of machine language operations
- An **interpreter** is software which serves a purpose very similar to that of a compiler But it carries out the computations specified in the source program

Phases of a compiler

- input : string of characters
- 3 phases: Lexical analysis, Syntax analysis, Code generation
- Addition phase : optimization is employed to produce efficient object code
- Lexical analysis or **Scanner**: split the string of characters into **word** known as *token* or *lexeme*
 - Examples: keywords, operators, constants, identifiers
- Syntax analysis or **Parser**: checks the proper syntax and build the *syntax tree*
 - The underlying structure of the source program
- Many compilers also include a phase for semantic analysis
 - The data types are checked, and type conversions are performed when necessary
- Code generation: the compiler produces
 - an intermediate form, known as byte code. The case of Java Compiler or .Net
 - Or a native code for a particular machine (executable binary code)

Phases of a compiler



Implementation techniques

- A compiler is a software tool
 - It must be written using a language
- bootstrapping
 - a small subset of the source language is implemented and used to compile a compiler for the full source language, written in the source language itself
- Cross compiling
 - an existing compiler can be used to implement a compiler for a new computer.
- Intermediate form can reduce the workload of the compiler writer
 - a language somewhere between the source high-level language and machine language
 - one needs only one translator for each high-level language to the intermediate form and only one translator (or interpreter) for the intermediate form on each computer

Some Applications of the compiler techniques

- Internet
 - Web browser : translate HTML text into graphical objects
 - PHP interpreter
- Data base Management Systems (DBMS)
 - Intrepretation and execution of SQL instructions
- Search
 - Advanced search in text editors, office software, and so on ...
- XML/JSON framework
 - Encoding/decoding text stream into XML structure
 - Translate SOAP message into object invocation
- Network, Sensors
 - Decoding data flow

Lexical analysis

breaking the input into individual words or “tokens”

Formal language and automata theory as design tools of the scanners

Formal language

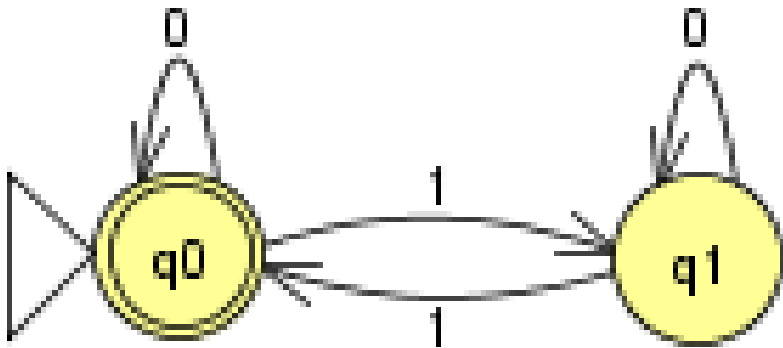
- An **alphabet** Σ is a set of symbols
 - Example: binary alphabet $=\{0, 1\}$
- A **string** is a list of characters from a given alphabet
 - the order in which the characters are listed is important
 - Example: “101” is different from “110”
- ϵ denotes the **null string** (string with 0 character)
- A (formal) language is a set of strings from a given alphabet
- Examples of language from the alphabet $\{0, 1\}$
 - $\{0, 10, 1011\}$
 - $\{ \}$ or \emptyset
 - $\{\epsilon, 0, 00, 000, 0000, 00000, \dots\}$
 - The set of all strings of zeroes and ones having an even number of ones (**infinite set**)
- **Problem:** How can we specify the strings in an infinite (or very large) language?

Finite State Machines (FSM)

- The study of (theoretical) finite state machine is called **automata theory**
 - automaton is just another word for machine
- An automaton $A = (\Sigma, Q, d, F, \delta)$ where
 - Σ is the alphabet
 - Q is the finite set of states
 - $d \in Q$ is the initial or starting state
 - $F \subseteq Q$ is the set of final or accepting state
 - $\delta: Q \times \Sigma \rightarrow Q$ is the state transition function
- a string is accepted when the automaton reaches a final state after the entire input string has been read.
- The language recognized by an automaton is the set of strings that it accepts.
 - $L(A) = \{u \in \Sigma^* \mid \exists q \in F, \delta^*(d, u) = q\}$

Representations

State diagram representation



Example: Even parity Checker

This machine accepts any string of zeroes and ones which contains an even number of ones (which includes the null string)

Table representation

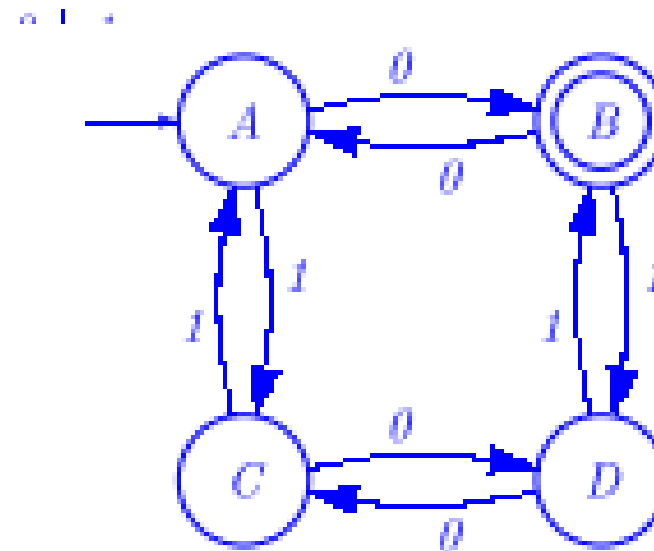
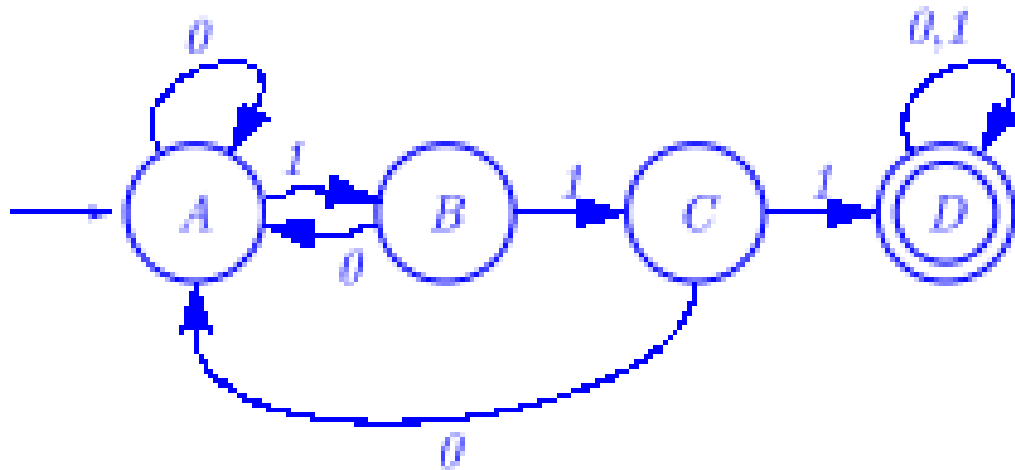
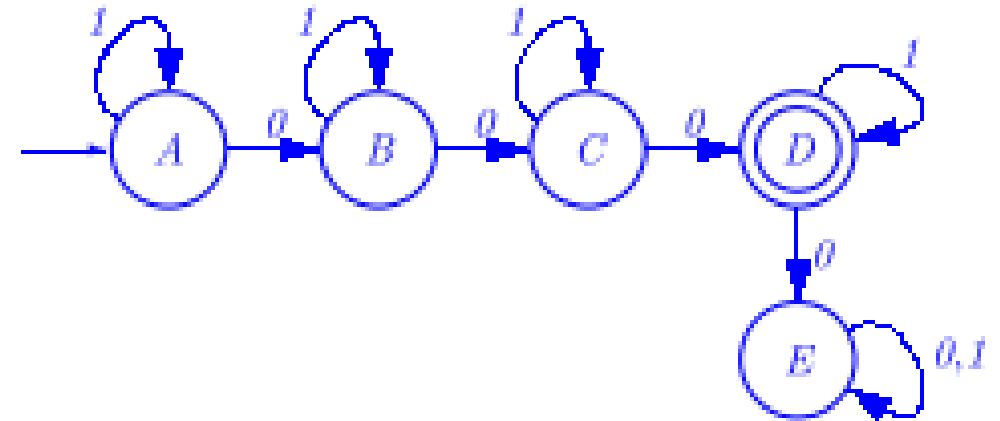
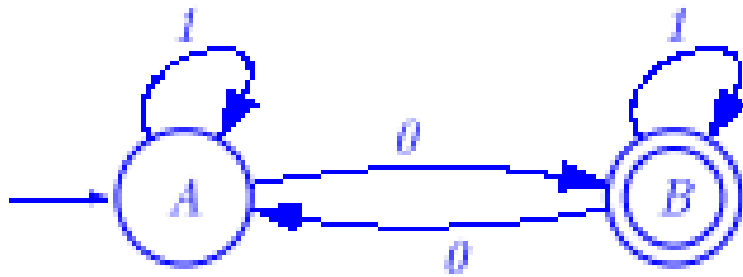
	0	1
q0	q0	q1
q1	q1	q0

- Each state of the machine is represented by a circle
- The transition function is represented by arcs labeled by input symbols leading from one state to another.
- The accepting states are double circles
- The starting state is indicated by an arc with no state at its source (tail) end

Exercises

- Show a finite state machine, in either state graph or table form for each of the following languages (in each case the input alphabet is $\{0,1\}$):
 1. Strings containing an odd number of zeros
 2. Strings containing three consecutive ones
 3. Strings containing exactly three zeros
 4. Strings containing an odd number of zeros and an even number of ones.

Solutions



Finite automata

- A **nondeterministic finite automaton** (NFA) is one that has a choice of edges—labeled with the same symbol—to follow out of a state
- In a **deterministic finite automaton** (DFA), no two edges leaving from the same state are labeled with the same symbol
- From the *table representation it is easier to ensure that* the machine is completely specified and deterministic
 - There should be exactly one entry in every cell of the table
- Finite automata are used to implement the lexical token as a computer program
- **Regular expressions** are convenient for specifying lexical tokens

Lexical Tokens and examples

- A **lexical token** is a sequence of characters that can be treated as a unit in the grammar of a programming language
- Example
 - ID foo n14 last
 - NUM 73 0 00 515 082
 - REAL 66.1 .5 10. 1e67 5.5e-10
 - IF if
 - COMMA ,
 - NOTEQ !=
 - LPAREN (
 - RPAREN)

Regular expressions

- RE are formulas or expressions consisting of three possible operations on languages: **union, concatenation, and Kleene star**
- **Union (or alternation)** Since a language is a set, this operation is the union operation as defined in set theory
 - On languages is designated with a '+'.
 $L_1 + L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$
- **Concatenation** of strings : the juxtaposition of two strings forming a new string
 - Designated by '.' but **can be omitted**
 - Example: $abc \cdot ba = abcba$
 - any string concatenated with the null string is that string itself: $s \cdot \epsilon = s$
- **Concatenation** of two languages is that language formed by concatenating each string in one language with each string in the other language.
 - Example: $\{ab, a, c\} \cdot \{b, \epsilon\} = \{ab \cdot b, ab \cdot \epsilon, a \cdot b, a \cdot \epsilon, c \cdot b, c \cdot \epsilon\} = \{abb, ab, a, cb, c\}$
 - $L_1 L_2 = \{uv \mid u \in L_1 \text{ et } v \in L_2\}$
 - **Requirement:** both languages must be defined on the same alphabet

Regular expressions

- **Kleene *** (**repetition**) This operation is a unary operation and is often called closure
 - Kleene * generates zero or more concatenations of strings from the language to which it is applied.
 - $L^* = \bigcup_{i \geq 0} L^i = L^0 + L^1 + L^2 + \dots + L^n + \dots$
- Precedence
 - concatenation takes precedence over union
 - Kleene * takes precedence over concatenation
 - Otherwise the precedence may be specified with parentheses
- *Union and concatenation are associative*
- *Concatenation is distributive w.r.t union*
- Example
 - $(a + b)^*$ => the set of all strings of 'a' and 'b'
 - $1(0 + 1)^*0$ => the set of all strings of zeros and ones which begin with a 1 and end with a 0
 - $(a + b) \cdot c = ac + bc$

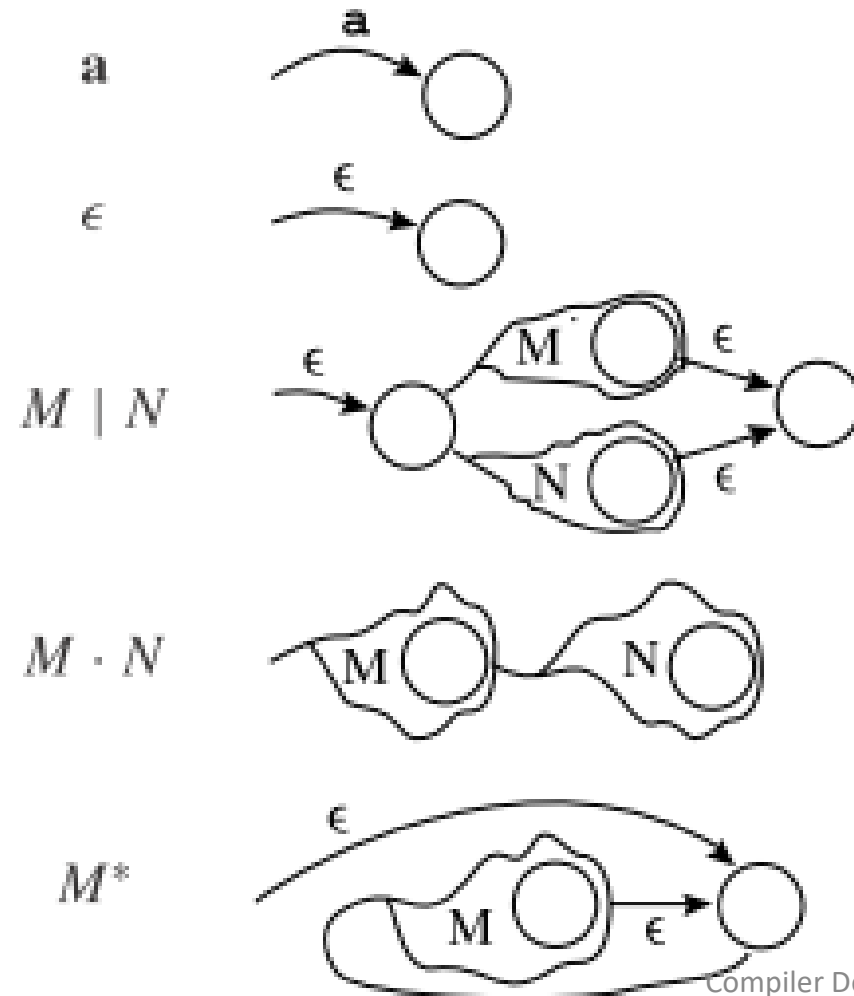
Regular expressions notation

- a an ordinary character stand for itself
- ϵ the empty string
- $M|N$ alternation, choosing from M or N
- $M \cdot N$ (ou MN) concatenation, an M followed by an N
- M^* repetition zero or more times
- M^+ repetition one or more times
- $M^?$ optional, zero or one occurrence of M
- $[a-zA-Z]$ character set alternation
- $.$ A period stands for any signe character except newline

Regular expression ambiguity

- These rules are a bit ambiguous.
 - Does **if8** match as a single identifier or as the two tokens **if** and **8**?
- **Longest match:** The longest initial substring of the input that can match any regular expression is taken as the next token.
- **Rule priority:** For a particular longest initial substring, the first regular expression that can match determines its token-type
 - The order of writing down the regular-expression rules has significance

Converting a RE to an NFA (Thompson's construction)



M^+ constructed as $M \cdot M^*$

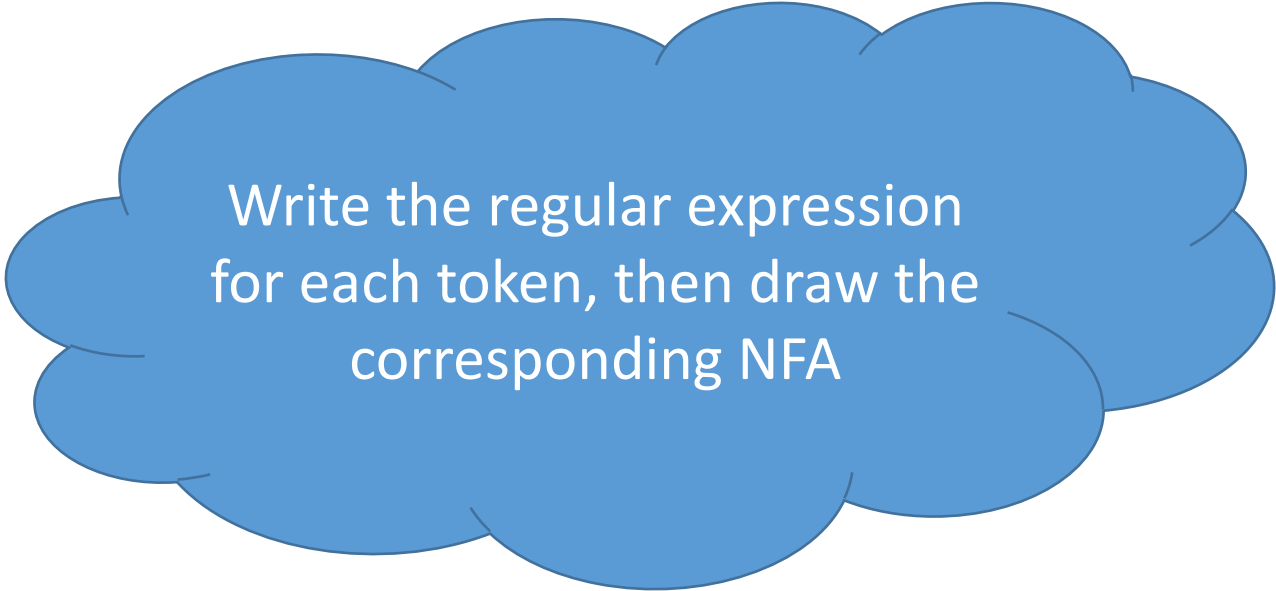
$M?$ constructed as $M \mid \epsilon$



"abc" constructed as **$a \cdot b \cdot c$**

Example

- ID
- NUM (Integer or decimal)
- REAL
- IF



Write the regular expression
for each token, then draw the
corresponding NFA

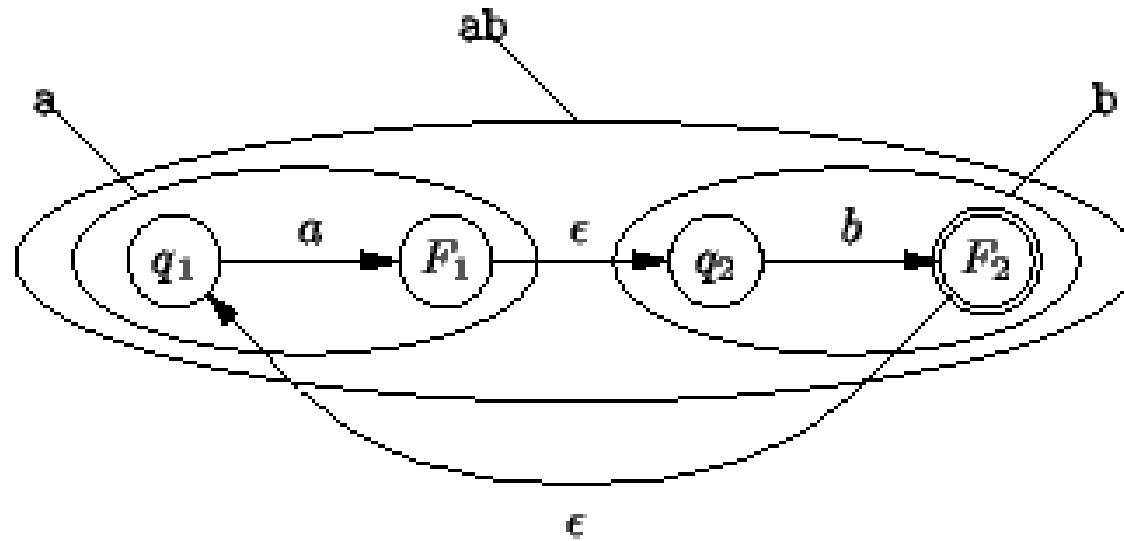
Converting an NFA to a DFA

- Definition (ϵ -closure)
 - Let S be a set of states. **closure** (S) is the set of states that can be reached from a state in S without consuming any of the input, that is, by going only through ϵ -edges
 - **closure**(S) is the smallest set T such that $T = S \cup (\bigcup_{s \in T} \mathbf{edge}(s, \epsilon))$
- Calculation of T
 - $T \leftarrow S$
 - **repeat** $T' \leftarrow T$
 - $T = T' \cup (\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon))$
 - **until** $T = T'$
- Algorithm to convert NFA to DFA
 - Initial state : $\text{closure}(\{d\})$ where d is the initial state of the NFA
 - The transition from S with a symbol c : $\delta(S, c) = \bigcup_{s \in S} (\text{closure}(\delta(s, c)))$
 - S is a final state if $S \cap F \neq \emptyset$

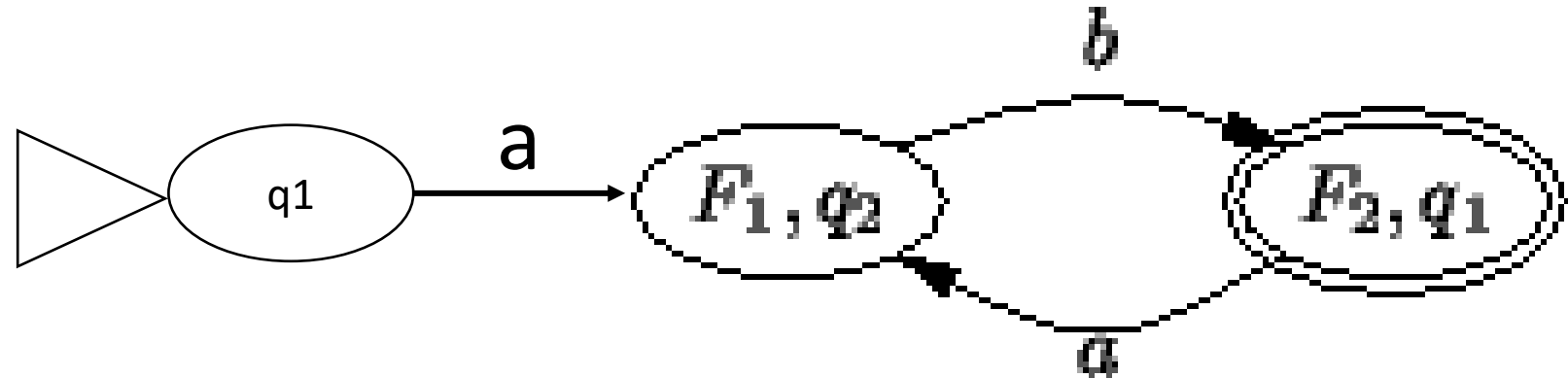
Example

Regular expression $(ab)^*$

Corresponding NFA

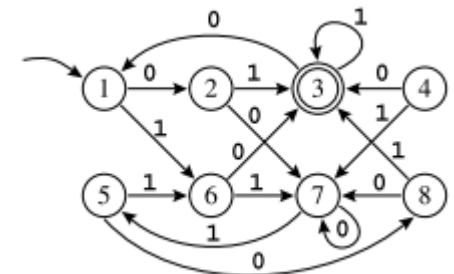


DFA obtained from NFA



Exercises

- In the book From Appel Page 34-35
- Exercise 2.1: write regular expression
 - From DFA to regular expression
- Exercise 2.4: convert regular expression to NFA
- Exercise 2.5 : convert NFA to DFA
- Exercise 2.6: algorithm to minimize a DFA



Parsing/Syntax analysis

Syntax: the way in which words are put together to form phrases, clauses, or sentences

For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token-types returned by the lexical analyzer

Context-free Grammars

- We have already seen two ways of formally specifying a language: regular expressions and finite state machines.
- We will now define a third way of specifying languages by using a grammar
- A **grammar** is a list of rules which can be used to produce or generate all the strings of a language, *and which does not generate any strings which are not in the language*
- **Formal definition:** A context-free grammar $G = (\Sigma, V, R, S)$ where
 - Σ is the input alphabet, a finite set of characters, the input symbols.
 - V is a finite set of symbols, distinct from the terminal symbols, called nonterminal symbols
 - $S \in V$ is the starting nonterminal or Axiom.
 - $R \subseteq V \times (\Sigma \cup V)^*$ A finite list of rewriting rules, also called productions, which define how strings in the language may be generated. Each of these rewriting rules is of the form $A \rightarrow \beta$, where $A \in V, \beta \in (\Sigma \cup V)^*$.

Derivations and Parse Trees

- A **derivation** is the substitution of a nonterminal by the right hand side (RHS) of a production
- A **leftmost derivation** is one in which the leftmost nonterminal symbol is always the one expanded
- a **rightmost derivation** is one in which the rightmost nonterminal is always the next to be expanded
- The language specified by the grammar G is $L(G)$ defined as:
 - $L(G) = \{u \in \Sigma^* \mid S \Rightarrow^* u\}$
- A **parse tree** is made by connecting each symbol in a derivation to the one from which it was derived
- A **grammar is ambiguous** if it can derive a sentence with two different parse trees
 - Ambiguity can be usually eliminated by transforming the grammar.

Examples

Grammar

Exemple of derivation

$S \rightarrow 0S0|1S1|0|1$

$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001S100 \Rightarrow 0010100 \quad 0010100 \in L(G)$

$S \rightarrow ASB$

$S \rightarrow \epsilon$

$A \rightarrow a$

$B \rightarrow b$

$S \Rightarrow ASB \Rightarrow AASBB \Rightarrow aASBB \Rightarrow aaSBB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabb \quad aabb \in L(G2)$

$L(G2) = \{a^n b^n, n \in \mathbb{N}\}$

Given the grammar

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow num$

$E \rightarrow id$

1. Show that the word $1-2-3$ is in the language generated by this grammar
2. Draw two different parse trees for the given word. Conclude that this grammar is ambiguous
3. Do the same exercise with the word $1+2*3$

Classes of grammars

- The [classification of Noam Chomsky](#) (a linguist, In 1959) suggested classes of grammars according to complexity

0. [Unrestricted](#): An unrestricted grammar is one in which there are no restrictions on the rewriting rules

1. [Context-Sensitive](#): A context-sensitive grammar is one in which each rule must be of the form $\alpha A \gamma \rightarrow \alpha \beta \gamma$

where each α, β, γ is any string of terminals and nonterminal,
and A represents a nonterminal

2. [Context-Free](#): A context-free grammar is one in which each rule must be of the form: $A \rightarrow \alpha$

3. [Right Linear](#): A right linear grammar is one in which each rule is of the form:

$A \rightarrow aB \mid a$ where A and B represent nonterminals, and a represents a terminal

Notice

- Right linear grammars can be used to define lexical items such as identifiers, constants, and keywords
- $class(3) \subset class(2) \subset class(1) \subset class(0)$

Exercises

- Ecrire une grammaire non ambiguë qui engendre les langages suivants:
 - (a). Parenthèses et crochets équilibrés. Exemple: $([[](([])[()])])$
 - (b). les palindromes sur l'alphabet $\{a,b\}$
- *Montrer que l'image miroir d'un langage algébrique est algébrique.*
- Montrer que la famille des langages algébriques est close par les opérations rationnelles (union, concaténation et itération).
 - Remarque : Elle n'est par contre pas close par intersection, ni par passage au complémentaire.
- *Montrer que la grammaire $S \rightarrow SS + aSb + 1$ est ambiguë, et construire une grammaire non ambiguë qui reconnaît le même langage.*

Exercises

- Write an unambiguous grammar for each of the following languages.
 - a. Palindromes over the alphabet $\{a, b\}$ (strings that are the same backward and forward).
 - b. Strings that match the regular expression a^*b^* and have more a's than b's.
 - c. Balanced parentheses and square brackets. Example: $([[[]() [()] []])$
- Show that the grammar $S \rightarrow SS | aSb | \varepsilon$ is ambiguous. Then build a non ambiguous grammar which recognize the same language.

Exercises

- **Objective**: convert a non finite automaton (DFA) to a context-free grammar
- Give a right linear grammar for each of the languages of Sample Problem
 1. Strings containing an odd number of zeros
 2. Strings containing three consecutive ones
 3. Strings containing exactly three zeros
 4. Strings containing an odd number of zeros and an even number of ones.

Answer: General algorithm

- Algorithm to convert a DFA into a right linear grammar
 1. Associate a variable X_i to each state i
 2. For each transition $i \xrightarrow{a} j$, add a new production $X_i \rightarrow aX_j$ to the set of rules
 3. For each final state k , add $X_k \rightarrow \varepsilon$ to the set of production rules
 4. the non terminal associated to the initial state is the start symbol (axiom)

Parsing problem

- Given a grammar and an input string, determine whether the string is in the language of the grammar, and, if so, determine its structure.
- Classification of Parsing algorithms
 - refers to the sequence in which a derivation tree is built or traversed
 - Two approaches: either **top down** or **bottom up**
- **top down parsing algorithm**, grammar rules are applied in a sequence which corresponds to a general top down direction in the derivation tree
- **Bottom up parsing algorithm** proceeds from the bottom of the derivation tree and applies grammar rules (in reverse)

Top down parsing

Presentation

- End-Of-File Marker
 - Parsers must read not only terminal symbols but also the end-of-file marker.
 - The \$ symbol is usually used to represent end of file
 - The given grammar is augmented with a new start symbol S' and a new production $S' \rightarrow S\$$
- Compute the FIRST and FOLLOW sets for each non-terminal
 - FIRST (γ) is the set of terminals that can begin strings derived from γ
 - FOLLOW (X) is the set of terminals that can immediately follow X
- Build the parsing table

Iterative computation of FIRST

- if X is a terminal symbol, $FIRST(X) = \{X\}$
- If $X \rightarrow \epsilon$ is a production, $FIRST(X) \supseteq \{\epsilon\}$
- If $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$ is a production,
 - $FIRST(X) = FIRST(Y_1) \setminus \{\epsilon\}$ (privé du mot vide)
 - $\cup FIRST(Y_2)$ if $\epsilon \in FIRST(Y_1)$
 - $\cup FIRST(Y_3)$ if $\epsilon \in FIRST(Y_2)$
 -
 - $\cup FIRST(Y_n)$ if $\epsilon \in FIRST(Y_{n-1})$
 - $\cup \{\epsilon\}$ if $\epsilon \in FIRST(Y_n)$
- find the union of the First(x) sets for each symbol on the right side of a rule, but stop when reaching a non-nullable symbol

Computation of FOLLOW sets

- Add the EOF marker \$ in FOLLOW(S) where S is the axiom
- if $A \rightarrow \alpha B \beta$ is a production where $A, B \in V$ and $\alpha, \beta \in (V \cup \Sigma)^*$ then $FOLLOW(B) \supseteq FIRST(\beta) - \{\epsilon\}$
- if $A \rightarrow \alpha B$ is a production where $A, B \in V$ and $\alpha \in (V \cup \Sigma)^*$ then $FOLLOW(B) \supseteq FOLLOW(A)$
- if $A \rightarrow \alpha B \beta$ and $FIRST(\beta)$ contains $\{\epsilon\}$ then $FOLLOW(B) \supseteq FOLLOW(A)$

Constructing the parsing table

- The predictive parsing table is a two-dimensional table where
 - The rows are indexed by nonterminals
 - The columns are indexed by the terminals
 - And the cells contains production rules
- The parsing tables encode the information which are needed to implement a predictive parser
- Algorithm (**How to fill the table**)
 - Enter production $X \rightarrow \gamma$ in row X , column T of the table for each $T \in \text{FIRST}(\gamma)$
 - if γ is nullable, enter the production in row X , column T for each $T \in \text{FOLLOW}(X)$.
- A grammar is LL(1) if its predictive parsing table contain no duplicate entries
 - *LL(1) stands for left-to-right parse, leftmost-derivation, 1-symbol lookahead*

Example

$$S \rightarrow ABc$$
$$A \rightarrow bA$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow c$$

1. $\text{First}(ABc) = \text{First}(A) \cup \text{First}(B) = \{b, c\}$ (because A is nullable)
2. $\text{First}(bA) = \{b\}$
3. $\text{First}(\varepsilon) = \{\varepsilon\}$
4. $\text{First}(c) = \{c\}$

$$\text{First}(S) = \{b, c\}$$
$$\text{First}(A) = \{b, \varepsilon\}$$
$$\text{First}(B) = \{c\}$$
$$\text{First}(b) = \{b\}$$
$$\text{First}(c) = \{c\}$$

1. $\text{Follow}(S) = \{\$ \}$
2. $\text{Follow}(A) = \text{First}(B) = \{c\}$
3. $\text{Follow}(B) = \text{First}(c) = \{c\}$

	b	c	\$
S	$S \rightarrow Abc$	$S \rightarrow Abc$	
A	$A \rightarrow bA$	$A \rightarrow$	
B		$B \rightarrow c$	

$$L_G = \{b^n c^2, n \geq 1\}$$

Show that $u = bbbcc \in L_G$

word	stack	Actions
$bbbcc\$$	$\$S$	$S \rightarrow ABC$
$bbbcc\$$	$\$cBA$	$A \rightarrow bA$
$bbbcc\$$	$\$cBAb$	match
$bbcc\$$	$\$cBA$	$A \rightarrow bA$
$bbcc\$$	$\$cBAb$	match
$bcc\$$	$\$cBA$	$A \rightarrow Ab$
$bcc\$$	$\$cBAb$	match
$cc\$$	$\$cBA$	$A \rightarrow \epsilon$
$cc\$$	$\$cB$	$B \rightarrow c$
$c\$$	$\$c$	match
$\$$	$\$$	match
		accept.

Parse tree diagram:

```

graph TD
    S --> A
    S --> B
    S --> C
    A --> b
    A --> A1[A]
    A1 --> b
    A1 --> A2[A]
    A2 --> b
    A2 --> A3[A]
    A3 --> b
    A3 --> A4[A]
    A4 --> b
    A4 --> A5[A]
    A5 --> b
    A5 --> A6[A]
    A6 --> b
    A6 --> A7[A]
    A7 --> b
    A7 --> A8[A]
    A8 --> b
    A8 --> A9[A]
    A9 --> b
    A9 --> A10[A]
    A10 --> b
    A10 --> A11[A]
    A11 --> b
    A11 --> A12[A]
    A12 --> b
    A12 --> A13[A]
    A13 --> b
    A13 --> A14[A]
    A14 --> b
    A14 --> A15[A]
    A15 --> b
    A15 --> A16[A]
    A16 --> b
    A16 --> A17[A]
    A17 --> b
    A17 --> A18[A]
    A18 --> b
    A18 --> A19[A]
    A19 --> b
    A19 --> A20[A]
    A20 --> b
    A20 --> A21[A]
    A21 --> b
    A21 --> A22[A]
    A22 --> b
    A22 --> A23[A]
    A23 --> b
    A23 --> A24[A]
    A24 --> b
    A24 --> A25[A]
    A25 --> b
    A25 --> A26[A]
    A26 --> b
    A26 --> A27[A]
    A27 --> b
    A27 --> A28[A]
    A28 --> b
    A28 --> A29[A]
    A29 --> b
    A29 --> A30[A]
    A30 --> b
    A30 --> A31[A]
    A31 --> b
    A31 --> A32[A]
    A32 --> b
    A32 --> A33[A]
    A33 --> b
    A33 --> A34[A]
    A34 --> b
    A34 --> A35[A]
    A35 --> b
    A35 --> A36[A]
    A36 --> b
    A36 --> A37[A]
    A37 --> b
    A37 --> A38[A]
    A38 --> b
    A38 --> A39[A]
    A39 --> b
    A39 --> A40[A]
    A40 --> b
    A40 --> A41[A]
    A41 --> b
    A41 --> A42[A]
    A42 --> b
    A42 --> A43[A]
    A43 --> b
    A43 --> A44[A]
    A44 --> b
    A44 --> A45[A]
    A45 --> b
    A45 --> A46[A]
    A46 --> b
    A46 --> A47[A]
    A47 --> b
    A47 --> A48[A]
    A48 --> b
    A48 --> A49[A]
    A49 --> b
    A49 --> A50[A]
    A50 --> b
    A50 --> A51[A]
    A51 --> b
    A51 --> A52[A]
    A52 --> b
    A52 --> A53[A]
    A53 --> b
    A53 --> A54[A]
    A54 --> b
    A54 --> A55[A]
    A55 --> b
    A55 --> A56[A]
    A56 --> b
    A56 --> A57[A]
    A57 --> b
    A57 --> A58[A]
    A58 --> b
    A58 --> A59[A]
    A59 --> b
    A59 --> A60[A]
    A60 --> b
    A60 --> A61[A]
    A61 --> b
    A61 --> A62[A]
    A62 --> b
    A62 --> A63[A]
    A63 --> b
    A63 --> A64[A]
    A64 --> b
    A64 --> A65[A]
    A65 --> b
    A65 --> A66[A]
    A66 --> b
    A66 --> A67[A]
    A67 --> b
    A67 --> A68[A]
    A68 --> b
    A68 --> A69[A]
    A69 --> b
    A69 --> A70[A]
    A70 --> b
    A70 --> A71[A]
    A71 --> b
    A71 --> A72[A]
    A72 --> b
    A72 --> A73[A]
    A73 --> b
    A73 --> A74[A]
    A74 --> b
    A74 --> A75[A]
    A75 --> b
    A75 --> A76[A]
    A76 --> b
    A76 --> A77[A]
    A77 --> b
    A77 --> A78[A]
    A78 --> b
    A78 --> A79[A]
    A79 --> b
    A79 --> A80[A]
    A80 --> b
    A80 --> A81[A]
    A81 --> b
    A81 --> A82[A]
    A82 --> b
    A82 --> A83[A]
    A83 --> b
    A83 --> A84[A]
    A84 --> b
    A84 --> A85[A]
    A85 --> b
    A85 --> A86[A]
    A86 --> b
    A86 --> A87[A]
    A87 --> b
    A87 --> A88[A]
    A88 --> b
    A88 --> A89[A]
    A89 --> b
    A89 --> A90[A]
    A90 --> b
    A90 --> A91[A]
    A91 --> b
    A91 --> A92[A]
    A92 --> b
    A92 --> A93[A]
    A93 --> b
    A93 --> A94[A]
    A94 --> b
    A94 --> A95[A]
    A95 --> b
    A95 --> A96[A]
    A96 --> b
    A96 --> A97[A]
    A97 --> b
    A97 --> A98[A]
    A98 --> b
    A98 --> A99[A]
    A99 --> b
    A99 --> A100[A]
    A100 --> b
    A100 --> A101[A]
    A101 --> b
    A101 --> A102[A]
    A102 --> b
    A102 --> A103[A]
    A103 --> b
    A103 --> A104[A]
    A104 --> b
    A104 --> A105[A]
    A105 --> b
    A105 --> A106[A]
    A106 --> b
    A106 --> A107[A]
    A107 --> b
    A107 --> A108[A]
    A108 --> b
    A108 --> A109[A]
    A109 --> b
    A109 --> A110[A]
    A110 --> b
    A110 --> A111[A]
    A111 --> b
    A111 --> A112[A]
    A112 --> b
    A112 --> A113[A]
    A113 --> b
    A113 --> A114[A]
    A114 --> b
    A114 --> A115[A]
    A115 --> b
    A115 --> A116[A]
    A116 --> b
    A116 --> A117[A]
    A117 --> b
    A117 --> A118[A]
    A118 --> b
    A118 --> A119[A]
    A119 --> b
    A119 --> A120[A]
    A120 --> b
    A120 --> A121[A]
    A121 --> b
    A121 --> A122[A]
    A122 --> b
    A122 --> A123[A]
    A123 --> b
    A123 --> A124[A]
    A124 --> b
    A124 --> A125[A]
    A125 --> b
    A125 --> A126[A]
    A126 --> b
    A126 --> A127[A]
    A127 --> b
    A127 --> A128[A]
    A128 --> b
    A128 --> A129[A]
    A129 --> b
    A129 --> A130[A]
    A130 --> b
    A130 --> A131[A]
    A131 --> b
    A131 --> A132[A]
    A132 --> b
    A132 --> A133[A]
    A133 --> b
    A133 --> A134[A]
    A134 --> b
    A134 --> A135[A]
    A135 --> b
    A135 --> A136[A]
    A136 --> b
    A136 --> A137[A]
    A137 --> b
    A137 --> A138[A]
    A138 --> b
    A138 --> A139[A]
    A139 --> b
    A139 --> A140[A]
    A140 --> b
    A140 --> A141[A]
    A141 --> b
    A
```

Exercise: build the LL(1) parse table

$$\begin{cases} S \rightarrow Ab|a|AA \\ A \rightarrow Sa|Ac|B \\ B \rightarrow Sd \end{cases}$$

Left Recursion

- Consider the two productions $E \rightarrow E + T$ and $E \rightarrow T$
 - The 1st rule is in the form: $A \rightarrow Aa$
 - This property is known as **left recursion**
- **Theorem:** Grammars with left recursion cannot be LL(1)
- **Eliminate left recursion**
 - The offending rule might be in the form: $\begin{cases} A \rightarrow A\alpha \\ A \rightarrow \beta \end{cases}$
 - in which we assume that β is a string of terminals and nonterminals that does not begin with an A
 - Eliminate the left recursion by **introducing a new nonterminal, R** , and **rewriting the rules** as: $\begin{cases} A \rightarrow \beta R \\ R \rightarrow \alpha R | \epsilon \end{cases}$
- **Theorem:** The resulting rules derive the same strings as the original productions

Left recursion

- Definition: A grammar is recursive on the left if it contains a non-terminal A such that there is a derivation $A \rightarrow^+ \alpha$ where α is any string
- Example $\begin{cases} S \rightarrow Aa|b \\ A \rightarrow Ac|Sd|\varepsilon \end{cases}$
- The non terminal S is left recursive since $S \rightarrow Aa \rightarrow Sda$
- Elimination of left recursion

Order the non terminals A_1, A_2, \dots, A_n

For $i=1$ to n

 for $j=1$ to $i-1$

 replace each production $A_i \rightarrow A_j \alpha$ where $A_j \rightarrow \beta_1 | \dots | \beta_p$ by $A_i \rightarrow \beta_1 \alpha | \dots | \beta_p \alpha$

 end for

 eliminate the immediate left recursion on the productions A_i

End for

Exemple

- Order S, A
- $i=1$ no immediate left recursion in $S \rightarrow Aa|b$
- $i=2$ et $j=1$ replace S by respectively Aa and d in $A \rightarrow Sd$, we obtain $A \rightarrow Ac|Aad|bd|\varepsilon$
- Eliminate the immediate left recursion $\begin{cases} A \rightarrow bdA'|A' \\ A' \rightarrow cA'|adA'|\varepsilon \end{cases}$
- The resulting grammar is $\begin{cases} S \rightarrow Aa|b \\ A \rightarrow bdA'|A' \\ A' \rightarrow cA'|adA'|\varepsilon \end{cases}$

Left factoring

- A grammar is not **left factor** when the same nonterminal start with the same symbols
- **Theorem:** a grammar which is not left factor can not be LL(1)
- **Left factor a grammar**
 - Take the allowable endings and make a new nonterminal to stand for them
 - Replace the production $A \rightarrow \alpha\beta_1 | \dots | \alpha\beta_n | \gamma_1 | \dots | \gamma_p$ where $\alpha \neq \epsilon$ and γ_i does not start with α , with the two rules:
 - $A \rightarrow \alpha A' | \gamma_1 | \dots | \gamma_p$
 - $A' \rightarrow \beta_1 | \dots | \beta_n$
- **Example**

$$\left\{ \begin{array}{l} S \rightarrow aEbS | aEbSeB | a \\ E \rightarrow bcB | bca \\ B \rightarrow ba \end{array} \right.$$

$$\left\{ \begin{array}{l} S \rightarrow aEbSS' | a \\ S' \rightarrow eB | \epsilon \\ E \rightarrow bcE' \\ E' \rightarrow B | a \\ B \rightarrow ba \end{array} \right.$$

Exercises

- Show how to eliminate the left recursion from each of the grammars shown below:
 - $A \rightarrow Abc|ab$
 - $ParmList \rightarrow ParmList, Parm|Parm$

Bottom up parsing

Motivation and presentation

- The weakness of LL(k) parsing techniques
 - Prediction which production to use, having seen only the first k tokens of the right-hand side
 - The situations in which it is not easy to use an LL(k) grammar
 - Recursive grammars
 - Not left factor grammar
- Bottom up techniques LR(k) parsing are more powerful
 - Postpone the decision until it has seen input tokens corresponding to the entire right-hand side of a production
 - Grammar rules are applied in reverse
 - Derivation trees are constructed, or traversed, from bottom to top
- LR(k) stands for *left-to-right* parse, *rightmost-derivation*, k-token lookahead.
 - L indicates we are reading input from the left
 - R indicates we are finding a right-most derivation

LR parsing engine

- The LR parser **performs two kinds of actions**
 - **Shift**: Move the input token to the top of the stack.
 - **Reduce**: Choose a grammar rule $X \rightarrow \alpha_1 \alpha_2 \dots \alpha_k$;
 - pop $\alpha_k, \dots \alpha_1$ from the top of the stack
 - push X onto the stack.
- LR parser uses a DFA (**deterministic finite automaton**)
 - The DFA is applied to the stack
 - The edges of the DFA are labeled by the symbols (terminals and non-terminals)
 - Four kinds of actions label the transition table
 - sn Shift into state n;
 - gn Goto state n;
 - rk Reduce by rule k;
 - a Accept;

Constructing a DFA

- Augment the grammar with $S' \rightarrow S\$$
 - S' is a new variable with derive the starting symbol
 - *Explanation: the input will be a complete S-sentence followed by \$*
 - $\$$ is the EOF marker
- An **item** is a grammar rule, combined with the dot that indicates a position in its right-hand side
 - Example: $S' \rightarrow .S\$$
- A **state** is a set of items
- The basic operations we have been performing on states are considering that I is a set of Items and X is a grammar symbol (terminal or variable)
 - $Closure(I)$ adds more items to a set of items when there is a dot to the left of a nonterminal
 - $goto(I, X)$ moves the dot past the symbol X in all items

Computation of the closure

Closure (I) =

repeat

for *any item* $A \rightarrow \alpha.X\beta$ *in* I

for *any production* $X \rightarrow \gamma$

$I \leftarrow I \cup \{ X \rightarrow .\gamma \}$

until I *does not change*

return I

$S' \rightarrow S\$$

$S \rightarrow (L)$

$S \rightarrow x$

$L \rightarrow S$

$L \rightarrow L, S$

Examples of closure

$S \rightarrow (.L)$

$L \rightarrow .S$

$L \rightarrow .L, S$

$S \rightarrow .x$

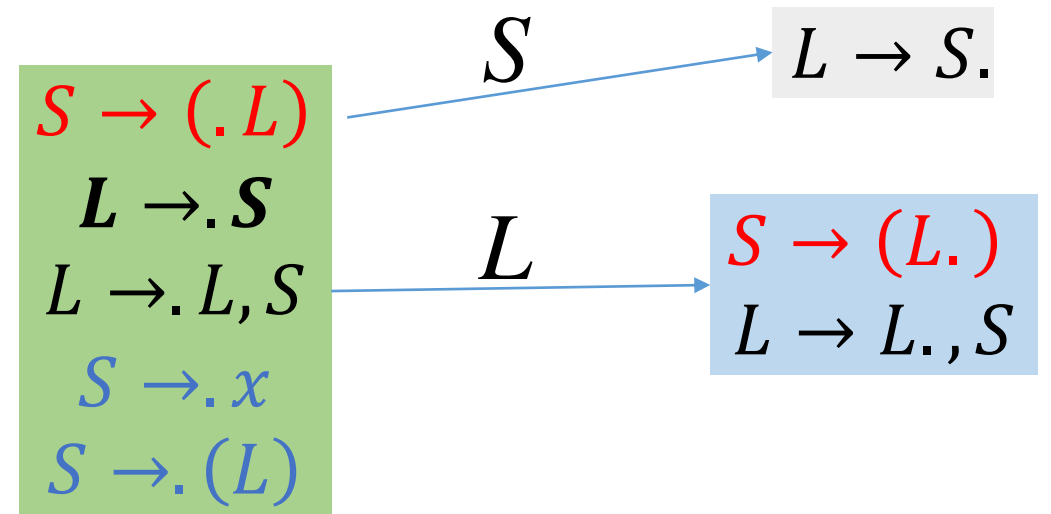
$S \rightarrow .(L)$

$S \rightarrow .x$

Performing Goto

Goto (I, X) =
set J to the empty set
for any item $A \rightarrow \alpha.X\beta$ in I
 add $A \rightarrow \alpha X.\beta$ to J
return $Closure(J)$

Examples of goto



DFA of items construction

1. Augment the grammar with an auxiliary start production $S' \rightarrow S\$$
2. Let T be the set of states seen so far,
3. Let E be the set of (shift or goto) edges found so far

Initialize E to empty

Initialize T to $\{ \textbf{Closure} (\{ S' \rightarrow .S\$ \}) \}$

repeat

***for** each state I in T*

***for** each item $A \rightarrow \alpha.X\beta$ in I*

***let** J be **Goto** (I, X)*

$T \leftarrow T \cup \{J\}$

$E \leftarrow E \cup \{ I \xrightarrow{X} J \}$

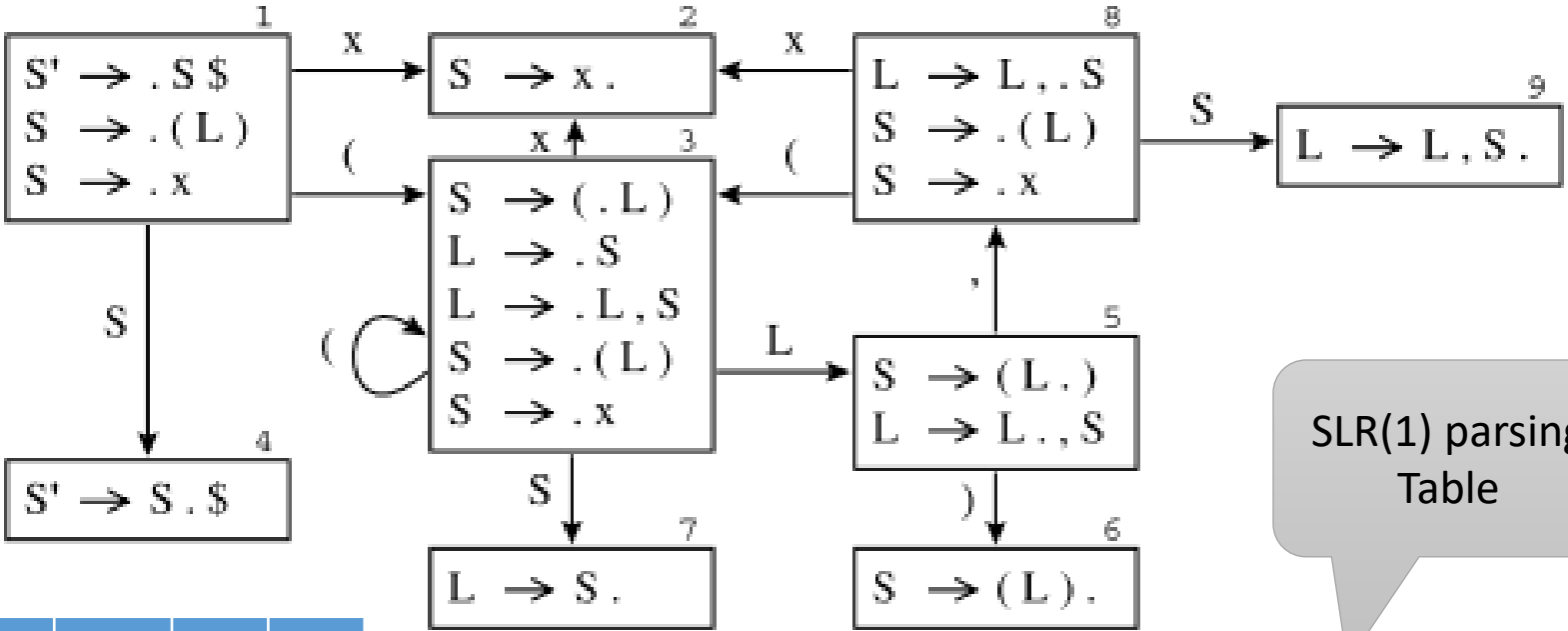
***until** E and T did not change in this iteration*

Constructing the parsing table

- The LR parsing table is a two-dimensional table where
 - The rows are indexed by the states of the DFA
 - The columns are indexed by the terminals and nonterminals
 - And the cells contain the action (shift, reduce, accept, goto)
- For each edge $I \xrightarrow{X} J$
 - If X is a terminal, put the action *shift J* at position (I,X) of the table
 - if X is a nonterminal, we put *goto J* at position (I,X)
- For each state I containing an item $S' \rightarrow S. \$$
 - Put an *accept* action at (I, \$)
- For a state containing an item $A \rightarrow \gamma.$ (production **n** with the dot at the end)
 - **Case of LR(0):** put a reduce **n** action at (I, Y) for every token Y
 - **Case of SLR(1):** put a reduce **n** action at (I, Y) for each token Y in **FOLLOW(A)**

Example

- 0. $S' \rightarrow S\$$
- 1. $S \rightarrow (L)$
- 2. $S \rightarrow x$
- 3. $L \rightarrow S$
- 4. $L \rightarrow L, S$



LR(0) parsing Table

	()	x	,	\$	S	L
1	S3		S2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					Acc		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Follow(S)={ $\$$) , }
Follow(L)={) , }

SLR(1) parsing Table

	()	x	,	\$	S	L
1	S3		S2			g4	
2		r2		r2	r2		
3	S3		S2			g7	g5
4					Acc		
5		s6		s8			
6		r1		r1	r1		
7		r3		r3			
8	S3		S2			g9	
9		r4		r4			

Parsing a word

We consider the word $u = (x, x)$

input	Stack	Action
$(x, x) \$$	$\$ \boxed{1}$	S3
$x, x) \$$	$\$ \boxed{1} (\boxed{3}$	S2
$, x) \$$	$\$ \boxed{1} (\boxed{3} x \boxed{2}$	r2: $S \rightarrow x$
$) x) \$$	$\$ \boxed{1} (\boxed{3} S$	g7
$, x) \$$	$\$ \boxed{1} (\boxed{3} S \boxed{7}$	r3: $L \rightarrow S$
$) x) \$$	$\$ \boxed{1} (\boxed{3} L$	g5
$, x) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}$	S8
$x) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}, \boxed{8}$	S2
$) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}, \boxed{8} x \boxed{2}$	r2: $S \rightarrow x$
$) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}, \boxed{8} S$	g9
$) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}, \boxed{8} S \boxed{9}$	r4: $L \rightarrow L, S$
$) \$$	$\$ \boxed{1} (\boxed{3} L$	g5
$) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}$	S6
$) \$$	$\$ \boxed{1} (\boxed{3} L \boxed{5}) \boxed{6}$	r1: $S \rightarrow (L)$
$\$$	$\$ \boxed{1} S$	g4
$\$$	$\$ \boxed{1} S \boxed{4}$	Acc.

Limitations of Context-Free grammars

- Cannot represent semantics
 - e.g. “every variable used in a statement should be declared earlier in the code”; or “the use of a variable should conform to its type” (type checking)
 - Need to allow only programs that satisfy certain **context-sensitive conditions**
- Cannot be used to generate things other than parse trees
 - e.g., what if we wanted to generate assembly code for the given program?

Attributed Grammars

Semantics of context-free languages

Adding context to context-free grammar

Overview

- Attribute grammars were first developed by Donald Knuth in 1968
- A means of formalizing the semantics of a context-free language.
 - Their primary application has been in compiler writing, they are a tool mostly used by programming language implementers
- Attribute grammars can perform several useful functions in specifying the syntax and semantics of a programming language.
- An attribute grammar can be used to specify the context-sensitive aspects of the syntax of a language, such as checking that an item has been declared and that the use of the item is consistent with its declaration

Semantic actions

- A compiler must do more than recognize whether a sentence belongs to the language of a grammar
 - it must do something useful with that sentence.
 - The **semantic actions** of a parser can do useful things with the phrases that are parsed.
 - The semantic actions are fragments of program code, written in a programming language (Java, C, ...), attached to grammar productions
- Grammars are extended further by introducing **attributed grammars**
 - Each of the terminals and non-terminals may have zero or more attributes
 - zero or more attributes **computation rules are associated** with each grammar rule
- Examples
 - the attribute of an input symbol (a lexical token) could be the value part of the token
 - Manage the type of each identifier

Syntax Directed Definition

- A syntax-directed definition (SDD) connects a set of semantic rules to production
- Terminals and non-terminals have attributes
- Formally, a $SDD = (G, Attr, R)$ where
 - G is a context-free grammar
 - $Attr$ is a set of attributes
 - R is a set of semantic rules
- $X.a$ denotes the attribute **a** attached to the symbol X (terminal or non-terminal)
- If X appears many times in a production
 - $X^0.a$ is the LHS
 - X^1, \dots, X^n are the RHS

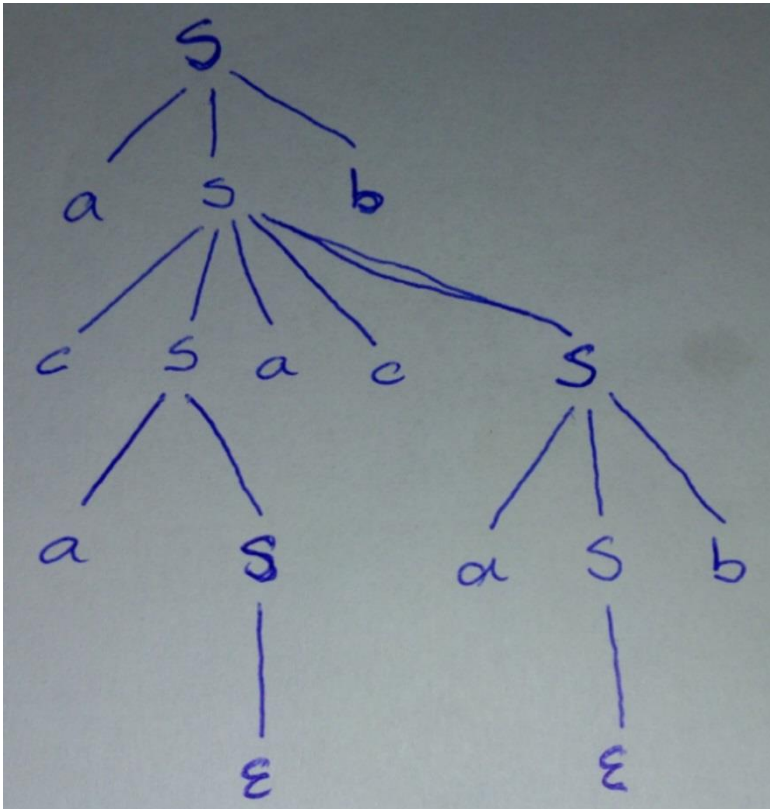
Example

- Consider the grammar $G: S \rightarrow aSb | aS | cSacS | \epsilon$
- Write the rules to calculate the number of **a** and **c** in a word $u \in L_G$
- Solution
 - Define two attributes
 - nba** which holds the number of a
 - nbc** which holds the number of c

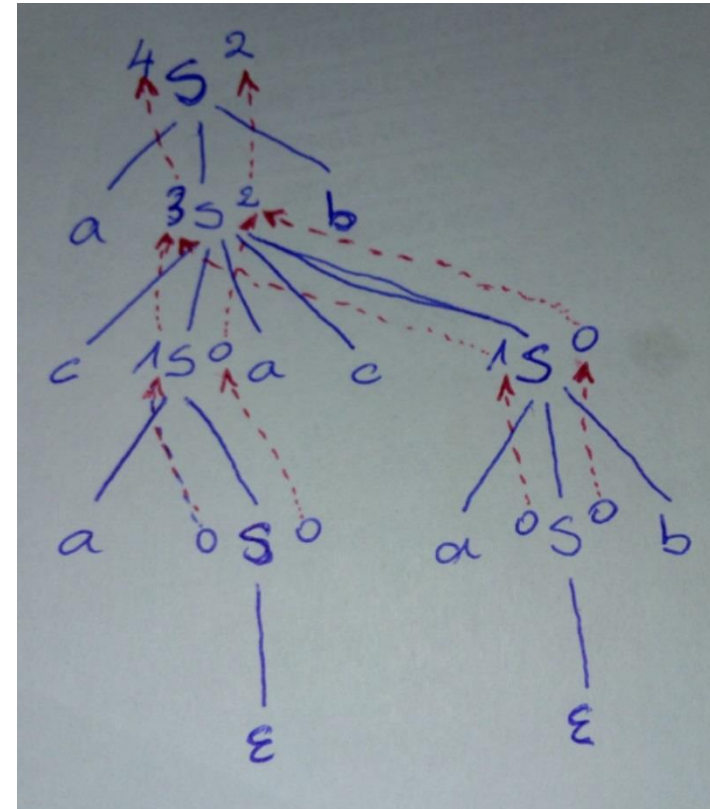
Productions	Semantic rules
$S \rightarrow aSb$	$S^0.nba = S^1.nba + 1$ $S^0.nbc = S^1.nbc$
$S \rightarrow aS$	$S^0.nba = S^1.nba + 1$ $S^0.nbc = S^1.nbc$
$S \rightarrow cSacS$	$S^0.nba = S^1.nba + S^2.nba + 1$ $S^0.nbc = S^1.nbc + S^2.nbc + 2$
$S \rightarrow \epsilon$	$S.nba = 0$ $S.nbc = 0$

Build decorated tree

Derivation tree



Anotated (Attributed) derivation tree



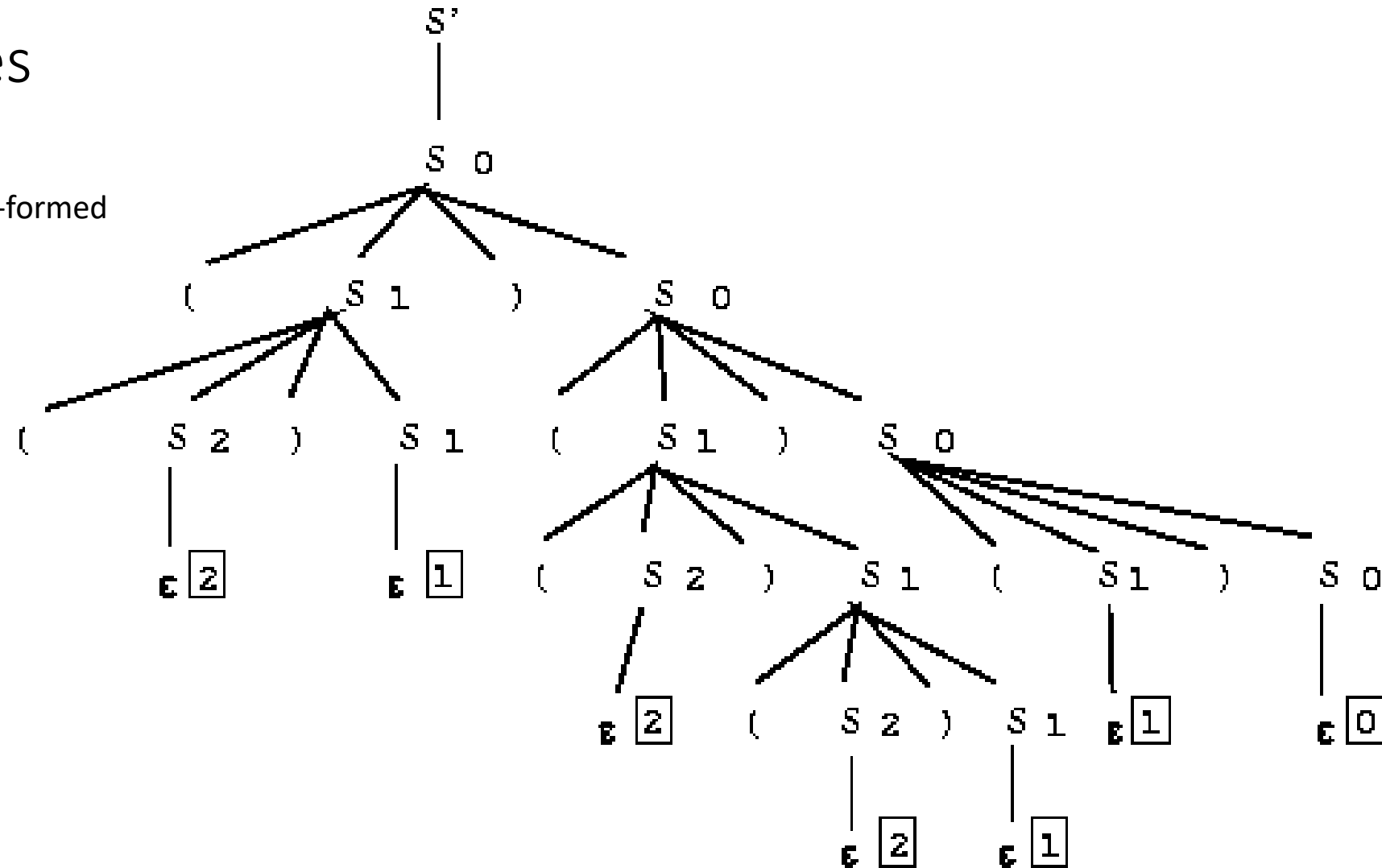
Categories of attributes

- Let $X_0 \rightarrow X_1 \dots X_n$ be a production
- If the computing rule of X_0 's attribute is of the form $A(X_0) = f(A(X_1), \dots, A(X_n))$
 - Synthesized attribute
- If the computing rule of X_j 's attribute is of the form $A(X_j) = f(A(X_0), \dots, A(X_i), \dots)$
 - Inherited attribute
- Terminals have intrinsic attributes
 - Lexical values supplied by the lexical analyzer

Inherited attributes example

calculating nesting level of) in a well-formed parenthesis system

$$\begin{cases} S' \rightarrow S \\ S \rightarrow (S)S | \epsilon \end{cases}$$



Productions	Semantic rules
$S' \rightarrow S$	$S.nb = 0$
$S \rightarrow (S)S$	$S^1.nb = S^0.nb + 1$ $S^2.nb = S^0.nb$
$S \rightarrow \epsilon$	<i>ecrire S.nb</i>

Exercise

- Design an attributed grammar to evaluate the arithmetic expressions

Productions	Semantic rules
$E \rightarrow E + T$	$E^0.val = E^1.val + T.val$
$E \rightarrow E - T$	$E^0.val = E^1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T^0.val = T^1.val * F.val$
$T \rightarrow T / F$	$T^0.val = T^1.val / F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow number$	$F.val = number.val$
$F \rightarrow (E)$	$F.val = E.val$

1. Implement your solution in SableCC

2. Additionnal requirements

Add the identifier (ID) token to your grammar so your calculator could accept expression with contains variable like: **3+x-y/(6*z)**
When the evaluator encounter an ID, it asks to the user to enter the value.

Synthesized vs inherited attributes

- synthesized attributes
 - The attributes take their values from attributes of lower nodes in the tree
 - This kind of attributes can be filled in a **bottom-up parsing**
- Inherited attributes
 - The attributes take their values from attributes of higher nodes in the tree and the siblings
 - The **top-down parsing** is indicated to fill these attributes if and only if the values come from the higher nodes
- In case a grammar holds synthesized and inherited attributes the process of filling in attribute values is not straightforward
 - Dependency graphs

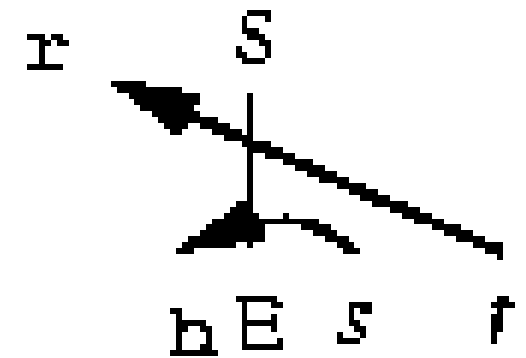
Dependency Graphs

- **Semantic rules** set up dependencies between attributes which can be represented by a *dependency graph*
- This dependency graph determines how attributes can be evaluated in parse trees
- For each symbol X, the dependency graph has a node for each attribute associated with X
- An edge from node A to node B means that the attribute of A is needed to compute the attribute of B
 - How to differentiate synthesized attributes from inherited attributes

Example

production	action sémantique
$S \rightarrow E$	$E.h := 2 * E.s + 1$
	$S.r := E.t$
$E \rightarrow EE$	$E(0).s := E(1).s + E(2).s$
	$E(0).t := 3 * E(1).t - E(2).t$
	$E(1).h := E(0).h$
	$E(2).h := E(0).h + 4$
$E \rightarrow \text{nombre}$	$E.s := \text{nombre}$
	$E.t := E.h + 1$

Dependency graph for productions



Dependancy graph for a syntax tree

