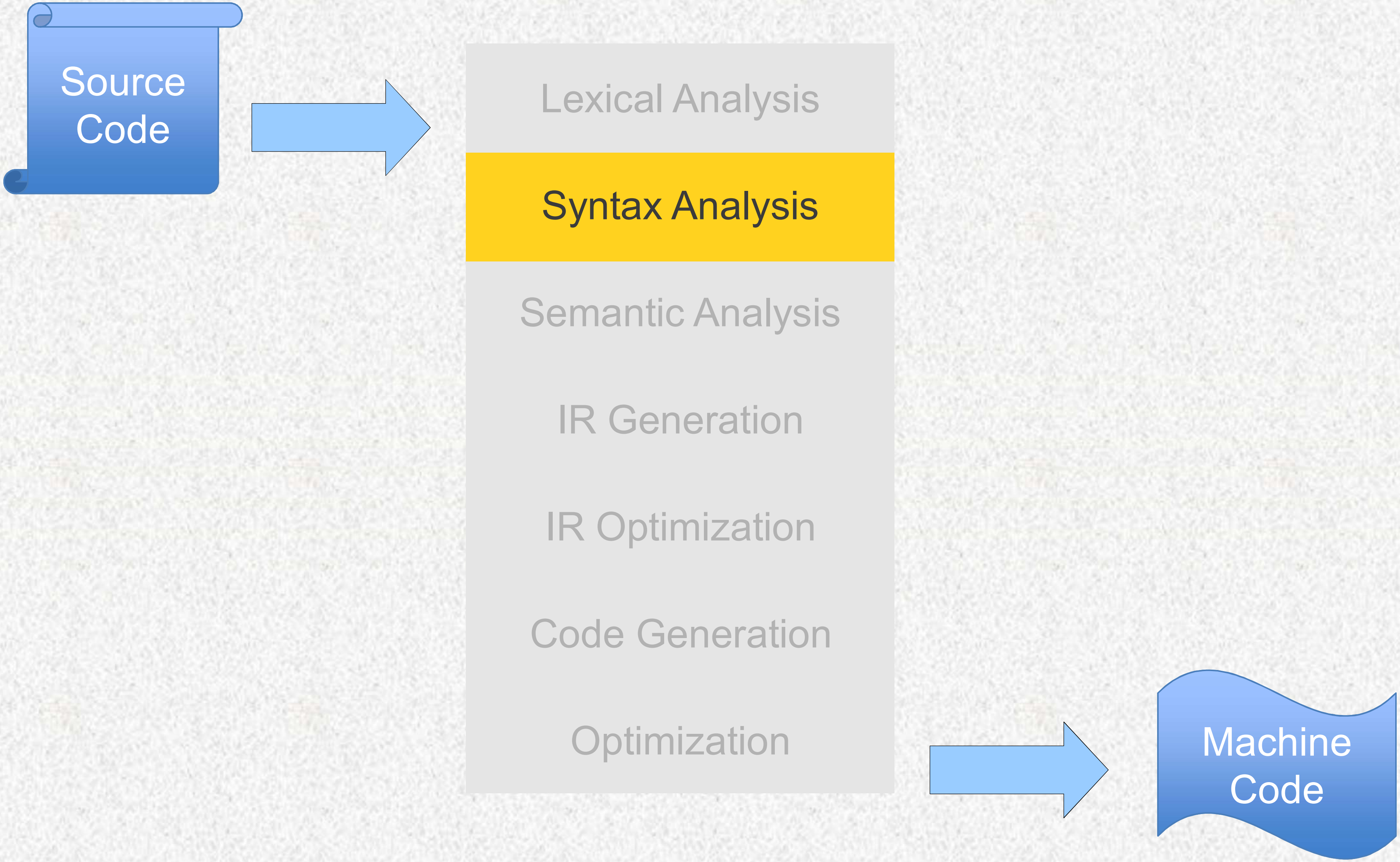
Lecture 5: Syntax Analysis

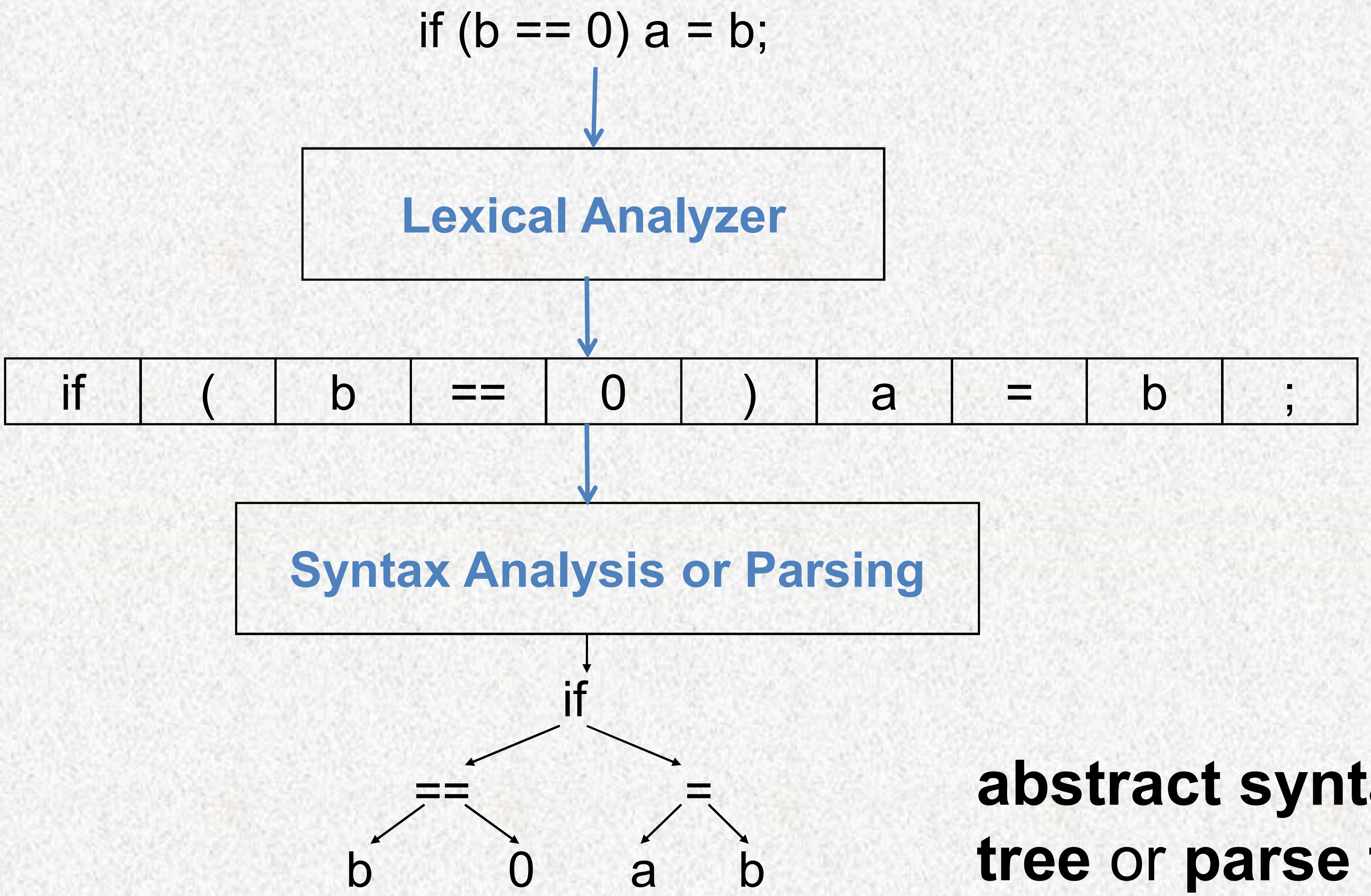
Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Syntax Analysis

Where are we ?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

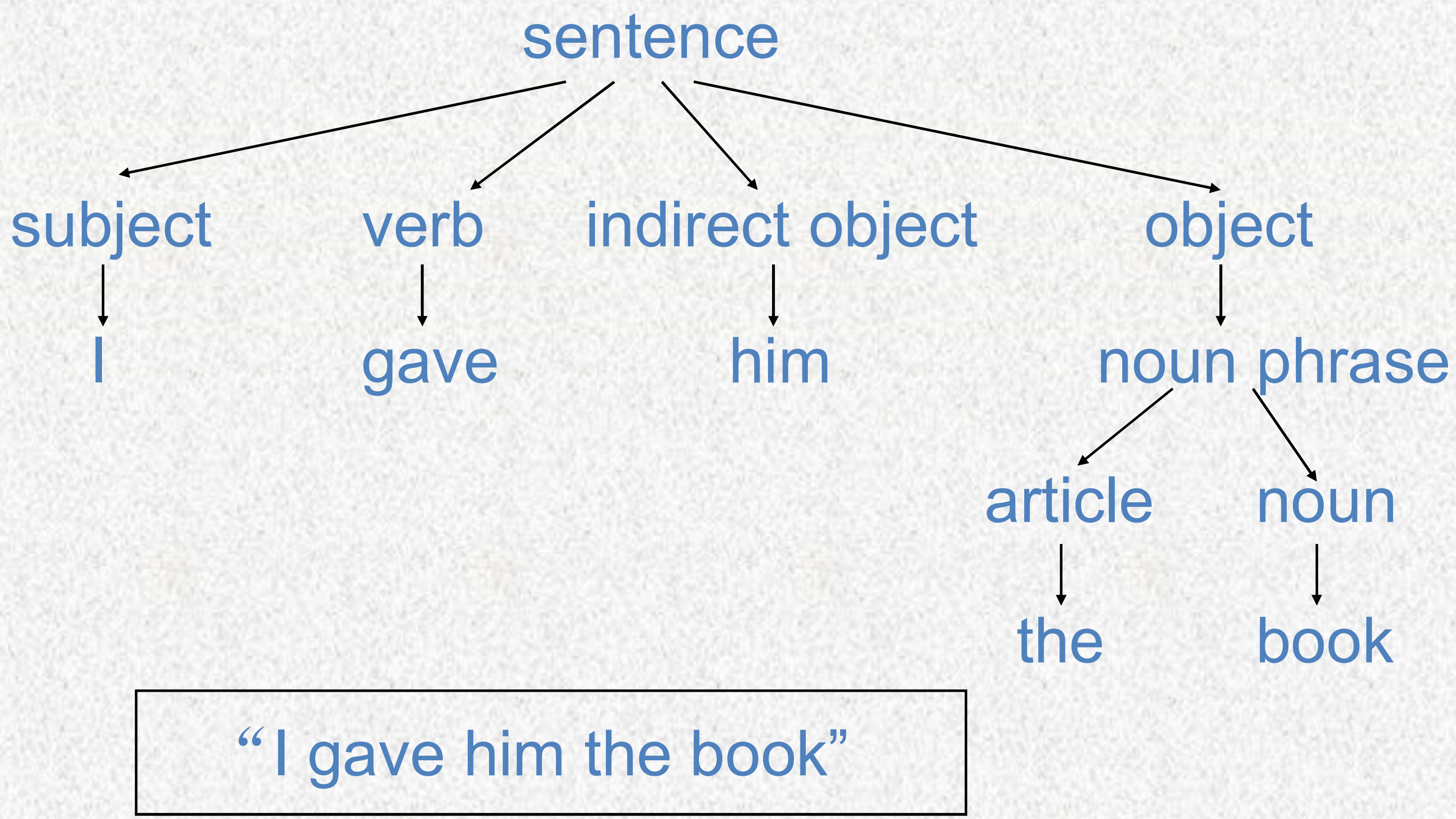
Where is Syntax Analysis Performed?



abstract syntax tree or **parse tree**

Parsing Analogy

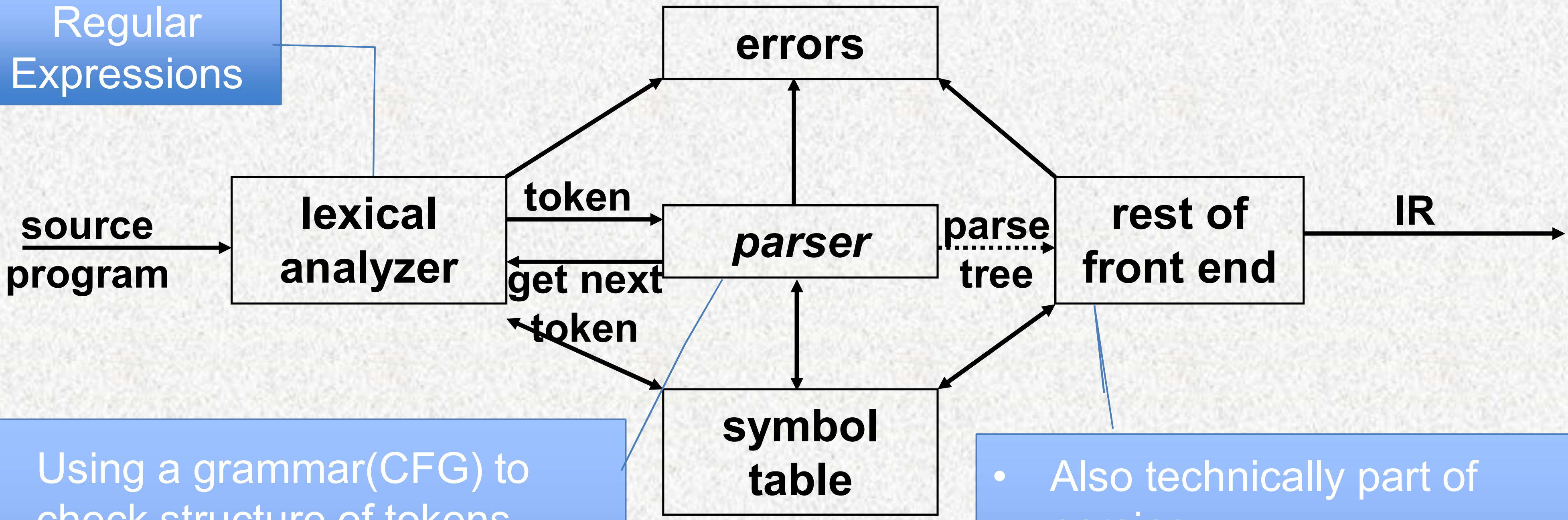
- Syntax analysis for natural languages
  - Recognize whether a sentence is grammatically correct
  - Identify the function of each word



Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.

Regular Expressions



- Using a grammar(CFG) to check structure of tokens
- Produce a parse tree
- Recognize correct syntax
- Syntactic errors and recovery
- Reprot errors

- Also technically part of parsing
- Includes augmenting info on tokens in source, type checking, semantic analysis

Error Processing

- Detecting errors
- Finding position at which they occur
- Clear / accurate presentation
- **Recover (pass over) to continue and find later errors**

Syntax Analysis Overview

- Goal – Determine if the input token stream **satisfies syntax** of the program
- What do we need to do this?
  - An expressive way to describe the syntax
  - A mechanism that determines if the input token stream satisfies the syntax description

Syntax Analysis Overview

For lexical analysis

- Regular expressions describe tokens
- Finite automata = mechanisms to generate tokens from input stream

For syntax analysis

- Concrete and Abstract Syntax Trees: formalisms for syntax analysis
- PushDown Automaton (PDA): top-down parsing, bottom-up parsing

Language Recognition Problem

- Let a *language* L be any set of some arbitrary objects s which will be dubbed "sentences."
  - "legal" or "grammatically correct" sentences of the language.
- Let the *language recognition problem* for L be:
  - Given a sentence s, is it a legal sentence of the language L?
    - That is, is $s \in L$?

Intro to Languages

- English grammar tells us if a given combination of words is a valid sentence.

The syntax of a sentence concerns its form while the semantics concerns its meaning.
e.g. the mouse wrote a poem

From a syntax point of view this is a valid sentence.

From a semantics point of view not so…perhaps in Disneyland

Natural languages (English, French, Portguese, etc) have very complex rules of syntax and not necessarily well-defined.

Formal Language

- An alphabet is a set Σ of symbols that act as letters.
- A language over Σ is a set of strings made from symbols in Σ.
- Formal language – is specified by well-defined set of rules of syntax
- We describe the sentences of a formal language using a grammar.

Grammars

- A formal *grammar G* is any compact, precise mathematical definition of a language *L*.
  - As opposed to just a raw listing of all of the language's legal sentences, or just examples of them.
- A grammar implies an algorithm that would generate all legal sentences of the language.
  - Often, it takes the form of a set of recursive definitions.
- A popular way to specify a grammar recursively is to specify it as a *phrase-structure grammar.*

Grammars (Semi-formal)

- Example: A grammar that generates a subset of the English language

$$\langle sentence \rangle \to \langle noun\_phrase \rangle \ \langle predicate \rangle$$

$$\langle noun\_phrase \rangle \to \langle article \rangle \ \langle noun \rangle$$

$$\langle predicate \rangle \to \langle verb \rangle$$

$$\langle article \rangle \to a$$

$$\langle article \rangle \to the$$

$$\langle noun \rangle \to boy$$

$$\langle noun \rangle \to dog$$

$$\langle verb \rangle \to runs$$

$$\langle verb \rangle \to sleeps$$

- A derivation of "the boy sleeps":

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ sleeps
\end{aligned}
$$

- A derivation of "a dog runs":

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ dog \ \langle verb \rangle \\
&\Rightarrow a \ dog \ runs
\end{aligned}
$$

- Language of the grammar:

L = { "a boy runs",
"a boy sleeps",
"the boy runs",
"the boy sleeps",
"a dog runs",
"a dog sleeps",
"the dog runs",
"the dog sleeps" }

Notation

- $$\langle noun \rangle \to boy$$
  $$\langle noun \rangle \to dog$$

Variable
or
Non-terminal

Symbols of
the vocabulary

Production
rule

Terminal
Symbols of
the vocabulary

Phrase-Structure Grammars

- A *phrase-structure grammar* (abbr. PSG) $G = (V,T,S,P)$ is a 4-tuple, in which:
  - V is a vocabulary (set of symbols)
    - The “template vocabulary” of the language.
  - $T \subseteq V$ is a set of symbols called *terminals*
    - Actual symbols of the language.
  - $N :\equiv V - T$ is a set of special “symbols” called *nonterminals*. (Representing concepts like “noun”)
  - $S \in N$ is a special nonterminal, the *start symbol.*
    - in our example the start symbol was “sentence”.
  - P is a set of *productions* (to be defined).
    - Rules for substituting one sentence fragment for another
    - Every production rule must contain at least one nonterminal on its left side.

Phrase-structure Grammar

► **EXAMPLE:**

- Let $G = (V, T, S, P)$,
- where $V = \{a, b, A, B, S\}$
- $T = \{a, b\}$,
- S is a start symbol
- $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, A \rightarrow Bb\}$.

What sentences can be generated with this grammar?

Derivation

- Let G=(V,T,S,P) be a phrase-structure grammar.
- Let $w_0=lz_0r$ (the concatenation of l, z_0, and r) $w_1=lz_1r$ be strings over V.
- If $z_0 \rightarrow z_1$ is a production of G we say that w1 is directly derivable from w0 and we write $w_o \Rightarrow w_1$.
- If $w_0, w_1, \ldots, w_n$ are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is derivable from w_0, and write $w_0 \Rightarrow^* w_n$.
- The sequence of steps used to obtain w_n from w_o is called a derivation.

Language

- Let G(V,T,S,P) be a phrase-structure grammar. The
- language generated by G (or the language of G)
- denoted by L(G) , is the set of all strings of terminals
- that are derivable from the starting state S.

- $L(G)= \{w \in T^* \mid S \Rightarrow^* w\}$

Language L(G)

► **EXAMPLE:**

- Let $G = (V, T, S, P)$, where $V = \{a, b, A, S\}$, $T = \{a, b\}$, S is a start symbol and $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$.

- The language of this grammar is given by L (G) = $\{b, aaa\}$;

1. we can derive aA from using $S \rightarrow aA$, and then derive aaa using $A \rightarrow aa$.

2. We can also derive b using $S \rightarrow b$.

- Language of the grammar with the productions:

$$S \to aSb,\ S \to \varepsilon$$

$$L = \{a^n b^n : n \geq 0\}$$

Types of Grammars - Chomsky hierarchy of languages

- Type 2: Context-Free PSG:
  - All before fragments have length 1 and are nonterminals: P： $A\rightarrow\beta$， where $A\in N$， $\beta\in V^*$。

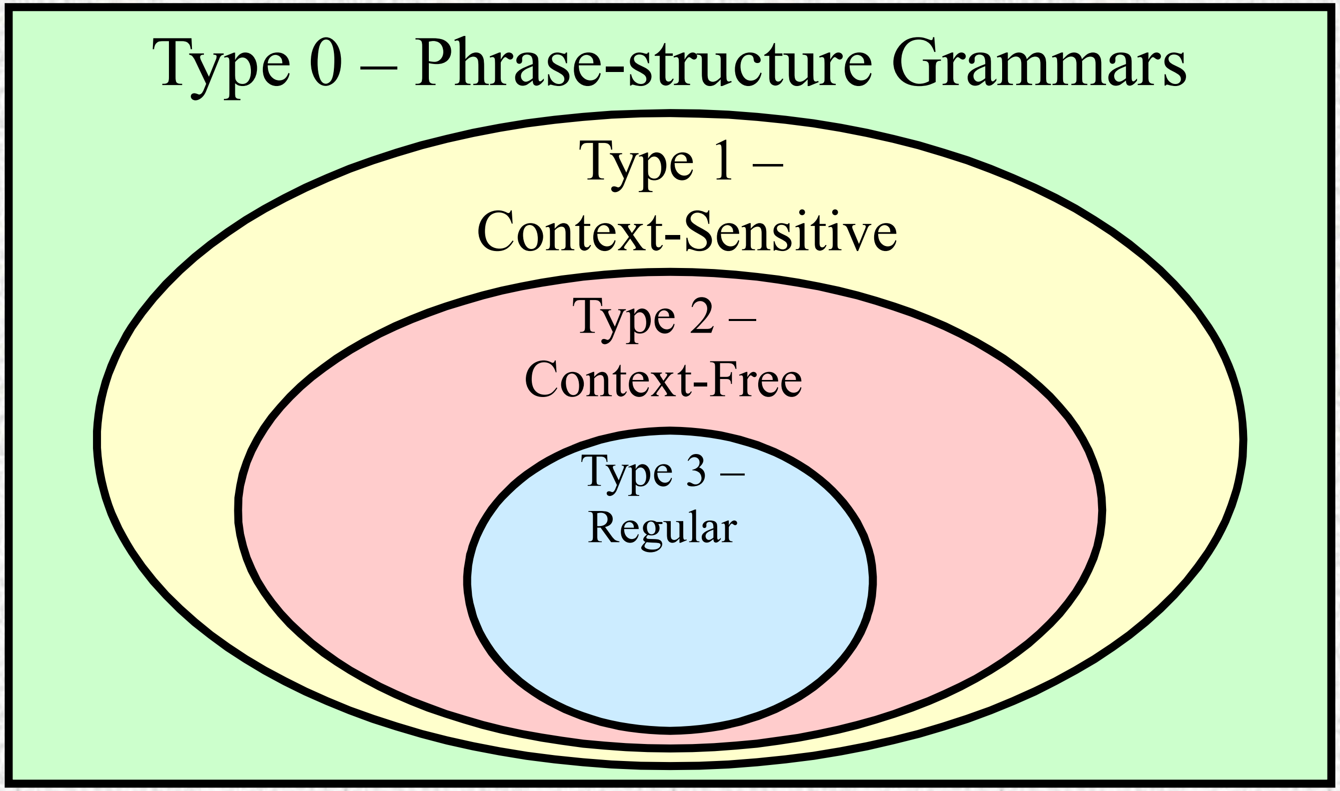
- Type 3: Regular PSGs:
  - All before fragments have length 1 and nonterminals
  - All after fragments are either single terminals, or a pair of a terminal followed by a nonterminal.

either $A\rightarrow \alpha B$， $A\rightarrow \alpha$ or, $A\rightarrow B\alpha$， $A\rightarrow \alpha$

where A， $B\in N$， $\alpha\in T^*$。

Types of Grammars - Chomsky hierarchy of languages

- Venn Diagram of Grammar Types:



The Limits of Regular Languages

- When scanning, we used **regular expressions** to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure (blocks, expressions, statements)

We need a more powerful formalism.

Context Free Grammars

- A context-free grammar (or CFG) is a formalism for defining languages.
- Can define the context-free languages, a strict superset of the the regular languages.

Context-Free Grammars

- Inherently **recursive** structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - **A finite set of terminals (in our case, this will be the set of tokens)**
  - **A finite set of non-terminals (syntactic-variables)**
  - **A finite set of productions rules in the following form**
    - **$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)**
  - **A start symbol (one of the non-terminal symbol)**

Example Grammar

$$
\begin{aligned}
expr &\rightarrow expr\ \ op\ \ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow id \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow /
\end{aligned}
$$

Black : Nonterminal

Blue : Terminal

expr **:** ***Start Symbol***

8 Production rules

Terminology

- L(G) is *the* *language* of G (the language generated by G) which is a set of sentences.
- A ***sentence*** of L(G) is a string of terminal symbols of G.
- If S is the start symbol of G then

  ω is a sentence of L(G) if $S \overset{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.
- A language that can be generated by a grammar is said to be a **context-free language.**
- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow^* \alpha$
  - If α contains non-terminals, it is called as a ***sentential form*** of G.
  - If α does not contain non-terminals, it is called as a ***sentence*** of G.

Terminology

EX. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

id * id **is a sentence**

Here's the derivation:

$exp \Rightarrow exp \text{ op } exp \Rightarrow exp * exp \Rightarrow \text{id} * exp \Rightarrow \text{id} * \text{id}$

Sentential forms

Sentence

$exp \Rightarrow^* id * id$

Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e. A, B, C, D
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e. **`t`**, **`u`**, **`v`**, **`w`**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e. α, γ, ω

Examples

- We might write an arbitrary production as

$$A \rightarrow \omega$$

- We might write a string of a nonterminal followed by a terminal as

$$A\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$B \rightarrow \alpha A \mathtt{t} \omega$$

Derivations

- The central idea here is that a production is treated as a **rewriting rule** in which the non-terminal on the left is replaced by the string on the right side of the production.
- $E \Rightarrow E+E$ E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

A Notational Shorthand

$expr \rightarrow expr \;\; op \;\; expr$

$expr \rightarrow (\; expr \;)$

$expr \rightarrow -\; expr$

$expr \rightarrow \mathbf{id}$

$op \rightarrow +$

$op \rightarrow -$

$op \rightarrow *$

$op \rightarrow /$

$expr \rightarrow expr \;\; op \;\; expr$

$\quad \mid (\; expr \;)$

$\quad \mid -\; expr$

$\quad \mid \mathbf{id}$

$op \rightarrow + \mid - \mid * \mid /$

Black : Nonterminal

Blue : Terminal

expr* : *Start Symbol

CFG for Programming Language

program	→	stmt-sequence
stmt-sequence	→	stmt-sequence **;** statement
	|	statement
Statement	→	if-stmt
	|	repeat-stmt
	|	assign-stmt
	|	read-stmt
	|	write-stmt
if-stmt	→	**if** exp **then** stmt-sequence **end**
	|	**if** exp **then** stmt-sequence **else** stmt-sequence **end**

Other Derivation Concepts

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

- Left-Most Derivation

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

- Right-Most Derivation (called *canonical derivation*

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- We will see that the ***top-down parsers*** try to find the ***left-most derivation*** of the given source program.
- We will see that the ***bottom-up parsers*** try to find the ***right-most derivation*** of the given source program in the reverse order.

Derivations Revisited

- A derivation encodes two pieces of information:
  - What productions were applied to produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

Derivation exercise 1

Productions:

***assign_stmt* → *id* := *expr* ;**

expr* → *expr op term

expr* → *term

term* → *id

term* → *real

term* → *integer

***op* → +**

***op* → -**

Let's derive:

id := id + real – integer ;

Please use left-most derivation

id := id + real – integer ;

Left-most derivation:	Using production:
assign_stmt	*assign_stmt* → *id := expr ;*
⇒ *id := expr ;*	*expr* → *expr op term*
⇒ *id := expr op term ;*	*expr* → *expr op term*
⇒ *id := expr op term op term ;*	*expr* → *term*
⇒ *id := term op term op term ;*	*term* → *id*
⇒ *id := id op term op term;*	*op* → *+*
⇒ *id := id + term op term ;*	*term* → *real*
⇒ *id := id + real op term ;*	*op* → *-*
⇒ *id := id + real - term ;*	*term* → *integer*
⇒ *id := id + real - integer;*	

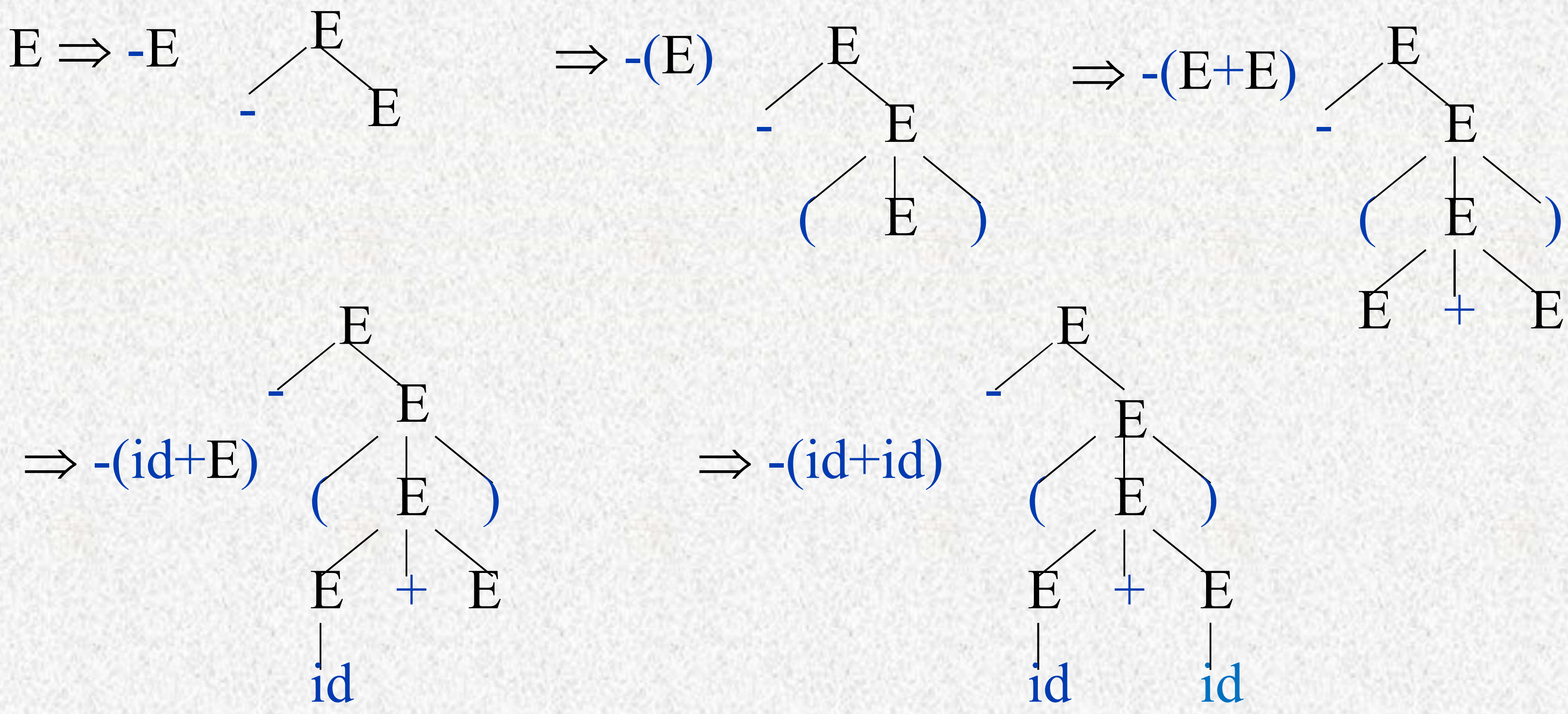
Parse Trees

- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- **A parse tree can be seen as a graphical representation of a derivation.**

EX. $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$



$E \rightarrow E \text{ op } E \mid (E) \mid -E \mid id$

$op \rightarrow + \mid - \mid * \mid /$

$E \Rightarrow E \text{ op } E$

$\Rightarrow id \text{ op } E$

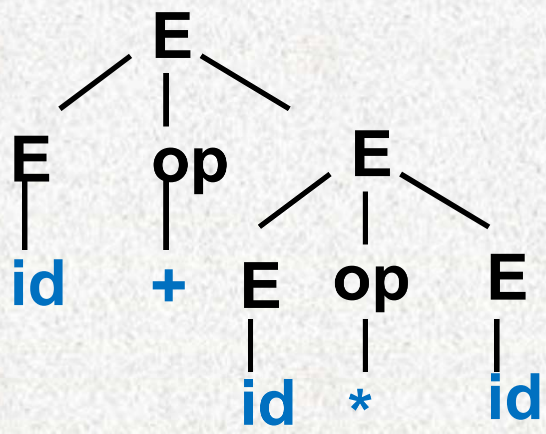
$\Rightarrow id + E$

$\Rightarrow id + E \text{ op } E$

$\Rightarrow id + id \text{ op } E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



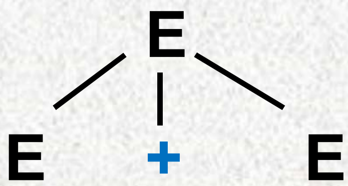
Parse Trees and Derivations

Consider the expression grammar:

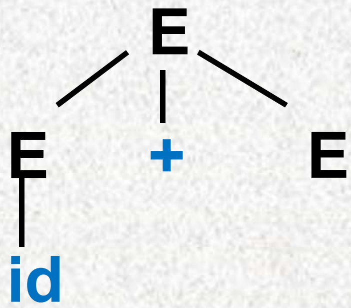
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

Leftmost derivations of $id + id * id$

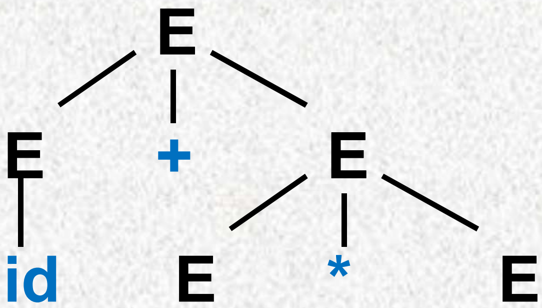
$E \Rightarrow E + E$



$E + E \Rightarrow id + E$

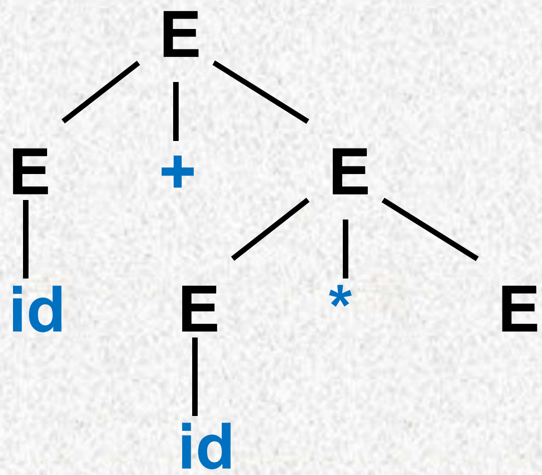


$id + E \Rightarrow id + E * E$

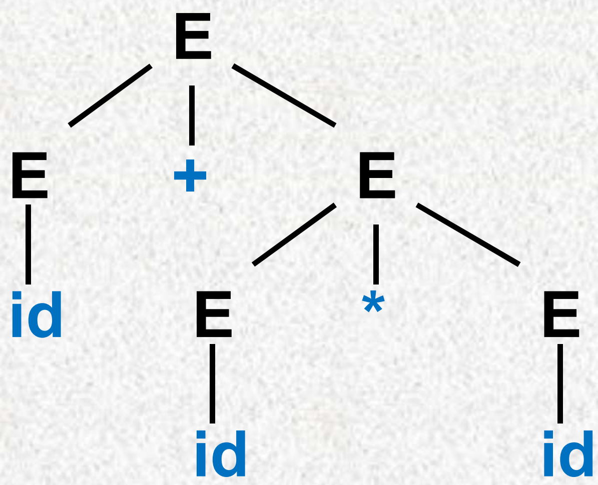


Parse Trees and Derivations (cont.)

$\text{id} + E * E \Rightarrow \text{id} + \text{id} * E$

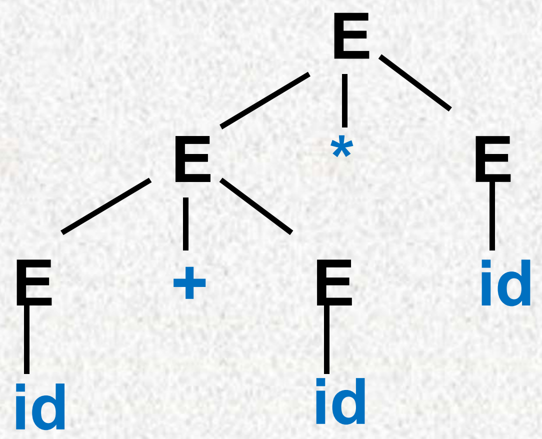


$\text{id} + \text{id} * E \Rightarrow \text{id} + \text{id} * \text{id}$



Alternative Parse Tree & Derivation

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$



WHAT'S THE ISSUE HERE ?

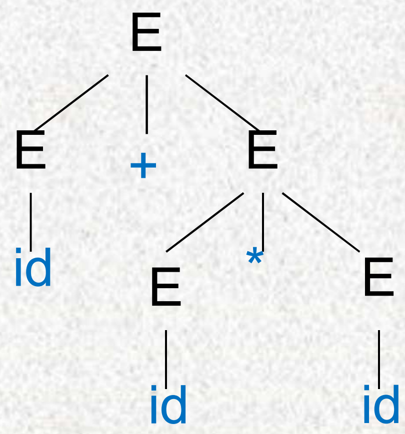
Two distinct leftmost derivations!

Challenges in Parsing

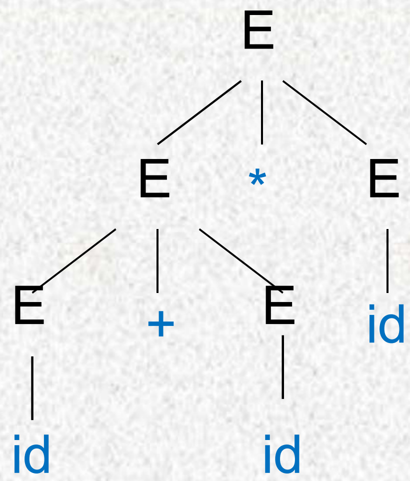
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$



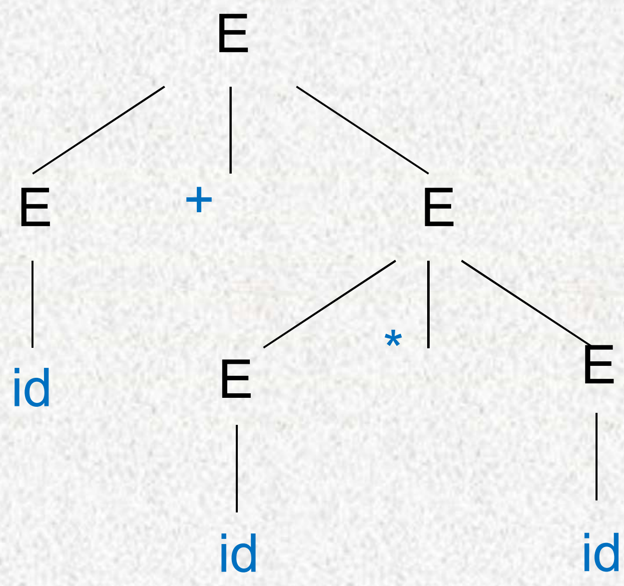
$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$

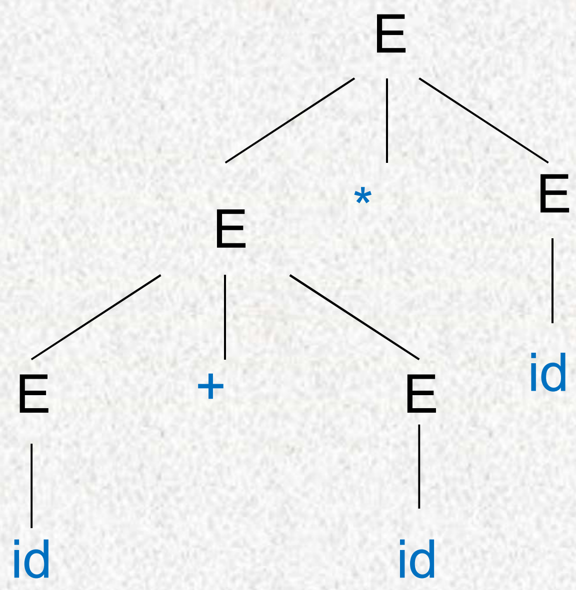


two parse trees for id+id*id.

Is Ambiguity a Problem?

Depends on semantics.





Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through layering.
  - Have exactly one way to build each piece of the string?
  - Have exactly one way of combining those pieces back together?

Resolving Ambiguity

- For the most parsers, the grammar must be unambiguous.
- **unambiguous grammar**
- ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity** rules.

$E \rightarrow E+E \mid E*E \mid E\hat{}E \mid id \mid (E)$

disambiguate the grammar

precedence: ^ (right to left)

* (left to right)

\+ (left to right)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G\hat{}F \mid G$

$G \rightarrow id \mid (E)$

Rewrite to eliminate the ambiguity

Or, simply tell which parse tree should be selected

Eliminating Ambiguity

Consider the following grammar segment:

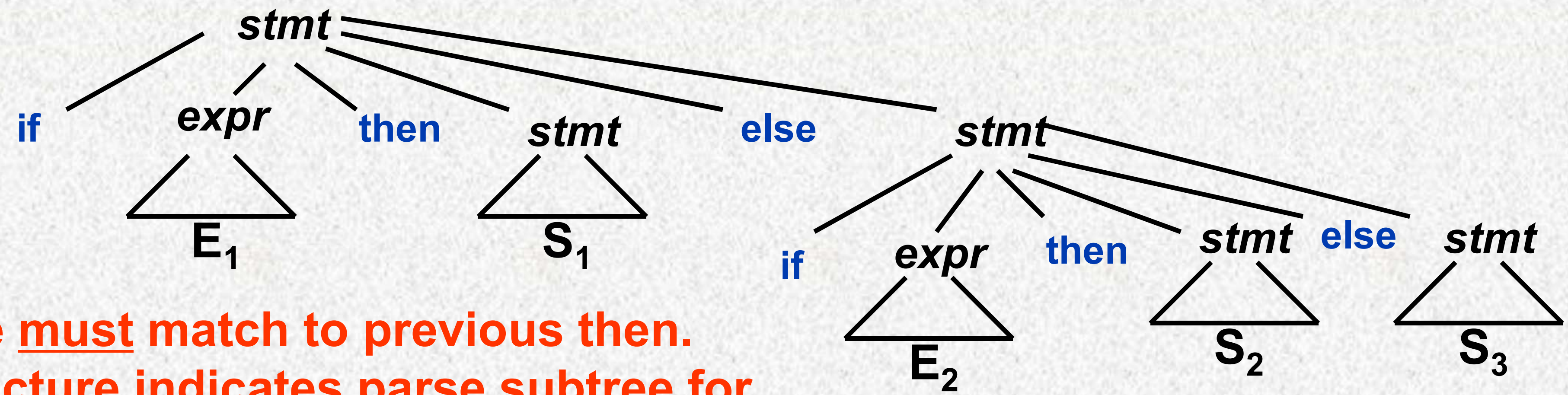
stmt $\rightarrow$ if *expr* then *stmt*

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

| other (any other statement)

What's problem here ?

Let's consider a simple parse tree:



Else must match to previous then. Structure indicates parse subtree for expression.

Example : What Happens with this string?

if E_1 then if E_2 then S_1 else S_2

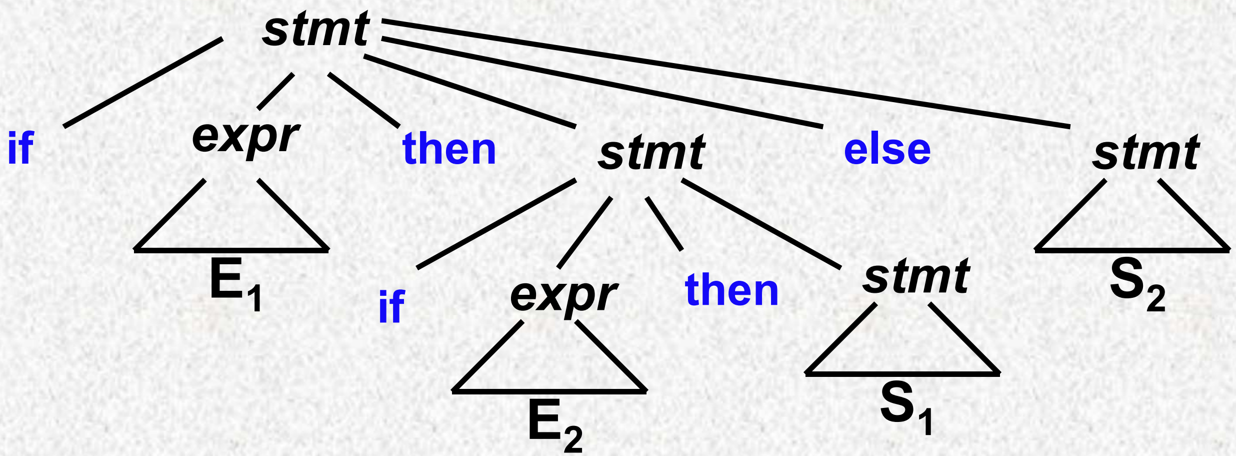
How is this parsed ?

if E_1 then
    if E_2 then
        S_1
else
        S_2
	vs.	if E_1 then
		    if E_2 then
		        S_1
		    else
		        S_2

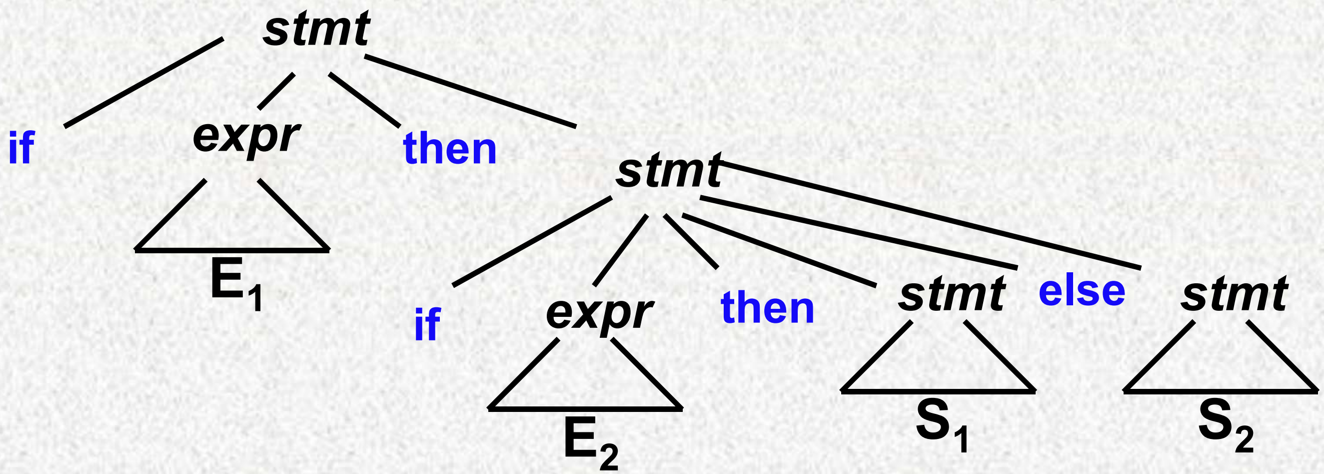
What's the issue here ?

Parse Trees for Example

Form 1:



Form 2:



What's the issue here ?

two parse trees for an ambiguous sentence.

Ambiguity (cont.)

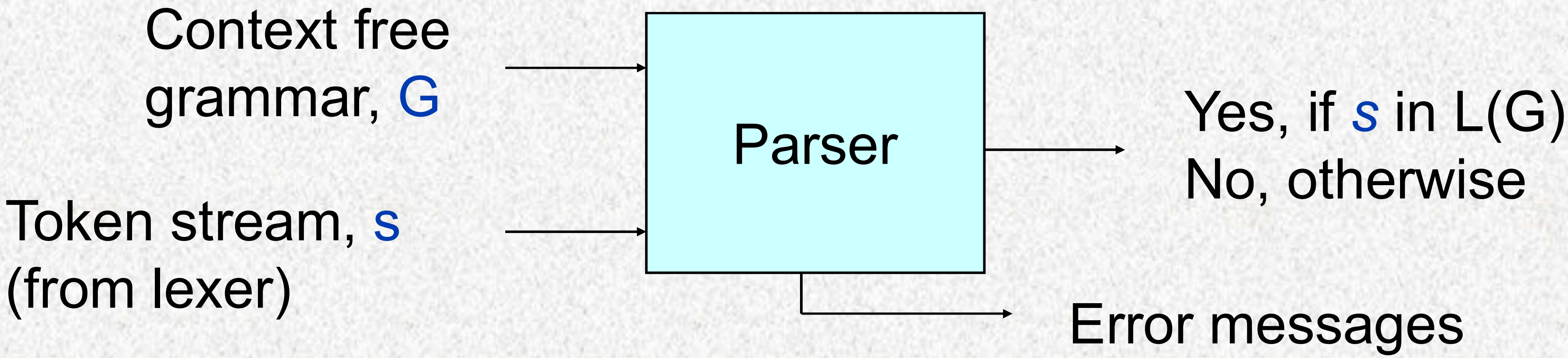
- We prefer the second parse tree (else matches with the closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt
| unmatchedstmt
matchedstmt → if expr then matchedstmt else matchedstmt
| otherstmts
unmatchedstmt → if expr then stmt
| if expr then matchedstmt else unmatchedstmt

The general rule is “match each **else** with the closest previous unmatched **then**.”

A Parser



- Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted

- Various kinds: **LL(k), LR(k), SLR, LALR**

Types

- Top-Down Parsing
  - Recursive descent parsing
  - Predictive parsing
  - LL(1)
- Bottom-Up Parsing
  - Shift-Reduce Parsing
  - LR parser

Homework

Page 206: Exercise 4.2.1
Page 207: Exercise 4.2.2 (d) (f) (g)

Top-Down Parsing

Two Key Points

$$
\begin{array}{rcl}
\textit{expression} & \rightarrow & \textit{expression} + \textit{term} \\
\textit{expression} & \rightarrow & \textit{expression} - \textit{term} \\
\textit{expression} & \rightarrow & \textit{term} \\
\textit{term} & \rightarrow & \textit{term} * \textit{factor} \\
\textit{term} & \rightarrow & \textit{term} \;/\; \textit{factor} \\
\textit{term} & \rightarrow & \textit{factor} \\
\textit{factor} & \rightarrow & (\ \textit{expression}\ ) \\
\textit{factor} & \rightarrow & \mathbf{id}
\end{array}
$$

```
expression => term
   => term*factor
   => term/factor*factor
```

– Q1: Which non-terminal to be replaced?

Leftmost derivation $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$

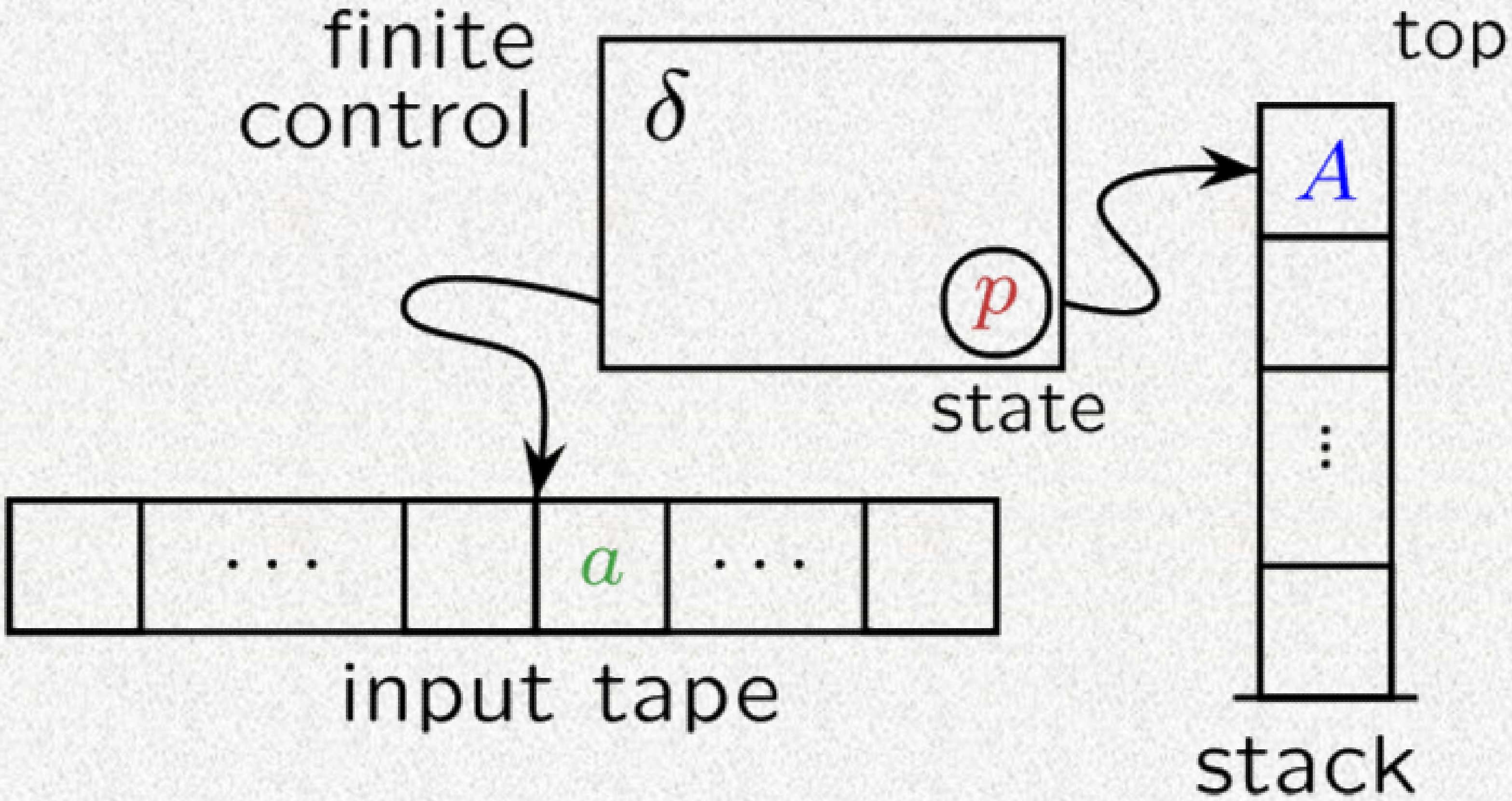
– Q2: Which production to be used?

Top-Down Parsing

The parse tree is created top to bottom (from root to leaves).

By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.

Pushdown Automaton



An illustration with PDA

P:
(1) Z → aBeA
(2) A → Bc
(3) B → d
(4) B → bB
(5) B → ε

a	b	e	c

Reading Head	Stack	Analysis	Derivation	Match?
abec	Z	Z production starting with a? - (1)	aBeA	a
bec	BeA	B production starting with b? –(4)	bBeA	b
ec	BeA	B production starting with e? -(5)	εeA	e
c	A	A production starting with c? -(2)(5)		

An illustration with PDA

P:

(1) Z → aBeA
(2) A → Bc
(3) B → d
(4) B → bB
(5) B → ε

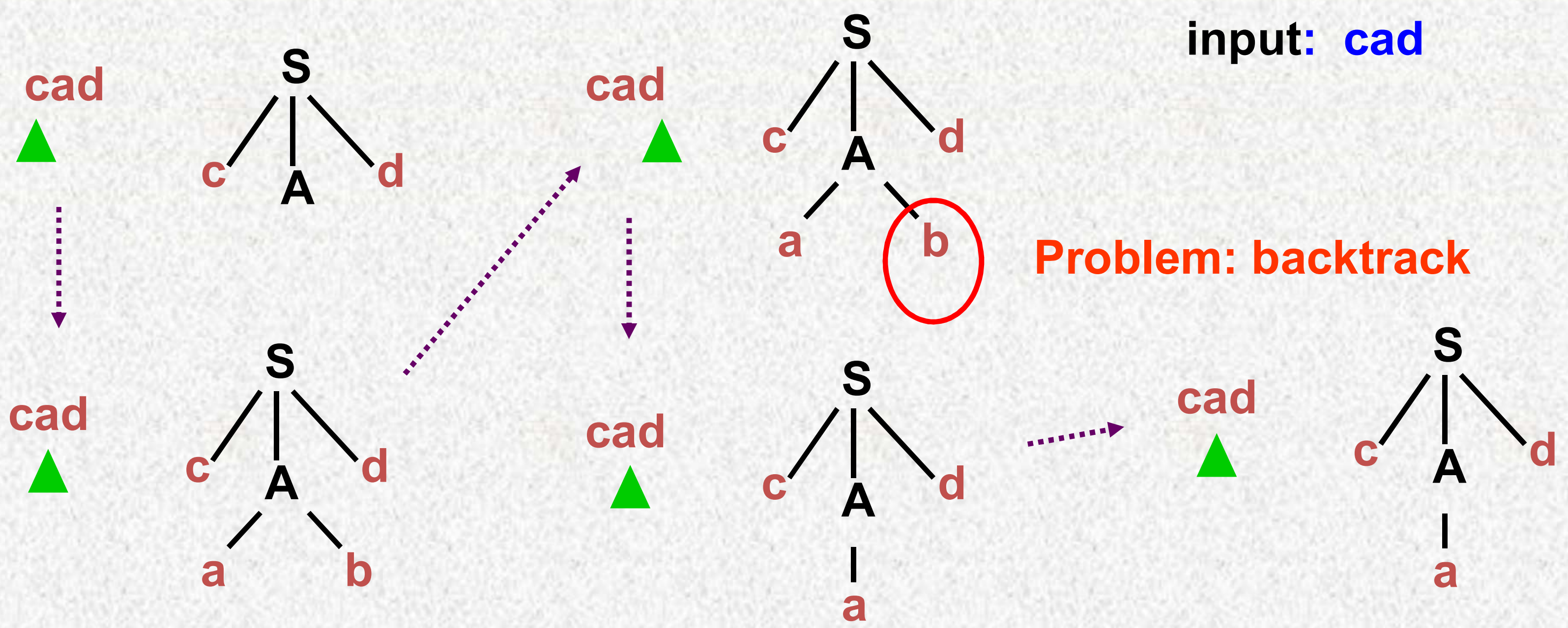
a	**b**	**e**	**c**

Reading Head	Stack	Analysis	Derivation	Match?
c	A	A production starting with c?-(2)	Bc	
c	Bc	A production starting with c? –(5)	εc	c

Problem - Backtraking

- General category of Top-Down Parsing
- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.

•Example: **$S \rightarrow c\ A\ d$**
$A \rightarrow ab \mid a$

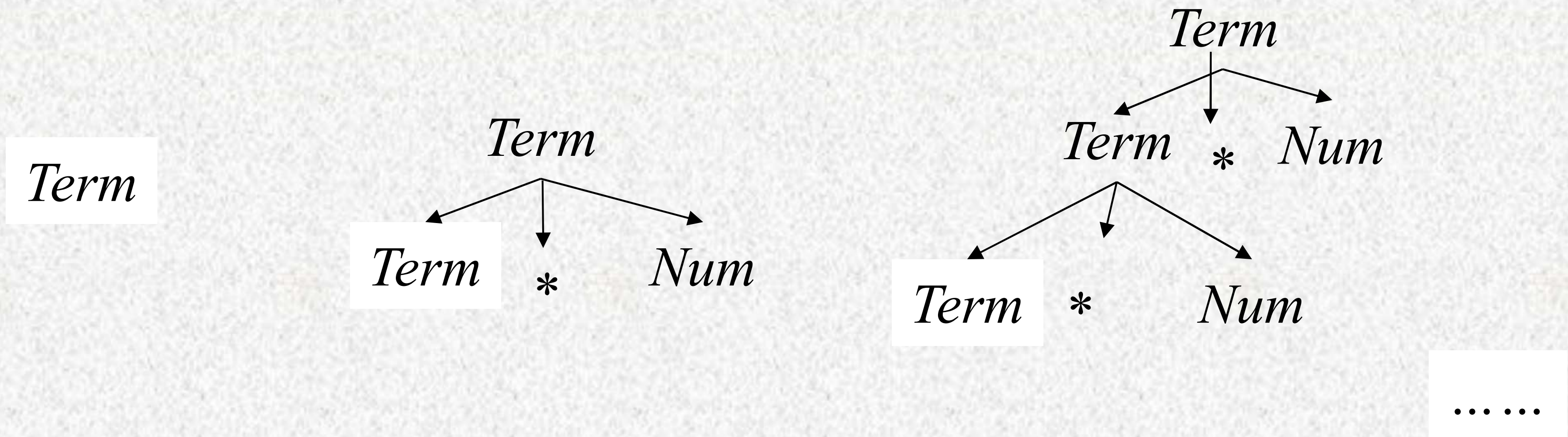


Problem – Left recursion

- A grammar is Left Recursion if it has a nonterminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α.

Left Recursion + top-down parsing = infinite loop

Eg. Term $\rightarrow$ Term*Num



Elimination of Left recursion

- Eliminating Direct Left Recursion

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$\beta_i \, \alpha_i{}^*$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

Elimination of Left recursion

- $A \rightarrow A\alpha \mid \beta$

  elimination of left recursion

  $P \rightarrow \beta P'$ $\qquad\qquad$ $P' \rightarrow \alpha P' \mid \varepsilon$

- $P \rightarrow P\alpha_1 \mid P\alpha_2 \mid \ldots \mid P\alpha^{1}_{m} \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- elimination of left recursion

  $P \rightarrow \beta_1 P' \mid \beta_2 P' \mid \ldots \mid \beta_n P'$

  $P' \rightarrow \alpha_1 P' \mid \alpha_2 P' \mid \ldots \mid \alpha_m P' \mid \varepsilon$

Elimination of Left recursion (eg.)

- G[E]: $E \rightarrow E+T|T$

  $T \rightarrow T*F|F$

  $F \rightarrow (E)|\ I$

Elimination of Left Recursion

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE'|\varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT'|\varepsilon \\
F &\rightarrow (E)|\ i
\end{aligned}
$$

Elimination of Left recursion (eg.)

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

P→ PaPb|BaP

- We have α = aPb, β=BaP
- So, P→ βP'

  P'→ αP'|ε
- 改写后：P→ BaPP'

  P'→ aPbP'|ε

Multiple P? Consider the most-left one.

Elimination of Indirect Left recursion

Direct： $S \rightarrow Sa$

Indirect： $S \rightarrow Aa,\ A \overset{+}{\rightarrow} Sb$, then we have $A \overset{+}{\rightarrow} Aab$

e.g： S $\rightarrow$ Aa | b, A $\rightarrow$ Sd |ε [1]

S => Aa => Sda

Elimination of Left recursion algorithm

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
arrange the nonterminals in some order A1, A2, ..., An.
for ( each i from 1 to n ) {
        for ( each j from 1 to i - 1 ) {
                replace each production of the form Ai → Aj γ by the
                    productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                    Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
        }
        eliminate the immediate left recursion among the Ai-productions
}
```

Elimination of Left recursion (eg.)

$S \rightarrow A\ b$
$A \rightarrow S\ a \mid b$

1:S
2:A

$A \rightarrow Aba \mid b$

$A \rightarrow bA'$
$A' \rightarrow baA' \mid \varepsilon$

Elimination of Left recursion (eg.)

$S \rightarrow Aa \mid b,$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

1:S
2:A

$S \rightarrow Aa \mid b,$
$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$S \rightarrow Aa \mid b,$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Elimination of Left recursion (eg.)

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow S\,a \mid a$

1:S
2:Q
3:R

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow Sa \mid a$
$\rightarrow (Qc|c)a \mid a$
$\rightarrow Qca \mid ca \mid a$
$\rightarrow (Rb|b)ca \mid ca \mid a$

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow (bca \mid ca \mid a)R'$
$R' \rightarrow bcaR' \mid \varepsilon$

Elimination of Left recursion (eg.)

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow S\,a \mid a$

1:R
2:Q
3:S

$R \rightarrow Sa \mid a$
$Q \rightarrow Rb \mid b \rightarrow Sab \mid ab \mid b$
$S \rightarrow Qc \mid c \rightarrow Sabc \mid abc \mid bc \mid c$

$S \rightarrow (abc \mid bc \mid c)S'$
$S' \rightarrow abcS' \mid \varepsilon$

Problem - Left Factoring

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
- elimination of left recursion

  $A \rightarrow \alpha A'$ $\qquad$ $A' \rightarrow \beta_1 \mid \beta_2$

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ $\qquad$ Which production to choose?
- elimination of left recursion

  $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Problem - Left Factoring

Algorithm 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{array}{l} A \to \alpha A' \mid \gamma \\ A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Problem - Left Factoring

- E.g
  - $S \rightarrow iEtS \mid iEtSeS \mid a$

    $E \rightarrow b$
  - For, S, the longest pre-fix is $iEtS$, Thus,

    $S \rightarrow iEtSS' \mid a$

    $S' \rightarrow eS \mid \varepsilon$

    $E \rightarrow b$

Problem - Left Factoring

- E.g.

G：

(1) $S \rightarrow aSb$

(2) $S \rightarrow aS$

(3) $S \rightarrow \varepsilon$

For (1)、(2)， extract the left factor：

$S \rightarrow aS(b|\varepsilon)$

$S \rightarrow \varepsilon$

We have G′：

$S \rightarrow aSA$

$A \rightarrow b$

$A \rightarrow \varepsilon$

$S \rightarrow \varepsilon$

Homework

Page 216: Exercise 4.3.1

Two Parsing Methods

A Naïve Method

- **Recursive-Descent Parsing**
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient

Recursive-Descent Parsing

void $A()$ {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

A typical procedure for a nonterminal in a top-down parse

Recursive-Descent Parsing

- Example

P:
(1) Z → aBd {a}
(2) B → d {d}
(3) B → c {c}
(4) B → bB {b}

a b c d

```
Z ( )
{
if (token == a)
 {   match(a);
      B( );
      match(d);
  }
else error();
}
```

```
B ( )
{
 case token of
  d:  match(d);break;
  c:   match(c); break;
  b:{  match(b);
        B( ); break;}
  other: error();
}
```

```
void main()
{read();
  Z();  }
```

A Non-Recursive Method

– **Predictive Parsing**

- no backtracking, efficient
- needs a special form of grammars (**LL(1) grammars**).
- Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

A Non-Recursive Method

- Predict($A \rightarrow \alpha$)
- First(α)
- Follow(A)

FIRST Set

FIRST(α)

If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

1. If X is terminal, then FIRST(X) is {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is nonterminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$. is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow \varepsilon$, then we add FIRST($Y_2$) and so on.

Now, we can compute FIRST for any string $X_1X_2 \ldots X_n$ as follows. Add to FIRST($X_1X_2 \ldots X_n$) all the non-ε symbols of FIRST(X_1). Also add the non-ε symbols of FIRST(X_2) if ε is in FIRST(X_1), the non-ε symbols of FIRST(X_3) if ε is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ε to FIRST($X_1X_2 \ldots X_n$) if, for all i, FIRST(X_i) contains ε.

FIRST Example

- First(α)

First(E'T'E) =?
First(T'E') = ?

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

P:
(1) $E \rightarrow TE'$
(2) $E' \rightarrow + TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow * F T'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow i$
(9) $F \rightarrow n$

Sε = {E', T'}

First(E'T'E) = {+,*,i,n,(}
First(T'E') = {+,*,ε}

Motivation Behind FIRST

- Is used to help find the appropriate reduction to follow given the top-of-the-stack non-terminal and the current input symbol.
- If $A \rightarrow \alpha$, and a is in FIRST(α), then when *a*=input, replace A with α. (*a* is one of first symbols of α, so when A is on the stack and a is input, POP A and PUSH α.)

Example:

$A \rightarrow aB \mid bC$
$B \rightarrow b \mid dD$
$C \rightarrow c$
$D \rightarrow d$

G: $E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow (E) | i$

input: i+i $

E: FIRST(TE′) = { (, i }
E′: FIRST(+TE′) = { + }   FIRST(ε) = { ε }
T: FIRST(FT′) = { (, i }
T′: FIRST(*FT′) = { * }   FIRST(ε) = {ε}
F: FIRST((E)) = { (}   FIRST(i) = {i}

i + i   i∈FIRST(TE′)

↑

i + i   i∈FIRST(FT ′)

↑

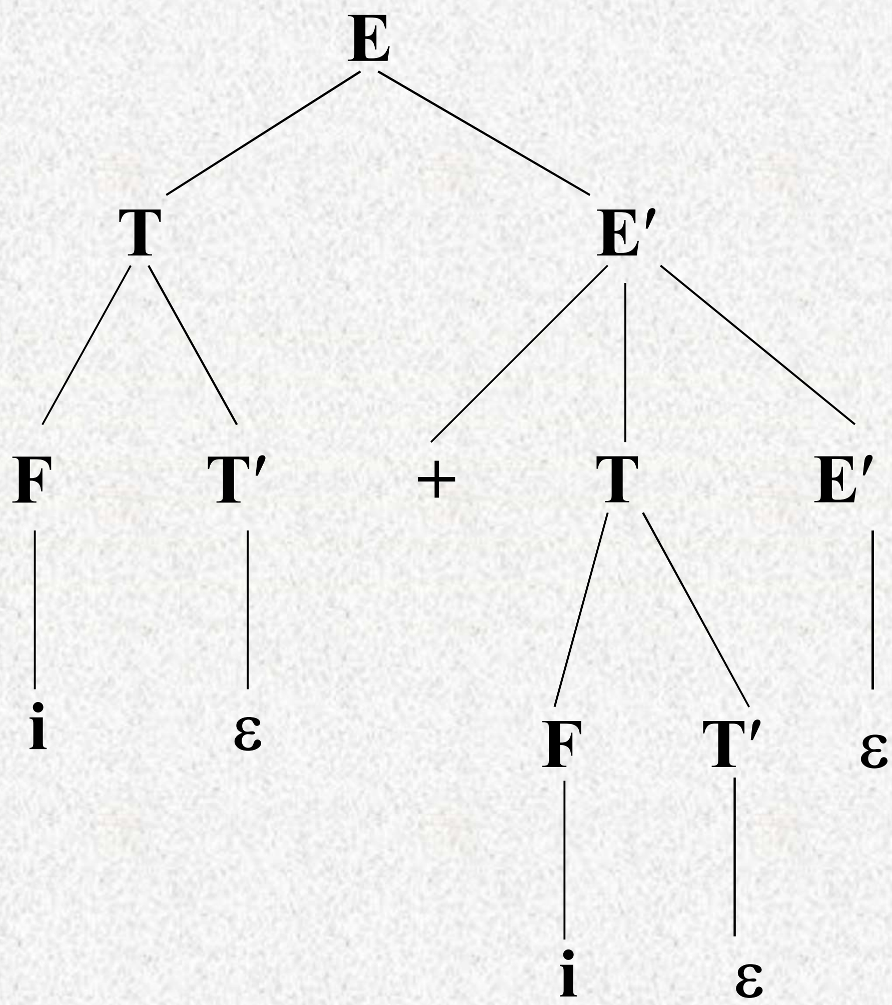
i + i   i∈FIRST(i)

↑

i + i   use $T' \rightarrow \varepsilon$

↑

……



Left Most Derivation of the Example

$E \Rightarrow TE'$

$\Rightarrow FT'E'$

$\Rightarrow iT'E'$

$\Rightarrow i\varepsilon E'$

$\Rightarrow i\varepsilon + TE'$

$\Rightarrow i\varepsilon+FT'E'$

$\Rightarrow i\varepsilon+iT'E'$

$\Rightarrow i\varepsilon+i\varepsilon E'$

$\Rightarrow i\varepsilon+i\varepsilon\varepsilon = i+i$

- $E \rightarrow TE'$
  $T \rightarrow FT'$
  $F \rightarrow i$
  $T' \rightarrow \varepsilon$
  $\boxed{E' \rightarrow +TE'}$
  $T \rightarrow FT'$
  $F \rightarrow i$
  $T' \rightarrow \varepsilon$
  $E' \rightarrow \varepsilon$

FOLLOW Set

Define FOLLOW(A), for nonterminal A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form, that is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow \alpha A a \beta$ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and *a*, but if so, they derived ε and disappeared. If A can be the rightmost symbol in some sentential form, then \$, representing the input right endmarker, is in FOLLOW(A).

FOLLOW Set (cont.)

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.

2. If there is a production $A \Rightarrow \alpha B \beta$, then everything in FIRST(β), except for ε, is placed in FOLLOW(B).

3. If there is a production $A \Rightarrow \alpha B$, or a production $A \Rightarrow \alpha B \beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

FOLLOW Set Example

P:

(1) E → TE'
(2) E' → + TE'
(3) E' → ε
(4) T → FT'
(5) T' → * F T'
(6) T' → ε
(7) F → (E)
(8) F → i
(9) F → n

First(X)

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

Follow(X)

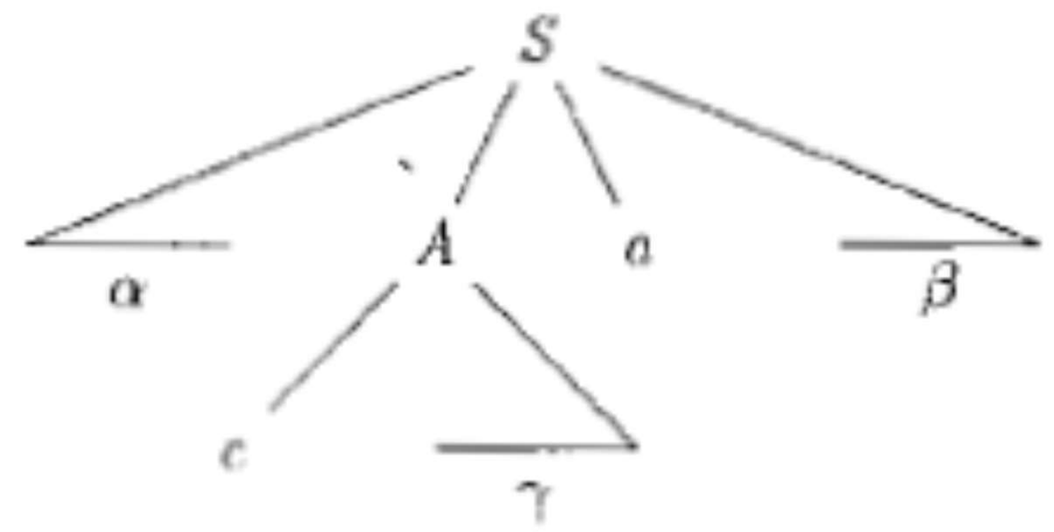
E	{#,)}
E'	{#,)}
T	{+,), #}
T'	{+,), #}
F	{*, +,), #}

Motivation Behind FOLLOW

- Is used when FIRST has a conflict, to resolve choices, or when FIRST gives no suggestion. When $\alpha \to \in$ or $\alpha \Rightarrow^* \varepsilon$, then what follows A dictates the next choice to be made.
- If $A \to \alpha$, and *b* is in FOLLOW(A), then when $\alpha \Rightarrow^* \varepsilon$ and *b* is an input character, then we expand A with α , which will eventually expand to ε, of which *b* follows! ($\alpha \Rightarrow^* \varepsilon$: i.e., FIRST(α) contains ε.)

Motivation Behind FOLLOW

$S \Rightarrow^* \alpha A a \beta$



a is in Follow(A); c is in First(A)

Predict Set

- Predict($A \rightarrow \alpha$)
  - Predict($A \rightarrow \alpha$) = First(α), if $\varepsilon \notin$ First(α);
  - Predict($A \rightarrow \alpha$) = First(α)- $\{\varepsilon\} \cup$ Follow(A), if $\varepsilon \in$ First(α);

Predict Set Example

P:

(1) $E \to TE'$ → First(TE')={i, n,(}

(2) $E' \to + TE'$ → First(+TE')={+}

(3) $E' \to \varepsilon$ → Follow(E')={#,)}

(4) $T \to FT'$ → First(FT')={i,n,(}

(5) $T' \to * F T'$ → First(*FT')={*}

(6) $T' \to \varepsilon$ → Follow(T')={),+, # }

(7) $F \to (E)$ → First((E))={ (}

(8) $F \to i$ → First(i)={i}

(9) $F \to n$ → First(n)={n}

first

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

Follow

E	{#,)}
E'	{#,)}
T	{+,), #}
T'	{+,), #}
F	{*, +,), #}

Now We consider LL(1)

Simple Predictive Parser: LL(1)

- Top-down, predictive parsing:
  - L: Left-to-right scan of the tokens
  - L: Leftmost derivation.
  - (1): One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a nonterminal, we predict the production to use by looking at the next token of the input. The decision is forced.

LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \to \alpha$ and $A \to \beta$
  - Both α and β cannot derive strings starting with same terminals.

    $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$, $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing \quad (1 \le i \ne j \le n)$
  - At most one of α and β can derive to ε.
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

    If $\varepsilon \in FIRST(\alpha_i)(1 \le i \le n)$, then $FIRST(\alpha_i) \cap FOLLOW(A) = \varnothing$

NOW predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.

Predictive Parser

a grammar ➔ ➔ a grammar suitable for predictive
eliminate left parsing (a LL(1) grammar)
left recursion factor no %100 guarantee.

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the **current symbol** in the input string.

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

input: ... a ....

current token

Revisit LL(1) Grammar

LL(1) grammars

== there have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

- Grammar can't be ambiguous or left recursive
- Grammar is LL(1) $\Leftrightarrow$ when $A \rightarrow \alpha \mid \beta$
  1. α & β do not derive strings starting with the same terminal a
  2. Either α or β can derive ε, but not both.

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

A Grammar which is not LL(1)

- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Examples

- Example: $S \rightarrow c\ A\ d \qquad A \rightarrow aa \mid a$

  Left Factoring: $S \rightarrow c\ A\ d \qquad A \rightarrow aB \qquad B \rightarrow a \mid \varepsilon$

- Example： $S \rightarrow Sa \mid *$

  Eliminate left recursion: $S \rightarrow *B \qquad B \rightarrow aB \mid \varepsilon$

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn’t eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar’s) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.

$S \to iEtSS'|a \quad S' \to eS|\varepsilon \qquad E \to b$

FIRST(S) = {i,a} FIRST(iEtSS′) = {i} FIRST(a)={a}
FIRST(S′) = {e, ε} FIRST(eS) = {e} FIRST(ε)={ε}
FIRST(E) = {b} FIRST(b) = {b}

FELLOW(S) = {e, \$}
FELLOW(S′) = {e, \$}
FELLOW(E) = {t}

V_N	input symbol					
	a	b	e	i	T	\$
S	$S \to a$			$S \to iEtSS'$		
S′			$S' \to eS$ $\cancel{S' \to \varepsilon}$			$S' \to \varepsilon$
E		$E \to b$				

LL(1) Process Illustration

- LL1-example.pdf