

## Production Rules

- Here's a production rule:

$\text{IfStat} \rightarrow \underline{\text{if}} \ ( \ \underline{\text{expr}} \ ) \ \underline{\text{stat}}$

- This rule states that to construct an if-statement in C you have to type
  1. an `if`, then
  2. a `(`, then
  3. some sort of expression, then
  4. a `)`, then finally
  5. some sort of statement.

## A Grammar for English

- A **grammar** can be used for
  1. sentence generation (i.e. which sentences does this grammar generate?), or
  2. parsing (i.e. is sentence  $S$  generated by this grammar?).
- Let's look at a simple grammar for a fragment of English.

## Syntax

- The **syntax** of a language (formal or natural) is the way the words in a sentence/program can be arranged.
- `「eats dog bone the」` is not a legal arrangement of words in English.
- `「= y x + 5」` is not a legal arrangement of tokens in Java.
- Somehow, we need to describe what constitutes legal and illegal sentences in a particular language.
- We use **production rules** to describe the syntax of a language.



University of  
Arizona

CSC 453

Compilers and Systems Software

Christian Collberg  
September 23, 2002

Grammars

Copyright © 2002 C. Collberg

Slide 5-1

Slide 5-2

Slide 5-3

## Sentence Generation

1. Start with the start symbol.
2. Pick a non-terminal  $X$  on the right hand side.
3. Pick a grammar rule  $X \rightarrow \gamma$ .
4. Replace  $X$  with  $\gamma$ .
5. Repeat until left with a string of words.

$S$	$\xRightarrow{S \rightarrow NP VP}$	NP VP
	$\xRightarrow{NP \rightarrow N}$	N VP
	$\xRightarrow{N \rightarrow John}$	John VP
	$\xRightarrow{VP \rightarrow V NP}$	John V NP
	$\xRightarrow{V \rightarrow kissed}$	John kissed NP
	$\xRightarrow{NP \rightarrow N}$	John kissed N
	$\xRightarrow{N \rightarrow Lisa}$	John kissed Lisa

## Syntactic Categories

**S** [Sentence] John likes Sarah's black hair

**N** [Noun] John, hair

**V** [Verb] eating, sat

**Adj** [Adjective] black, long

**Det** [Determiner] the, a, every

**NP** [Noun Phrase] Sarah's long black hair

**VP** [Verb Phrase] eating apples

## Terminology

- A grammar is a 4-tuple  
(non-terminals, terminals, productions, start-symbol)

or

$(N, \Sigma, P, S)$

- A production is of the form  $\alpha \rightarrow \beta$  where  $\alpha, \beta$  are taken from  $N \cup \Sigma$ .
- Read  $\alpha \rightarrow \beta$  as "rewrite  $\alpha$  with  $\beta$ ".
- Read  $\Rightarrow$  as "directly derives".
- Read  $\xRightarrow{r}$  as "directly derives using rule  $r$ ".
- Read  $\xRightarrow{*}$  as "derives in one or more steps".

## A Simple English Grammar

$S \rightarrow NP VP$

$VP \rightarrow V NP$

$VP \rightarrow V$

$NP \rightarrow N$

$NP \rightarrow Det N$

$N \rightarrow John$

$N \rightarrow Lisa$

$N \rightarrow house$

$V \rightarrow died$

$V \rightarrow kissed$

$Det \rightarrow the$

$Det \rightarrow a$

- S, NP, VP, N, Det, V are **non-terminal** symbols.
- John, Lisa, house, died, ... are **terminal symbols**.
- S is the **start symbol**.

## Terminology...

- Our English grammar is the 4-tuple  
( $\{S, NP, V, \dots\}$ ,  
 $\{\text{John, house, died, } \dots\}$ ,  
 $\{S \rightarrow NP VP, VP \rightarrow V, \dots\}$ ,  
 $S$ )
- Our PL grammar is the 4-tuple  
( $\{\text{Program, Stat, } \dots\}$ ,  
 $\{\text{BEGIN, :=, *, } \dots\}$ ,  
 $\{\text{Program} ::= \text{BEGIN Stat END, } \dots\}$ ,  
 $\text{Program}$ )

## Parse Trees

- We often want to show how a particular sentence was derived. We can do this without listing all the steps explicitly by drawing a **parse tree**.
- A parse tree is a tree where
  1. The root is labeled by the start symbol.
  2. Each leaf is labeled by a terminal symbol.
  3. Each interior node is labeled by a non-terminal symbol.

## A Simple PL Grammar

Here's a grammar for a simple programming language:

$\text{Program} ::= \text{BEGIN Stat END}$

$\text{Stat} ::= \text{ident} := \text{Expr}$

$\text{Expr} ::= \text{Expr} \pm \text{Expr} \mid$

$\text{Expr} * \text{Expr} \mid$

$\text{ident} \mid \text{number}$

- We write terminal symbols like this.
- We write non-terminal symbols like this.
- Sometimes we write  $::=$  instead of  $\rightarrow$ .
- $A \rightarrow b \mid c$  is the same as  $A \rightarrow b; A \rightarrow c$ . Read  $\mid$  as “or”.

## A Simple PL Grammar...

We know the sentence

$\text{BEGIN } a := 5 + 4 * 3 \text{ END}$

is in the language because we can derive it from the start symbol:

$\text{Program} \Rightarrow \text{BEGIN Stat END}$

$\Rightarrow \text{BEGIN ident} := \text{Expr END}$

$\Rightarrow \text{BEGIN "a"} := \text{Expr END}$

$\Rightarrow \text{BEGIN "a"} := \text{Expr} \pm \text{Expr END}$

$\Rightarrow \text{BEGIN "a"} := 5 \pm \text{Expr END}$

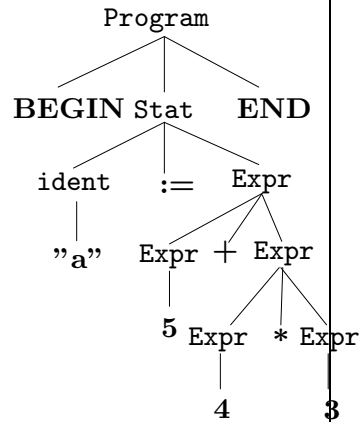
$\Rightarrow \text{BEGIN "a"} := 5 \pm \text{Expr} * \text{Expr END}$

$\Rightarrow \text{BEGIN "a"} := 5 \pm 4 * \text{Expr END}$

$\Rightarrow \text{BEGIN "a"} := 5 \pm 4 * 3 \text{ END}$

## Parse Trees...

Program  $\Rightarrow$  BEGIN Stat END  
 $\Rightarrow$  BEGIN ident := Expr END  
 $\Rightarrow$  BEGIN "a" := Expr END  
 $\Rightarrow$  BEGIN "a" := Expr + Expr END  
 $\Rightarrow$  BEGIN "a" := 5 + Expr END  
 $\Rightarrow$  BEGIN "a" := 5 + Expr \* Expr END  
 $\Rightarrow$  BEGIN "a" := 5 + 4 \* Expr END  
 $\Rightarrow$  BEGIN "a" := 5 + 4 \* 3 END

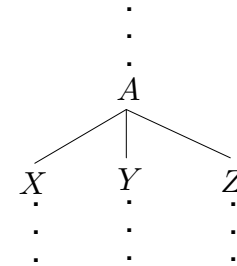


## Parse Trees...

- If one step of our derivation is

$$\dots A \dots \Rightarrow \dots X Y Z \dots$$

(i.e, we used the rule  $A \rightarrow XYZ$ ) then we'll get a parse (sub-)tree



Slide 5-12

## Regular Grammars

- A grammar is **regular** if all rules are of the form

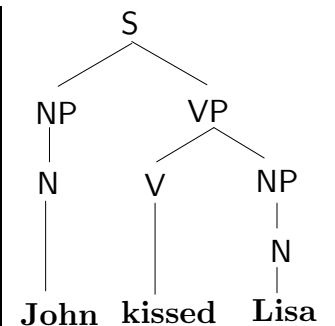
$$A \rightarrow aB$$

$$A \rightarrow a$$

- By convention, the symbols  $A, B, C, \dots$  are non-terminals,  $a, b, c, \dots$  are terminals, and  $\alpha, \beta, \gamma, \dots$  are strings of symbols.
- Regular grammars are used to describe the lexical structure of programs, i.e. what tokens look like.

## Parse Trees...

$S \xRightarrow{S \rightarrow NP VP} NP VP$   
 $\xRightarrow{NP \rightarrow N} N VP$   
 $\xRightarrow{N \rightarrow John} John VP$   
 $\xRightarrow{VP \rightarrow V NP} John V NP$   
 $\xRightarrow{V \rightarrow kissed} John kissed NP$   
 $\xRightarrow{NP \rightarrow N} John kissed N$   
 $\xRightarrow{N \rightarrow Lisa} John kissed Lisa$



Slide 5-13

Slide 5-14

Slide 5-15

## EBNF...

- In BNF, our variable declaration

```
int x,y,z,w,...;
```

looks like this:

```
vars ::= ident ident idlist ;
```

```
idlist ::= , ident idlist |  $\epsilon$ 
```

- In EBNF, it looks like this:

```
vars ::= ident ident { , ident } ;
```

- I.e.  $\{e\}$  means that  $e$  is repeated 0 or more times.

## EBNF...

- In BNF, our class declaration

```
class C extends D { ... }
```

looks like this:

```
class ::= class ident extends { ... }
```

```
extends ::= extends ident |  $\epsilon$ 
```

- In EBNF, it looks like this:

```
class ::= class ident [extends ident] { ... }
```

- I.e.  $[e]$  means that  $e$  is optional.

## Context-Free Grammars

- Programming language syntax is described by a **context free grammar** (CFG).

- In a CFG all rules are of the form

$$A \rightarrow \gamma$$

$\gamma$  is any sequence of terminals or non-terminals.  $A$  is a single non-terminal.

- Example: an **if**-statement consists of an **if**-token, expression, **then**-token, statement, and (maybe) an **else**-token followed by a statement.

## EBNF

- BNF is **Backus-Naur Form**, a way to write CFGs. EBNF (**Extended** BNF) is a more expressive way to write CFGs.

- Repetition and choice are common structures in a language (and hence, its grammar).

- Repetition:

```
int x,y,z,w,...;
```

- Choice:

```
class C { ... }
```

```
class C extends D { ... }
```

## EBNF for Luca...

statement ::=  
designator := expression |  
WRITE expression | READ designator | Writeln  
ident( [ actual\_list ] )  
IF expression THEN stat\_seq [ ELSE stat\_seq ] ENDIF |  
FOR ident := expression TO expression [ BY expression ] DO  
stat\_seq ENDFOR |  
WHILE expression DO stat\_seq ENDDO |  
REPEAT stat\_seq UNTIL expression |  
LOOP stat\_seq ENDLOOP | EXIT

expression ::=  
expression bin\_operator expression | unary\_operator  
expression |  
( expression ) |  
real\_literal | integer\_literal | char\_literal | string\_literal |  
designator |  
designator ::=  
ident | designator [ expression ] | designator :: ident  
unary\_operator ::= - | TRUNC | FLOAT | NOT  
bin\_operator ::= + | - | \* | / | % | < | <= | = | # | >= | > | AND | OR

## EBNF for Luca

program ::=  
PROGRAM ident ; decl\_list block .  
decl\_list ::=  
{ declaration ; }  
declaration ::=  
VAR ident : ident |  
TYPE ident = RECORD [ [ field\_list ] ] |  
TYPE ident = ARRAY expression OF ident |  
CONST ident : ident = expression |  
PROCEDURE ident ( [ formal\_list ] ) decl\_list block ;

## EBNF for Luca...

field\_list ::= field\_decl { ; field\_decl }  
field\_decl ::= ident : ident  
formal\_list ::= formal\_param { ; formal\_param }  
formal\_param ::= [ VAR ] ident : ident  
actual\_list ::= expression { , expression }  
block ::= BEGIN stat\_seq END  
stat\_seq ::= { statement ; }

## Operator Precedence

- The **precedence** of an operator is a measure of its **binding power**, i.e. how strongly it attracts its operands.
- Usually  $*$  has higher precedence than  $+$ :

$$4 + 5 * 3$$

means

$$4 + (5 * 3),$$

not

$$(4 + 5) * 3.$$

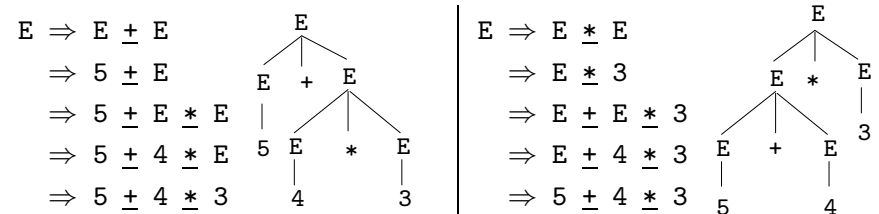
- We say that  $*$  binds harder than  $+$ .

## Ambiguous Grammars

- A grammar is ambiguous if some string of tokens can produce two (or more) different parse trees.

$$E ::= E + E \mid E * E \mid \text{number}$$

$$5 + 4 * 3$$



## Operator Associativity

- The **associativity** of an operator describes how operators of equal precedence are grouped.
- $+$  and  $-$  are usually **left associative**:

$$4 - 2 + 3$$

means

$$(4 - 2) + 3 = 5,$$

not

$$4 - (2 + 3) = -1.$$

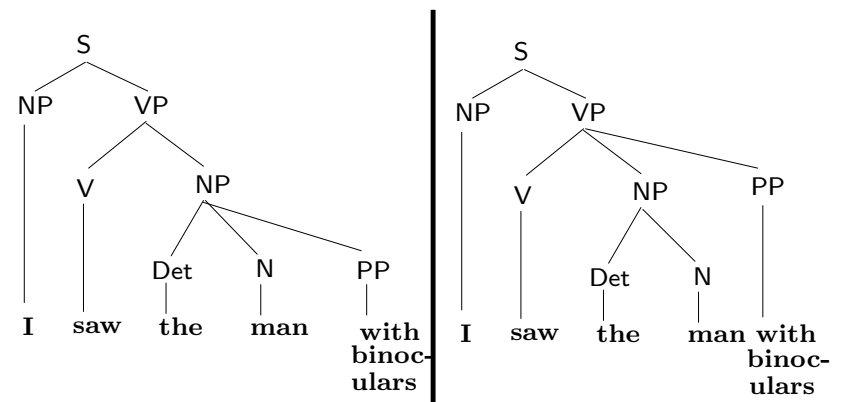
We say that  $+$  **associates to the left**.

- $\wedge$  associates to the right:

$$2 \wedge 3 \wedge 4 = 2 \wedge (3 \wedge 4).$$

## Structural Ambiguity in English

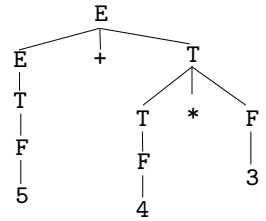
- Ambiguities occur in natural languages also:



- 1. Create one non-terminal for each precedence level, for example  $p_1, p_2, \dots, p_n$ , where  $p_n$  has the highest precedence level.
- 2. For operator **op** at precedence level  $i$  construct the following production if the operator is
  - left associative:  
 $p_i ::= p_i \text{ op } p_{i+1} \mid p_{i+1}$
  - right associative:  
 $p_i ::= p_{i+1} \text{ op } p_i \mid p_{i+1}$
- 3. Construct a production for nonterminal  $p_{n+1}$  which represents **primary** expressions such as identifiers, numbers, parenthesized expressions, etc:  
 $p_{n+1} ::= (p_1) \mid \text{num} \mid \text{id}$

$E ::= E \pm T \mid T$   
 $T ::= T * F \mid F$   
 $F ::= \underline{\text{number}}$

5 + 4 \* 3



Operators in C

OPERATOR	KIND	PREC	ASSOC
*, /, %	Binary	13	Left
+, -	Binary	12	Left
<<, >>	Binary	11	Left
<, >, <=, >=	Binary	10	Left
==, !=	Binary	9	Left
&	Binary	8	Left
^	Binary	7	Left
	Binary	6	Left
&&	Binary	5	Left
	Binary	4	Left
? :	Ternary	3	Right
=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=,  =	Binary	2	Right
,	Binary	1	Left

Expression Grammars

- We must write unambiguous expression grammars that reflect the associativity and precedence of all operators.
- The next slide gives the algorithm for writing such grammars.

\_\_\_\_\_ Resulting Expression Grammar: \_\_\_\_\_

$\text{expr} ::= \text{expr} \pm \text{term} \mid \text{term}$   
 $\text{term} ::= \text{term} * \text{factor} \mid \text{factor}$   
 $\text{factor} ::= ( \text{expr} ) \mid \underline{\text{number}}$



## Abstract Syntax...

- The abstract syntax also describes the structure of the **abstract syntax tree** (AST).
- Each abstract syntax rule represents the structure of an AST node-type.
- A parser converts from the program's concrete syntax to its corresponding abstract syntax, i.e. it reads the source code of the input program and produces an AST.

$E \Rightarrow E \underline{+} T$	$E \Rightarrow E \underline{+} T$
$\Rightarrow T \underline{+} T$	$\Rightarrow E \underline{+} T \underline{*} F$
$\Rightarrow F \underline{+} T$	$\Rightarrow E \underline{+} T \underline{*} 3$
$\Rightarrow 5 \underline{+} T$	$\Rightarrow E \underline{+} F \underline{*} 3$
$\Rightarrow 5 \underline{+} T \underline{*} F$	$\Rightarrow E \underline{+} 4 \underline{*} 3$
$\Rightarrow 5 \underline{+} F \underline{*} F$	$\Rightarrow T \underline{+} 4 \underline{*} 3$
$\Rightarrow 5 \underline{+} 4 \underline{*} F$	$\Rightarrow F \underline{+} 4 \underline{*} 3$
$\Rightarrow 5 \underline{+} 4 \underline{*} 3$	$\Rightarrow 5 \underline{+} 4 \underline{*} 3$

## Grammar Example I

Concrete Grammar:

$S ::= \underline{\text{ident}} \underline{:=} E \mid$   
 $\quad \underline{\text{if}} \ E \ \underline{\text{then}} \ SS_1 [\underline{\text{else}} \ SS_2] \ \underline{\text{end}} \mid \underline{\text{while}} \ E \ \underline{\text{do}} \ SS \ \underline{\text{end}} \mid \epsilon$   
 $SS ::= S \ ; \ SS \mid \epsilon$

Abstract Grammar:

$\text{Assign} ::= \underline{\text{ident}} \ \text{Expr}$   
 $\text{If} ::= \text{Expr} \ \text{StatSeq}$   
 $\text{IfElse} ::= \text{Expr} \ \text{StatSeq} \ \text{StatSeq}$   
 $\text{While} ::= \text{Expr} \ \text{StatSeq}$   
 $\text{Stat} ::= \text{Assign} \mid \text{If} \mid \text{IfElse} \mid \text{While}$   
 $\text{StatSeq} ::= \text{Stat} \ \text{StatSeq} \mid \text{NULL}$

## Abstract Syntax

- We distinguish between a language's **concrete** and **abstract** syntax.
- The concrete syntax describes the textual layout of programs written in the language, eg. what if-statements look like.
- The abstract syntax describes the **logical** structure of the language; eg. that if-statements consist of three parts (expression, statement, statement).

## Concrete Grammar Example II

$\text{Program} ::= \text{program } \underline{\text{ident}} ; \text{DeclSeq } \underline{\text{begin}} \text{ StatSeq } \underline{\text{end}} .$

$\text{DeclSeq} ::= \text{Decl} ; \text{DeclSeq} \mid \epsilon$

$\text{Decl} ::= \text{var } \underline{\text{ident}} : \underline{\text{ident}}$

$\text{Stat} ::= \underline{\text{ident}} := \text{Expr} \mid \text{if } \text{Expr} \text{ then } \text{StatSeq} \text{ else } \text{StatSeq}$

$\text{StatSeq} ::= \text{Stat} ; \text{StatSeq} \mid \epsilon$

$\text{Expr} ::= \underline{\text{ident}} \mid \text{const}$

Example: \_\_\_\_\_

PROGRAM P;

VAR I : INTEGER;

VAR C : CHAR;

VAR J : INTEGER;

BEGIN I := 6; J := I; END.

## Abstract Grammar...

- Some items in the grammar are **attributes** (names of identifiers, e.g.) some are **children** (expression & statements in an if-statement, e.g.).
- Every child & attribute in the abstract grammar is given a name:

LOP: Expr.

- Example:

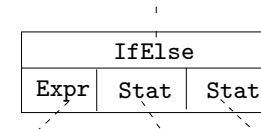
IfStat ::= Expr:Expr Then:Stat Else:Stat

## Grammar Example I...

- The rule

IfElse ::= Expr StatSeq StatSeq

says that an **if**-statement consists of three parts, or, equivalently, that an AST **if**-node will have three children:

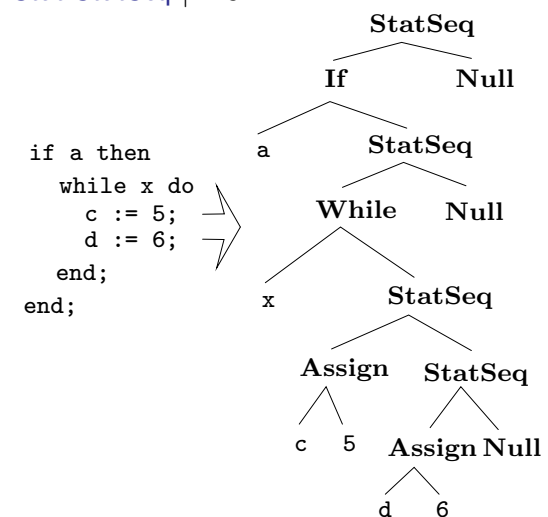


- We use recursive rules to define lists (e.g. declaration-lists, statement-lists):

StatSeq ::= Stat StatSeq | NULL

Stat ::= Assign | If | IfElse | While

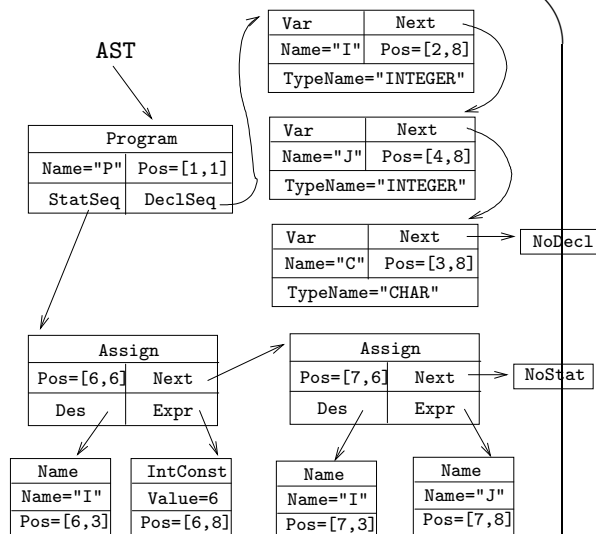
StatSeq ::= Stat StatSeq | NULL



**Name** ::=  $\Leftarrow$ Name:String  $\Leftarrow$ Pos:Position

**IntConst** ::=  $\Leftarrow$ Value:INTEGER  $\Leftarrow$ Pos:Position

```
PROGRAM P;
  VAR I : INTEGER;
  VAR J : INTEGER;
  VAR C : CHAR;
BEGIN
  I := 6;
  J := I;
END.
```



## Abstract Grammar...

- Input attributes** are data (e.g. identifiers, constants) created by the lexer/parser. I write them:

$\Leftarrow$ Name:String.

- Example:

**IntConst** ::=  $\Leftarrow$ Value:INTEGER  $\Leftarrow$ Pos:Position

- I prefer linked lists to recursion to define lists. A statement sequence are statements linked on a child **Next:StatSeq**. Lists end with an empty node: **NoDecl**.

## Grammar Example...

Abstract Grammar:

**Program** ::=  $\Leftarrow$ Name:String DeclSeq:Decl

StatSeq:StatSeq  $\Leftarrow$ Pos:Position

**Decl** ::= VarDecl | ProcDecl | ... | NoDecl

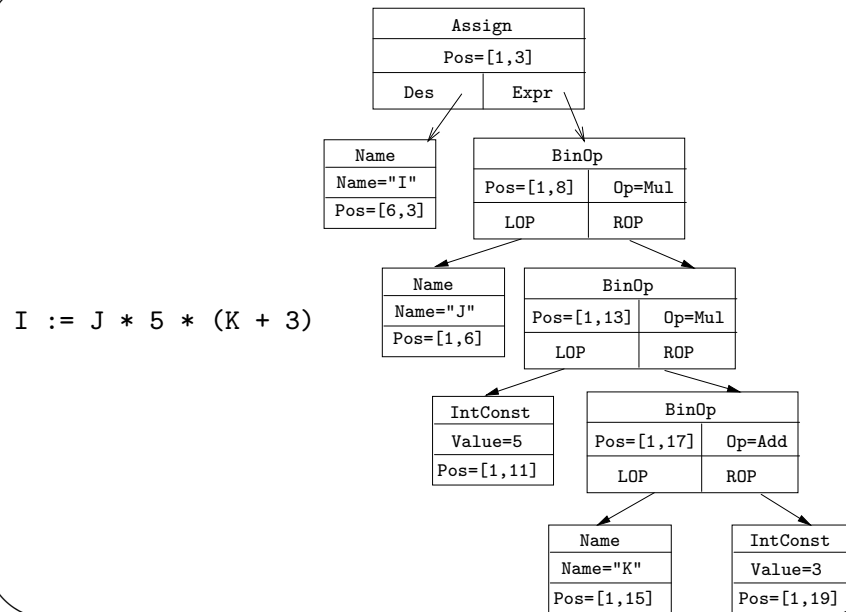
**VarDecl** ::=  $\Leftarrow$ Name:String  $\Leftarrow$ TypeName:String  $\Leftarrow$ Pos:Position  
Next:Decl

**Stat** ::= Assign | IfStat | ... | NoStat

**Assign** ::= Des:Name Expr:Expr  $\Leftarrow$ Pos:Position Next:Stat

**IfStat** ::= Expr:Expr Then:Stat Else:Stat  $\Leftarrow$ Pos:Position  
Next:Stat

**Expr** ::= Name | IntConst



### Grammar Example III

$\text{Assign} ::= \underline{\text{ident}} := \text{Expr}$

$\text{Expr} ::= \text{Expr} + \text{Term} \mid \text{Term}$

$\text{Term} ::= \text{Term} * \text{Factor} \mid \text{Factor}$

$\text{Factor} ::= \underline{(\text{Expr})} \mid \underline{\text{ident}} \mid \text{const}$

\_\_\_\_\_ Abstract Grammar (A): \_\_\_\_\_

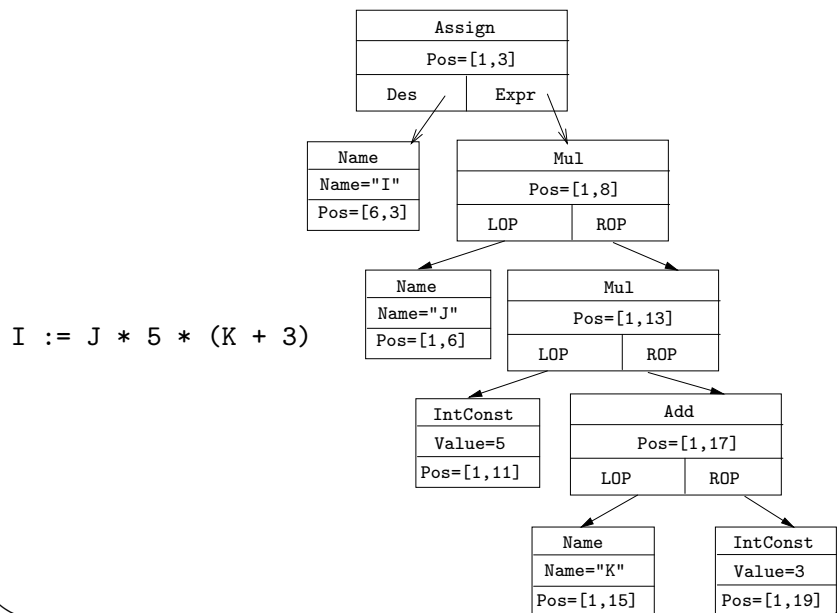
$\text{Assign} ::= \text{Des}:\text{Name} \text{ Expr}:\text{Expr} \Leftarrow \text{Pos}:\text{Position}$

$\text{Expr} ::= \text{BinOp} \mid \text{Name} \mid \text{IntConst}$

$\text{BinOp} ::= \text{LOP}:\text{Expr} \Leftarrow \text{Op}:(\text{Add}, \text{Mul}) \text{ ROP}:\text{Expr} \Leftarrow \text{Pos}:\text{Position}$

$\text{Name} ::= \Leftarrow \text{Name}:\text{String} \Leftarrow \text{Pos}:\text{Position}$

$\text{IntConst} ::= \Leftarrow \text{Value}:\text{INTEGER} \Leftarrow \text{Pos}:\text{Position}$



### Grammar Example III...

- There is often more than way to design the abstract grammar.
- We can turn attributes into node-kinds and vice versa.

\_\_\_\_\_ Abstract Grammar (B): \_\_\_\_\_

$\text{Assign} ::= \text{Des}:\text{Name} \text{ Expr}:\text{Expr} \Leftarrow \text{Pos}:\text{Position}$

$\text{Expr} ::= \text{Add} \mid \text{Mul} \mid \text{Name} \mid \text{IntConst}$

$\text{Add} ::= \text{LOP}:\text{Expr} \text{ ROP}:\text{Expr} \Leftarrow \text{Pos}:\text{Position}$

$\text{Mul} ::= \text{LOP}:\text{Expr} \text{ ROP}:\text{Expr} \Leftarrow \text{Pos}:\text{Position}$

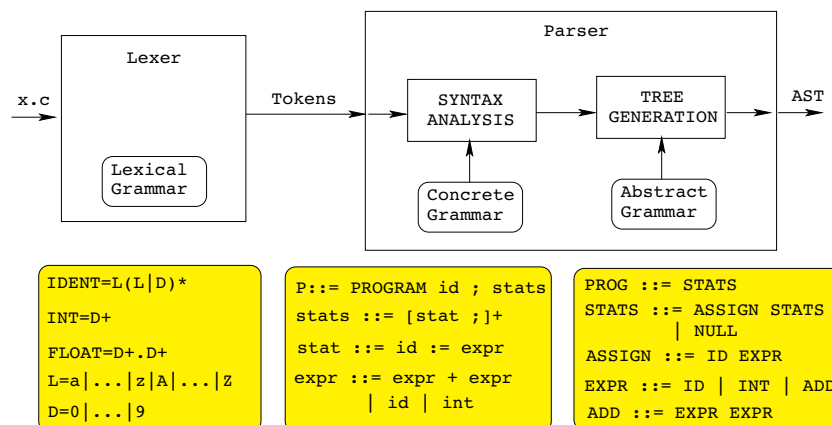
$\text{Name} ::= \Leftarrow \text{Name}:\text{String} \Leftarrow \text{Pos}:\text{Position}$

$\text{IntConst} ::= \Leftarrow \text{Value}:\text{INTEGER} \Leftarrow \text{Pos}:\text{Position}$

## The Chomsky Hierarchy...

- Regular languages are less powerful than context free languages.
- Languages are organized in **the Chomsky Hierarchy** according to their generative power.
- Type 3 languages are more restrictive (can describe simpler languages than) type 2 languages.
- Type 3 languages can be parsed in linear time, type 2 languages in cubic time.
- Programming languages are in between type 2 and 3.
- Two natural languages (Swiss German and Bambara) are known not to be context free.

## Compiler Grammars



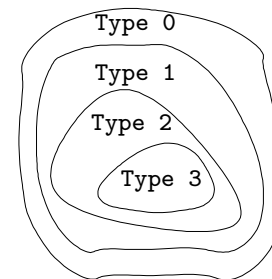
## Noam Chomsky

[www.geocities.com/Athens/Acropolis/5148/chomskybio.html](http://www.geocities.com/Athens/Acropolis/5148/chomskybio.html)

Linguist, social/political theorist; born in Philadelphia. Son of a distinguished Hebrew scholar, he was educated at the University of Pennsylvania, where he was especially influenced by Zellig Harris; after taking his M.A. there in 1951, he spent four years as a junior fellow at Harvard (1951–55), then was awarded a Ph.D. from the University of Pennsylvania (1955). In 1955 he began what would be his long teaching career at the Massachusetts Institute of Technology. He became known as one of the principal founders of transformational-generative grammar, a system of linguistic analysis that challenges much traditional linguistics and has much to do with philosophy, logic, and psycholinguistics; his book Syntactic Structures (1957) was credited with revolutionizing the discipline of linguistics.

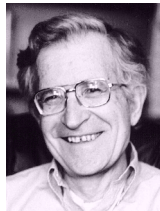
## The Chomsky Hierarchy

TYPE	GRAMMAR	PSR
0	Unrestricted	$\alpha \rightarrow \beta$
1	Context Sensitive	$\alpha \rightarrow \beta,$ $ \alpha  \leq  \beta $
2	Context Free	$A \rightarrow \beta$
3	Regular	$A \rightarrow a\beta$ $A \rightarrow a$



## Noam Chomsky...

- Chomsky vs B. F. Skinner: Famous debate in the late 50's, early 60's. Skinner was a **behaviorist**, believing that children learn language by imitating their parents. Chomsky refuted this, claiming that we all have innate language mechanisms.
- Nim Chimpsky was taught sign language in 1970s. It was a lost cause. He could ask for things, but not much more.



## Summary

- The job of a parser is to convert from concrete syntax to abstract syntax.
- We use context free grammars to describe both the concrete and the abstract syntax.
- The concrete syntax is described in the language manual of the language we're compiling.
- The abstract syntax we make up ourselves. There are many ways to define the abstract syntax of a language and personal preference will play a role in how we construct it.

## Noam Chomsky...

Chomsky's theory suggests that every human utterance has two structures: surface structure, the superficial combining of words, and "deep structure," which are universal rules and mechanisms. In more practical terms, the theory argues that the means for acquiring a language is innate in all humans and is triggered as soon as an infant begins to learn the basics of a language. Outside this highly rarefied sphere, Chomsky early on began to promote his radical critique of American political, social, and economic policies, particularly of American foreign policy as effected by the Establishment and presented by the media; he was outspoken in his opposition to the Vietnam War and later to the Persian Gulf War. His extensive writings in this area include *American Power* and the *New Mandarins* (1969) and *Human Rights and American Foreign Policy* (1978).

## Noam Chomsky...

- "If the Nüremberg laws were applied today, then every Post-War American president would have to be hanged."
- "The **corporatization of America** during the past century [has been] an attack on democracy."
- "Any dictator would admire the uniformity and obedience of the [U.S.] media."
- "Judged in terms of the power, range, novelty and influence of his thought, Noam Chomsky is arguably the most important intellectual alive." (The New York Times Book Review)
- Chomsky on terrorism:  
<http://www.zmag.org/GlobalWatch/chomskymit.htm>.

Slide 5-56

## Readings and References

- Read Louden:  
**Regular Expressions** 34-47.  
**Context-Free Grammars** 95-142.
- or the Dragon Book:  
**grammars** 165-171  
**associativity & precedence** 30-32  
**ambiguity** 171,174-175  
**derivations** 167-169  
**parse trees** 169-171  
**top-down parsing** 41-43  
**left recursion** 47-48

Slide 5-57

## Exam Problem

Use this abstract syntax to draw an AST for the TINY program below:

PROGRAM	→	STATSEQ
STATSEQ	→	STAT STATSEQ   NULL
STAT	→	ASSIGN   PRINT   DECL
DECL	→	ident type
ASSIGN	→	ident EXPR
PRINT	→	EXPR
EXPR	→	BINOP   IDENT   INTLIT
BINOP	→	op EXPR EXPR
IDENT	→	ident
INTLIT	→	int
FLTIT	→	float

BEGIN
INT x;
PRINT x + 9.9;
END