CSc 520

Principles of Programming Languages

51: Semantics — Syntax

Christian Collberg

`collberg@cs.arizona.edu`

Department of Computer Science

University of Arizona

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.

Production Rules

- Here's a production rule:

$$\mathrm{IfStat} \longrightarrow \underline{\texttt{if}}\ \underline{(}\ \mathrm{expr}\ \underline{)}\ \mathrm{stat}$$

- 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.

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

A Simple English Grammar

S → NP VP
VP → V NP
VP → V
NP → N
NP → Det N

N → `John`
N → `Lisa`
N → `house`
V → `died`
V → `kissed`
Det → `the`
Det → `a`

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

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 γ.
5. Repeat until left with a string of words.

$$\begin{aligned}
S &\overset{\text{S}\rightarrow\text{NP VP}}{\Longrightarrow} \text{NP VP}\\
&\overset{\text{NP}\rightarrow\text{N}}{\Longrightarrow} \text{N VP}\\
&\overset{\text{N}\rightarrow\texttt{John}}{\Longrightarrow} \texttt{John}\ \text{VP}\\
&\overset{\text{VP}\rightarrow\text{V NP}}{\Longrightarrow} \texttt{John}\ \text{V NP}\\
&\overset{\text{V}\rightarrow\texttt{kissed}}{\Longrightarrow} \texttt{John kissed}\ \text{NP}\\
&\overset{\text{NP}\rightarrow\text{N}}{\Longrightarrow} \texttt{John kissed}\ \text{N}\\
&\overset{\text{N}\rightarrow\texttt{Lisa}}{\Longrightarrow} \texttt{John kissed Lisa}
\end{aligned}$$

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 α, β are taken from $N \bigcup \Sigma$.
- Read $\alpha \rightarrow \beta$ as "rewrite α with β".
- Read $\Rightarrow$ as "directly derives".
- Read $\overset{r}{\Rightarrow}$ as "directly derives using rule r".
- Read $\overset{*}{\Rightarrow}$ as "derives in one or more steps".

A Simple PL Grammar

Here's a grammar for a simple programming language:

Program ::= `BEGIN` Stat `END`
Stat ::= `ident` `:=` Expr
Expr ::= Expr `+` Expr |
Expr `*` Expr |
`ident` | `number`

- We write terminal symbols like `this`.
- We write non-terminal symbols like this.
- Sometimes we write `::=` instead of →.
- $A \to b \mid c$ is the same as $A \to b; A \to c$. Read | as "or".

A Simple PL Grammar…

We know the sentence

`BEGIN a := 5 + 4 * 3 END`

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

Program ⇒ `BEGIN` Stat `END`
⇒ `BEGIN` `ident` `:=` Expr `END`
⇒ `BEGIN` "a" `:=` Expr `END`
⇒ `BEGIN` "a" `:=` Expr `+` Expr `END`
⇒ `BEGIN` "a" `:=` 5 `+` Expr `END`
⇒ `BEGIN` "a" `:=` 5 `+` Expr `*` `Expr` `END`
⇒ `BEGIN` "a" `:=` 5 `+` 4 `*` Expr `END`
⇒ `BEGIN` "a" `:=` 5 `+` 4 `*` 3 `END`

Terminology…

- Our English grammar is the 4-tuple
  ({S,NP,V,…},
  {`John`,`house`,`died`,…},
  {S → NP VP, VP → V,…},
  S)
- Our PL grammar is the 4-tuple
  ({Program,Stat,…},
  {`BEGIN`,`:=`,`*`,…},
  { Program `::=` `BEGIN` Stat `END`,…},
  Program)

Parse Trees

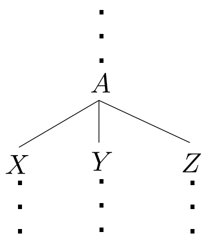
- We often want to show how a particular sentence was derived. We can do this without listing all the steps explicitly by drawing a parse tree.
- A parse tree is a tree where
  1. The root is labeled by the start symbol.
  2. Each leaf is labeled by a terminal symbol.
  3. Each interior node is labeled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

Parse Trees…

- If one step of our derivation is

$$\cdots\ A\ \cdots\ \Rightarrow\ \cdots\ X\ Y\ Z\ \cdots$$

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



Parse Trees…

$S \xRightarrow{S\rightarrow NP\ VP}$ NP VP

$\xRightarrow{NP\rightarrow N}$ N VP

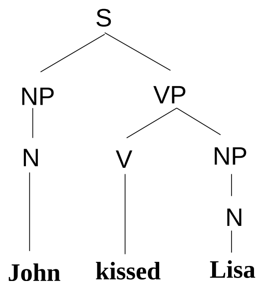
$\xRightarrow{N\rightarrow \texttt{John}}$ `John` VP

$\xRightarrow{VP\rightarrow V\ NP}$ `John` V NP

$\xRightarrow{V\rightarrow \texttt{kissed}}$ `John kissed` NP

$\xRightarrow{NP\rightarrow N}$ `John kissed` N

$\xRightarrow{N\rightarrow \texttt{Lisa}}$ `John kissed Lisa`



Parse Trees…

gram ⇒ BEGIN Stat END

⇒ BEGIN ident := Expr END

⇒ BEGIN "a" := Expr END

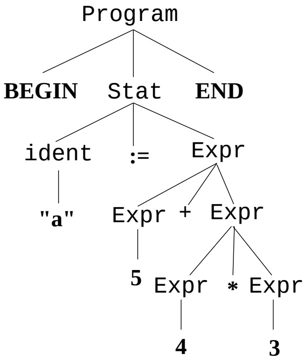
⇒ BEGIN "a" := Expr + Expr END

⇒ BEGIN "a" := 5 + Expr END

⇒ BEGIN "a" := 5 + Expr * Expr END

⇒ BEGIN "a" := 5 + 4 * Expr END

⇒ BEGIN "a" := 5 + 4 * 3 END



Regular Grammars

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

$$\begin{aligned} A &\rightarrow aB \\ A &\rightarrow a \end{aligned}$$

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

Context-Free Grammars

- Programming language syntax is described by a context free grammar (CFG).
- In a CFG all rules are of the form

  $$A \rightarrow \gamma$$

  γ 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...

- In BNF, our variable declaration

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

  looks like this:

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

  idlist ::= `,` `ident` idlist | ϵ
- 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` | ϵ
- In EBNF, it looks like this:

  class ::= `class` `ident` [`extends` `ident`] `{` ... `}`
- I.e. $[e]$ means that e is optional.

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; }
```

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
```

EBNF for Luca. . .

```
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
```

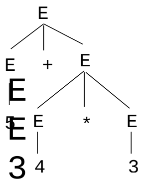
Ambiguous Grammars

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

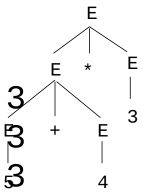
$$\mathtt{E} ::= \mathtt{E}\ \underline{+}\ \mathtt{E} \mid \mathtt{E}\ \underline{*}\ \mathtt{E} \mid \underline{\mathtt{number}}$$

$$\boxed{5\ +\ 4\ *\ 3}$$

$$\begin{aligned} E &\Rightarrow E\ \underline{+}\ E \\ &\Rightarrow 5\ \underline{+}\ E \\ &\Rightarrow 5\ \underline{+}\ E\ \underline{*}\ E \\ &\Rightarrow 5\ \underline{+}\ 4\ \underline{*}\ E \\ &\Rightarrow 5\ \underline{+}\ 4\ \underline{*}\ 3 \end{aligned}$$

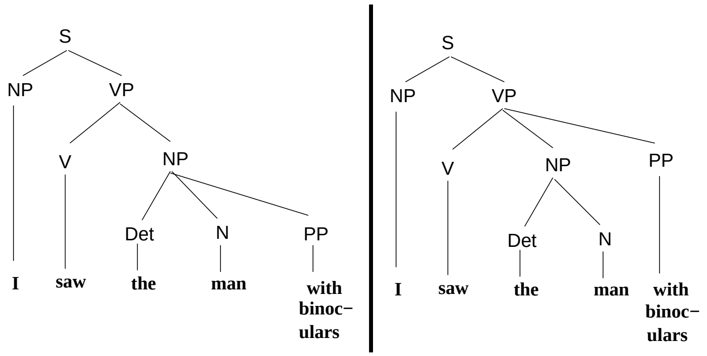


$$\begin{aligned} E &\Rightarrow E\ \underline{*}\ E \\ &\Rightarrow E\ \underline{*}\ 3 \\ &\Rightarrow E\ \underline{+}\ E\ \underline{*}\ 3 \\ &\Rightarrow E\ \underline{+}\ 4\ \underline{*}\ 3 \\ &\Rightarrow 5\ \underline{+}\ 4\ \underline{*}\ 3 \end{aligned}$$



Structural Ambiguity in English

- Ambiguities occur in natural languages also:



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 $\underline{*}$ binds harder than $\underline{+}$.

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.

- ˆ associates to the right:

$$2\,\hat{}\,3\,\hat{}\,4 = 2\,\hat{}\,(3\,\hat{}\,4).$$

Operators in C

OPERATOR	KIND	PREC	ASSOC
`a[k]`	Primary	16	
`f(···)`	Primary	16	
`.`	Primary	16	
`->`	Primary	16	
`a++, a--`	Postfix	15	
`++a, --a`	Unary	14	
`~`	Unary	14	
`!`	Unary	14	
`-`	Unary	14	
`&`	Unary	14	
`*`	Unary	14	

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:

expr ::= expr + term | term

term ::= term * factor | factor

factor ::= (expr) | `number`

Expression Grammars. . .

1. Create one non-terminal for each precedence level, for example $p_1, p_2, \cdots, p_n$, 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 op p_{i+1} | p_{i+1}
   - right associative:

     p_i ::= p_{i+1} op p_i | 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) | num | id

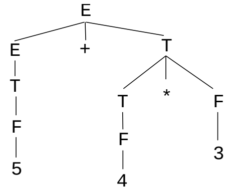
Expression Grammars. . .

E ::= E + T | T

T ::= T * F | F

F ::= `number`

`5 + 4 * 3`



E ⇒ E + T
⇒ T + T
⇒ F + T
⇒ 5 + T
⇒ 5 + T * F
⇒ 5 + F * F
⇒ 5 + 4 * F
⇒ 5 + 4 * 3

E ⇒ E + T
⇒ E + T * F
⇒ E + T * 3
⇒ E + F * 3
⇒ E + 4 * 3
⇒ T + 4 * 3
⇒ F + 4 * 3
⇒ 5 + 4 * 3

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).

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.

Grammar Example I

Concrete Grammar:

S ::= `ident` `:=` E |
`if` E `then` SS_1 [`else` SS_2] `end` | `while` E `do` SS `end` | ϵ
SS ::= S `;` SS | ϵ

Abstract Grammar:

Assign ::= `ident` Expr

If ::= Expr StatSeq

IfElse ::= Expr StatSeq StatSeq

While ::= Expr StatSeq

Stat ::= Assign | If | IfElse | While

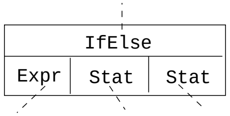
StatSeq ::= Stat StatSeq | NULL

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):

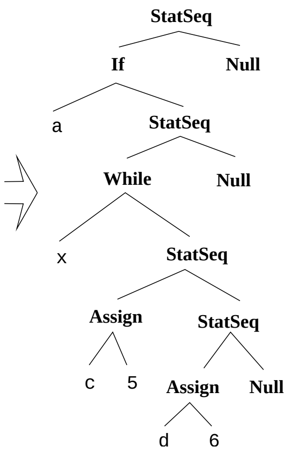
  StatSeq ::= Stat StatSeq | NULL

Grammar Example I…

Stat ::= Assign | If | IfElse | While

StatSeq ::= Stat StatSeq | NULL

```
if a then
   while x do
     c := 5;
     d := 6;
   end;
end;
```



Concrete Grammar Example II

Program ::= `program` `ident` `;` DeclSeq `begin` StatSeq `end` `.`

DeclSeq ::= Decl `;` DeclSeq | ϵ

Decl ::= `var` `ident` `:` `ident`

Stat ::= `ident` `:=` Expr | `if` Expr `then` StatSeq `else` StatSeq

StatSeq ::= Stat `;` StatSeq | ϵ

Expr ::= `ident` | 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

Abstract Grammar…

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

  ⇐`Name`:**String**.

- Example:

  IntConst ::= ⇐`Value`:**INTEGER** ⇐`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 ::= ⇐Name:**String** DeclSeq:Decl
StatSeq:StatSeq ⇐Pos:**Position**

Decl ::= VarDecl | ProcDecl | · · · | NoDecl

VarDecl ::= ⇐Name:**String** ⇐TypeName:**String** ⇐Pos:**Position**
Next:Decl

Stat ::= Assign | IfStat | · · · | NoStat

Assign ::= Des:Name Expr:Expr ⇐Pos:**Position** Next:Stat

IfStat ::= Expr:Expr Then:Stat Else:Stat ⇐Pos:**Position**
Next:Stat

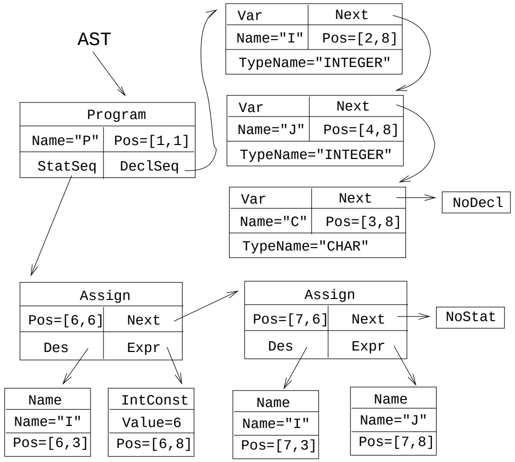
Expr ::= Name | IntConst

Name ::= ⇐Name:**String** ⇐Pos:**Position**

IntConst ::= ⇐Value:**INTEGER** ⇐Pos:**Position**

Grammar Example. . .

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



Grammar Example III

Assign ::= `ident` `:=` Expr

Expr ::= Expr + Term | Term

Term ::= Term * Factor | Factor

Factor ::= `(` Expr `)` | `ident` | const

Abstract Grammar (A):

Assign ::= Des:Name Expr:Expr ⇐Pos:**Position**

Expr ::= BinOp | Name | IntConst

BinOp ::= LOP:Expr ⇐Op:**(Add,Mul)** ROP:Expr ⇐Pos:**Position**

Name ::= ⇐Name:**String** ⇐Pos:**Position**

IntConst ::= ⇐Value:**INTEGER** ⇐Pos:**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):

Assign ::= Des:Name Expr:Expr ⇐Pos:**Position**

Expr ::= Add | Mul | Name | IntConst

Add ::= LOP:Expr ROP:Expr ⇐Pos:**Position**

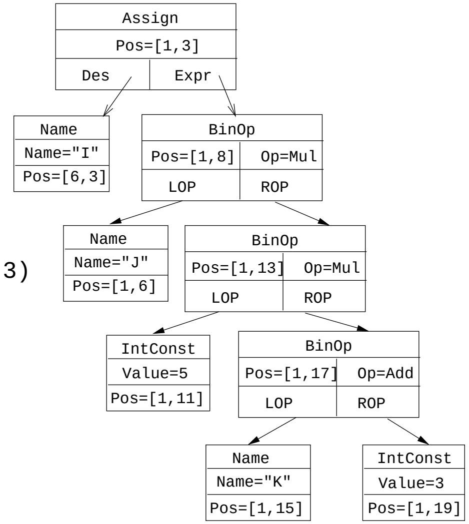
Mul ::= LOP:Expr ROP:Expr ⇐Pos:**Position**

Name ::= ⇐Name:**String** ⇐Pos:**Position**

IntConst ::= ⇐Value:**INTEGER** ⇐Pos:**Position**

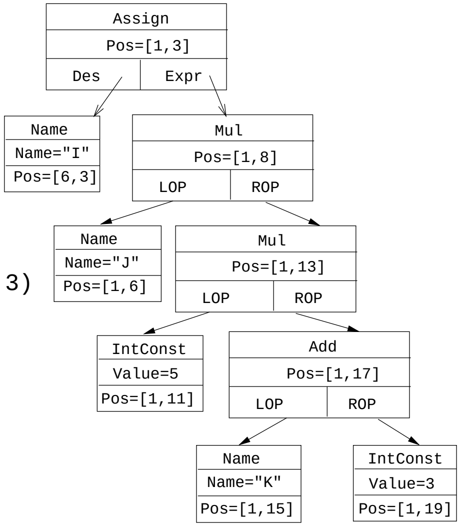
Grammar Example III. . .

I := J * 5 * (K + 3)

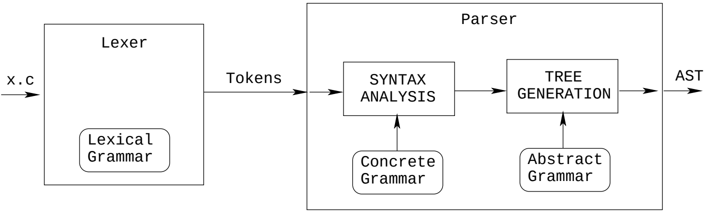


Grammar Example III. . .

I := J * 5 * (K + 3)



Compiler Grammars



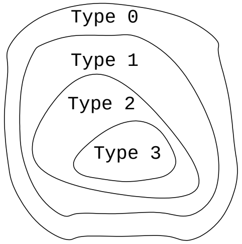
```
IDENT=L(L|D)*
INT=D+
FLOAT=D+.D+
L=a|...|z|A|...|Z
D=0|...|9
```

```
P::= PROGRAM id ; stats
stats ::= [stat ;]+
stat ::= id := expr
expr ::= expr + expr
       | id | int
```

```
PROG ::= STATS
STATS ::= ASSIGN STATS
        | NULL
ASSIGN ::= ID EXPR
EXPR ::= ID | INT | ADD
ADD ::= EXPR EXPR
```

The Chomsky Hierarchy

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



The Chomsky Hierarchy…

- Regular languages are less powerful than context free languages.
- Languages are organized in the Chomsky Hieararchy 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.

Noam Chomsky

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.

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`.

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.

Readings and References

- Read Scott, Chapter 2: Programming Language Syntax
- 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

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
FLTLIT → float

```
BEGIN
   INT x;
   PRINT x + 9.9;
END
```