CS321 Week 4: Context-Free Grammars

Jingke Li, Portland State University

Winter 2014

Today's Topics

- Context-free grammars
- Derivations and parse trees
- Ambiguity and recursions
- Equivalence and transformations

Next Two Weeks:

- Top-down parsing
- Bottom-up parsing

Language Specifications

To process a language, we need specifications for all its components: words, sentence structures, semantics, etc.

While specifications can come in many different forms, to compilers, we want the specifications be both *precise* and *concise*.

Formal systems for programming language specifications:

- lexical tokens — regular expressions
- syntax structures — context-free grammars
- semantics — denotation, abstract machine, or logic axioms

Formal Grammars

> Grammar is a system of rules that defines the syntactical structure of a language. (*Merriam-Webster.com*)

A *formal grammar* is used to define the syntactical structure of a formal language, It consists of the following components:

- A finite set of *terminals* — symbols that form sentences.
- A finite set of *nonterminals* — variables that can be extended to sentences by the grammar rules.
  - One of the nonterminals is designated as the *start symbol*.
- A finite set of *productions* — grammar rules of the form "lhs $\rightarrow$ rhs", where lhs and rhs are sequences of terminals and nonterminals, with lhs containing at least one nonterminal. The productions specify the relationship between terminals and nonterminals, and how they should be combined to form sentences.

Chomsky Hierarchy

- *Unrestricted* Grammars: $\alpha A \beta \rightarrow \gamma$
  - Generate *recursively enumerable* languages
- *Context-Sensitive* Grammars: $\alpha A \beta \rightarrow \alpha \gamma \beta$
  - Generate *context-sensitive* languages
- *Context-Free* Grammars: $A \rightarrow \gamma$
  - Generate *context-free* languages (which include most programming languages)
- *Regular* Grammars: $A \rightarrow a$ or $A \rightarrow aB$
  - Generate *regular* languages, i.e. regular expressions

Context-Free Grammars (CFGs)

A CFG is defined by

- A finite set of terminals V_t
  - For a PL, terminals are just tokens recognized by the lexer
- A finite set of nonterminals V_n, including a start symbol S
- A finite set of productions of the form
  $A \rightarrow x_1 \cdots x_m$ where $A \in V_n,\ x_i \in V_n \cup V_t$

Backus-Naur Form (BNF):

BNF is a particular notation for CFGs. It was first used in the formal description of Algol 60. The notation convention has evolved through the years; current convention include:

- Nonterminals begin with upper-case letters.
- Terminals are quoted. However, terminals that do not correspond to literal characters or strings can be represented as they are.

Extended BNF

EBNF is *any* extension of BNF, usually with the following features:

- A vertical bar, |, represents a choice,
- Parentheses, (and), represent grouping,
- Square brackets, [and], represent an optional construct,
- Curly braces, { and }, represent zero or more repetitions,

EBNF does not have extra power than BNF. In fact, any grammar in EBNF can be transformed into one in proper BNF.

An Original (Algol 60-Style) BNF Example

⟨program⟩ ::= begin ⟨statement list⟩ end
⟨statement list⟩ ::= ⟨statement⟩ ⟨statement list⟩
⟨statement list⟩ ::= ⟨statement⟩
⟨statement⟩ ::= ⟨assignment⟩
⟨statement⟩ ::= ⟨read statement⟩
⟨statement⟩ ::= ⟨write statement⟩
⟨assignment⟩ ::= ID := ⟨expression⟩ ;
⟨read statement⟩ ::= read (⟨id list⟩) ;
⟨write statement⟩ ::= write (⟨expr list⟩) ;
⟨id list⟩ ::= ID , ⟨id list⟩
⟨id list⟩ ::= ID
⟨expr list⟩ ::= ⟨expression⟩ , ⟨expr list⟩
⟨expr list⟩ ::= ⟨expression⟩
⟨expression⟩ ::= ⟨primary⟩ + ⟨expression⟩
⟨expression⟩ ::= ⟨primary⟩ − ⟨expression⟩
⟨expression⟩ ::= ⟨primary⟩
⟨primary⟩ ::= (⟨expression⟩)
⟨primary⟩ ::= ID
⟨primary⟩ ::= INTEGER

A Corresponding Modern Version

Program → "begin" StmtList "end"
StmtList → Statement StmtList
StmtList → Statement
Statement → Assignment
Statement → ReadStmt
Statement → WriteStmt
Assignment → id ":=" Expression ";"
ReadStmt → "read" "(" IdList ")" ";"
WriteStmt → "write" "(" ExprList ")" ";"
IdList → id "," IdList
IdList → id
ExprList → Expression "," ExprList
ExprList → Expression
Expression → Primary "+" Expression
Expression → Primary "−" Expression
Expression → Primary
Primary → "(" Expression ")"
Primary → id
Primary → integer

A Corresponding EBNF

Program → "begin" StmtList "end"
StmtList → Statement { Statement }
Statement → Assignment | ReadStmt | WriteStmt
Assignment → id ":=" Expression ";"
ReadStmt → "read" "(" IdList ")" ";"
WriteStmt → "write" "(" ExprList ")" ";"
IdList → id { "," id }
ExprList → Expression { "," Expression }
Expression → Primary { ("+" | "−") Primary }
Primary → "(" Expression ")" | id | integer

Transforming EBNF to BNF

Rules:

$A \to \alpha\,[\,\beta\,]\,\gamma \quad \Rightarrow$
$A \to \alpha B \gamma$
$B \to \beta$
$B \to \epsilon$

$A \to \alpha\,\{\beta\}\,\gamma \quad \Rightarrow$
$A \to \alpha B \gamma$
$B \to \beta B$
$B \to \epsilon$

Example:

IdList → id { "," id } ⇒
IdList → id IdTail
IdTail → "," id IdTail
IdTail → ϵ

Language Defined by a Grammar

The set of sentences *derived* from the start symbol S of a grammar G by repeatedly applying the production rules defines the language of the grammar, denoted $L(G)$: $L(G) = \{x \in V_t^* \mid S \stackrel{+}{\Rightarrow} x\}$.

A *derivation* step is to replace the lhs of a production by its rhs.

Example:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * P$
(4) $T \to P$
(5) $P \to \text{id}$

$E \stackrel{(1)}{\Rightarrow} E + T \stackrel{(2)}{\Rightarrow} T + T$
$\stackrel{(3)}{\Rightarrow} T * P + T \stackrel{(4)}{\Rightarrow} P * P + T$
$\stackrel{(5)}{\Rightarrow} \text{id} * P + T \stackrel{(5)}{\Rightarrow} \text{id} * \text{id} + T$
$\stackrel{(4)}{\Rightarrow} \text{id} * \text{id} + P \stackrel{(5)}{\Rightarrow} \text{id} * \text{id} + \text{id}$

Notations:

$\Rightarrow$ one derivation step
$\stackrel{+}{\Rightarrow}$ one or more steps
$\stackrel{*}{\Rightarrow}$ zero or more steps

Note: When context is clear, we omit quotes around terminal symbols in our examples.

Derivations

A sentence can be derived by different sequences of productions.

Example (Revisit):

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * P$
(4) $T \to P$
(5) $P \to \text{id}$

$$E \overset{(1)}{\Rightarrow} E + T \overset{(2)}{\Rightarrow} T + T \overset{(3)}{\Rightarrow} T * P + T \overset{(4)}{\Rightarrow} P * P + T \overset{(5)}{\Rightarrow} \text{id} * P + T \overset{(5)}{\Rightarrow} \text{id} * \text{id} + T \overset{(4)}{\Rightarrow} \text{id} * \text{id} + P \overset{(5)}{\Rightarrow} \text{id} * \text{id} + \text{id}$$

Two alternative derivation sequences:

$$E \overset{(1)}{\Rightarrow} E + T \overset{(2)}{\Rightarrow} T + T \overset{(3)}{\Rightarrow} T * P + T \overset{(4)}{\Rightarrow} P * P + T \overset{(4)}{\Rightarrow} P * P + P \overset{(5)}{\Rightarrow} P * P + \text{id} \overset{(5)}{\Rightarrow} P * \text{id} + \text{id} \overset{(5)}{\Rightarrow} \text{id} * \text{id} + \text{id}$$

$$E \overset{(1)}{\Rightarrow} E + T \overset{(4)}{\Rightarrow} E + P \overset{(5)}{\Rightarrow} E + \text{id} \overset{(2)}{\Rightarrow} T + \text{id} \overset{(3)}{\Rightarrow} T * P + \text{id} \overset{(4)}{\Rightarrow} P * P + \text{id} \overset{(5)}{\Rightarrow} \text{id} * P + \text{id} \overset{(5)}{\Rightarrow} \text{id} * \text{id} + \text{id}$$

Derivations of Compiler Interests

- *Left-Most Derivation* ($\Rightarrow_{lm}$, $\overset{+}{\Rightarrow}_{lm}$, $\overset{*}{\Rightarrow}_{lm}$): Always choose the left-most nonterminal to expand.
- *Right-Most Derivation* ($\Rightarrow_{rm}$, $\overset{+}{\Rightarrow}_{rm}$, $\overset{*}{\Rightarrow}_{rm}$): Always choose the right-most nonterminal to expand.

Example:

$S \to S\ ;\ S \mid \text{id} := E \mid \text{print}\ (\ E\ )$
$E \to \text{id} \mid \text{num}$

Sentence: id := num ; print (id)

$$\underline{S} \Rightarrow_{lm} \underline{S}\ ;\ S \Rightarrow_{lm} \text{id} := \underline{E}\ ;\ S \Rightarrow_{lm} \text{id} := \text{num}\ ;\ \underline{S} \Rightarrow_{lm} \text{id} := \text{num}\ ;\ \text{print}\ (\ \underline{E}\ ) \Rightarrow_{lm} \text{id} := \text{num}\ ;\ \text{print}\ (\ \text{id}\ )$$

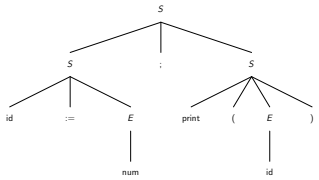
$$\underline{S} \Rightarrow_{rm} S\ ;\ \underline{S} \Rightarrow_{rm} S\ ;\ \text{print}\ (\ \underline{E}\ ) \Rightarrow_{rm} \underline{S}\ ;\ \text{print}\ (\ \text{id}\ ) \Rightarrow_{rm} \text{id} := \underline{E}\ ;\ \text{print}\ (\ \text{id}\ ) \Rightarrow_{rm} \text{id} := \text{num}\ ;\ \text{print}\ (\ \text{id}\ )$$

Parse Tree

A tree representation of the derivation from the start symbol to a sentence, *a.k.a. concrete syntax tree*.

- *root* — the start symbol
- *internal nodes* — nonterminals
- *leaf nodes* — terminals

Both the left-most derivation and the right-most derivation are linear representations of a parse tree.



Ambiguity in Grammars

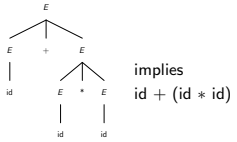
Sometimes, multiple parse trees can be derived from a grammar for the same sentence.

Example 1: Arithmetic expressions

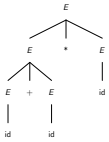
Grammar: $E \to E + E \mid E * E \mid (\ E\ ) \mid - E \mid \text{id}$

Sentence: id + id * id

Parse Trees:



implies id + (id * id)



implies (id + id) * id

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Parsers can not be built with an ambiguous grammar!

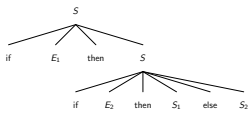
Ambiguity in Grammars (cont.)

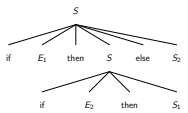
Example 2: The "Dangling Else"

Grammar: $S \to$ if E then $S \mid$ if E then S else $S \mid$ other

Sentence: if E_1 then if $E2$ then $S1$ else $S2$

Parse Trees:





Implied semantics:

if E_1 then { if $E2$ then $S1$ else $S2$ } if E_1 then { if E_2 then S_1 } else S_2

Recursion in Grammars

Recursion in productions is necessary for any non-trivial CFGs. (*Why?*)

Recursions can appear in different positions in a production:

$P \to a\,P\,b \mid P\,c\,d \mid e\,P \mid f$

$S \to A\,a \mid b$
$A \to A\,c \mid S\,d \mid e$

Of Compiler Interests:

- *Left-recursion:* $E \overset{+}{\Rightarrow} E\alpha$ for some string α.
  - *Immediate form:*: $E \to E + \text{id}$
  - *General form:* $E \to F$, $F \to E + \text{id}$
- *Right-recursion:* $E \overset{+}{\Rightarrow} \alpha E$ for some string α.
  - *Immediate form:*: $E \to \text{id} + E$
  - *General form:* $E \to G$, $G \to \text{id} + E$

Grammars with left-recursive productions are not suitable for the predictive parsing method (*i.e.* top-down parsing).

Equivalence of Grammars

As shown earlier, each grammar G defines a single language $L(G)$.

However, the reverse is not true. For a given language L, there can be multiple grammars that define it. I.e., we can have $G_1, G_2, G_3, \cdots$ that

$$L = L(G_1) = L(G_2) = L(G_3) = \cdots.$$

Example 1:

G_1: $S \rightarrow S\ \mathrm{x} \mid \epsilon$

G_2: $S \rightarrow \mathrm{x}\ S \mid \epsilon$

Both define the language x^*.

Example 2:

G_1: $E \rightarrow E + E \mid E - E \mid$ id (*ambiguous!*)

G_2: $E \rightarrow E + id \mid E - id \mid$ id

Both define the set of binary expressions with $+$ and $-$.

Grammar Transformations

For writing parsers, it is often necessary to transform a given grammar to an equivalent, but more parser-friendly grammar.

Examples:

- Elimination of ambiguity
- Elimination of left recursion
- Left factoring

Elimination of Ambiguity

Approach 1: Rewrite grammar with operator precedence.

Example:

Precedence (from high to low): unary $-$, $*$, $+$.

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow - E$
4. $E \rightarrow (E)$
5. $E \rightarrow$ id

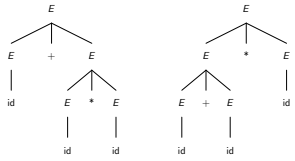
$\Rightarrow$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow - P$
6. $P \rightarrow (E)$
7. $P \rightarrow$ id

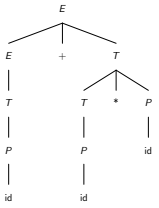
Operator Precedence Example (cont.)

Sentence: id $+$ id $*$ id

Before: multiple parse trees



After: unique parse tree



Eliminating Ambiguity (cont.)

Approach 2: Rewrite grammar with more syntactic constraints.

Example:

$S \rightarrow$ if E then $S \mid$ if E then S else $S \mid$ *other*

$\Rightarrow$

$S \rightarrow m\_S \mid unm\_S$

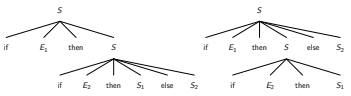
$m\_S \rightarrow$ if E then m_S else $m\_S \mid$ *other*

$unm\_S \rightarrow$ if E then $S \mid$ if E then m_S else unm_S

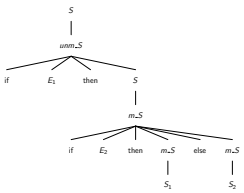
Syntactic Constraints Example (cont.)

Sentence: if E_1 then if E_2 then S_1 else S_2

Before: multiple parse trees



After: unique parse tree



Syntactic Constraints Example (cont.)

Details are important. If we change the transformation slightly:

$S \rightarrow$ if E then S | if E then S else S | *other*

$\Rightarrow$

$S \rightarrow m\_S \mid unm\_S$
$m\_S \rightarrow$ if E then m_S else $S \mid$ *other*
$unm\_S \rightarrow$ if E then S

then the resulting grammar is still ambiguous!

Elimination of Left Recursions

Case 1. Simple lists.

Approach: Convert directly.

- $A \rightarrow A\alpha \mid \epsilon$

  Sentence patterns: $L = \alpha*$

  Equivalent productions: $A \rightarrow \alpha A \mid \epsilon$

  With EBNF operators: $A \rightarrow \{\, \alpha \,\}$

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

  Sentence patterns: $L = \beta\alpha*$

  Equivalent productions: $A \rightarrow \beta A'$
  $A' \rightarrow \alpha A' \mid \epsilon$

  With EBNF operators: $A \rightarrow \beta\, \{\, \alpha \,\}$

Elimination of Left Recursions (cont.)

Case 2. Immediate left recursions.

Approach: Collect all productions for a given nonterminal; group them into left-recursive ones, and non-left-recursive ones.

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

  Grouping and factoring:

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

  Rewriting: $A \rightarrow A\,B \mid C$
  $B \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_m$
  $C \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

  Sentence patterns: $L = C\,B*$

  Equivalent productions:

  $A \rightarrow (\beta_1 \mid \cdots \mid \beta_n)A'$
  $A' \rightarrow (\alpha_1 \mid \cdots \mid \alpha_m)A' \mid \epsilon$

Elimination of Left Recursions (cont.)

Case 3. General non-immediate left recursions.

Approach: Inline productions to turn general non-immediate left recursions into immediate left recursions; then apply the replacement transformation.

- $A \rightarrow B\,\alpha_1 \mid C\,\alpha_2 \mid \alpha_3$
  $B \rightarrow A\,\beta_1 \mid \beta_2$
  $C \rightarrow A\,\gamma_1 \mid \gamma_2$

  Inlining: $A \rightarrow A\,\beta_1\alpha_1 \mid \beta_2\alpha_1 \mid A\,\gamma_1\alpha_2 \mid \gamma_2\alpha_2 \mid \alpha_3$

Example:

$E \rightarrow F \mid T$
$F \rightarrow E + x \mid E - y \mid P$

$\Rightarrow$

$E \rightarrow E + x \mid E - y \mid P \mid T$
$F \rightarrow E + x \mid E - y \mid P$

$\Rightarrow$

$E \rightarrow P\,E' \mid T\,E'$
$E' \rightarrow +\,x\,E' \mid -\,y\,E' \mid \epsilon$
$F \rightarrow E + x \mid E - y \mid P$

Left Factoring

Productions with common prefix cause problems for predictive parsers:

$S \rightarrow$ if E then S end if ;
$S \rightarrow$ if E then S else S end if ;

An easy fix is to perform left factoring:

$S \rightarrow$ if E then $S\ T$
$T \rightarrow$ end if ;
$T \rightarrow$ else S end if ;

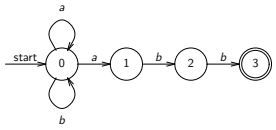
When the common prefix is not explicit, it is hard to fix.

CFGs vs. Regular Expressions

Every set that can be described by a regular expression can also be described by a context-free grammar.

Example: the R.E. $(a|b)^*abb$ can be described by

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \epsilon$



Regular Grammars are just context-free grammars with their productions limited to the following two forms,

$A \rightarrow aB$
$C \rightarrow \epsilon$

Chomsky Hierarchy Revisit

The Chomsky hierarchies are properly nested, *i.e.*

Regular languages $\subset$ CFG languages $\subset$ CSG languages $\subset$ R.E. languages

Examples:

- The language $\{ [^i]^i \mid i \geq 1\}$ is not regular, but it can be defined by a context-free grammar:

  $S \rightarrow [\ T\ ]$
  $T \rightarrow [\ T\ ] \mid \epsilon$

- The following language is not describable by CFG:

  $L = \{wcw \mid w \text{ is in } (a|b)*\}$

  In contrast, the following is a CFG:

  $L' = \{wcw^R \mid w \text{ is in } (a|b)*\}$