

Context-Free Grammars

©SoftMoore Consulting

Slide 1

1

Specification of a Programming Language

- **Syntax (form)**
 - basic language symbols (or tokens)
 - allowed structure of symbols to form programs
 - specified by a context-free grammar
- **Contextual Constraints**
 - program rules and restrictions that can not be specified in a context-free grammar
 - consist primarily of type and scope rules
- **Semantics (meaning)**
 - behavior of program when it is run on a machine
 - usually specified informally

©SoftMoore Consulting

Slide 2

2

Formal Versus Informal Specification

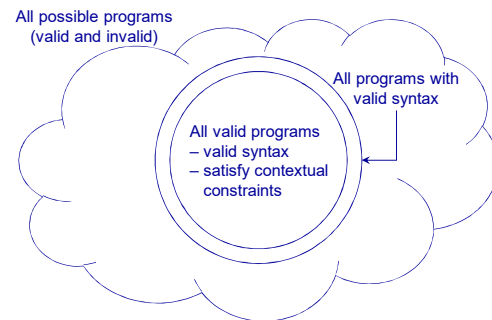
- **Informal specification**
 - usually written in English (or some other natural language)
 - easily understood
 - difficult to make sufficiently precise
- **Formal specification**
 - precise notation
 - effort required to understand
- **Common practice**
 - syntax: formal specification (context-free grammar)
 - contextual constraints: informal specification
 - semantics: informal specification

©SoftMoore Consulting

Slide 3

3

Syntax Versus Contextual Constraints



©SoftMoore Consulting

Slide 4

4

Syntax Versus Contextual Constraints (continued)

- In general, each programming language is “approximated” by a context-free grammar. Contextual constraints further restrict the approximating language to coincide with the desired language.
- Examples: Valid syntax but not valid with respect to contextual constraints

```

var x : Integer;      var c : Char;
begin                 begin
  y := 5;              c := -3;
end.                  end.
  
```

©SoftMoore Consulting

Slide 5

5

Context-Free Grammars

- Provide a formal notation for specifying the syntax of a programming language (metalanguage)
- Use a finite notation to describe the syntax for all syntactically-valid programs of the language
- Are powerful enough to handle infinite languages
- Show the structure of the programs in the language
- Have been used extensively for almost every programming language since the definition of ALGOL 60.
- Drive parser development

©SoftMoore Consulting

Slide 6

6

Context-Free Grammars (continued)

- Are also known as Backus-Naur Form (BNF) grammars
- Are often processed by compiler tools (so called "compiler compilers")

Note: There are many different (but similar) notations for defining context-free grammars.

©SoftMoore Consulting

Slide 7

7

Context-Free Grammar

A context-free grammar (CFG) consists of four components

1. Finite set T of **terminal symbols** (a.k.a. the vocabulary)
 - represent symbols appearing in the language
 - examples: 25, x, if, <=
2. Finite set N of **nonterminal symbols**
 - represent the syntactic classes in the language
 - examples: expression, statement, loopStmt
3. Start symbol
 - one of the nonterminal symbols
 - often something like program or compilationUnit

©SoftMoore Consulting

Slide 8

8

Context-Free Grammar (continued)

4. Finite set of rules that define how syntax phrases are structured from terminal and nonterminal symbols
 - often called "syntax equations", "production rules" or simply "productions"
 - characterize possible substitutions for nonterminal symbols

©SoftMoore Consulting

Slide 9

9

Format for Rules

Rules have the following form:

- An equals symbol "=" separates the left side of the rule from the right side.
- The left side of the rule is a single nonterminal symbol, and every nonterminal symbol must appear on the left side of exactly one rule.
- The right side of the rule is a sequence of terminal symbols, nonterminal symbols, and other special symbols as define on the next slide.
- A period "." is used to terminate rules.

©SoftMoore Consulting

Slide 10

10

Extended Grammars (a.k.a., EBNF)

Allow the use of extra symbols on right-hand side of a rule:

- "|" for alternation (read "or" or "or alternatively")
- "(" and ")" for grouping
- "*" as a postfix operator to indicate that a syntax expression may be repeated zero or more times
- "+" as a postfix operator to indicate that a syntax expression may be repeated one or more times
- "?" as a postfix operator to indicate that a syntax expression is optional (i.e., it may be repeated zero or one times)

©SoftMoore Consulting

Slide 11

11

Using Extended Grammars

- Extended grammars allow rules to be expressed in a simple, easily understood form
- Example:


```
identifiers = identifier ( "," identifier )* .
```

©SoftMoore Consulting

Slide 12

12

Common Conventions for Rules

- Terminal symbols are quoted or specified explicitly.
- Start symbol is the left-hand side of the first rule.
- Set T consists of all terminal symbols appearing in the rules.
- Set N consists of all nonterminals appearing in the rules.

Using these conventions we can specify a grammar simply by specifying the set of rules.

©SoftMoore Consulting

Slide 13

13

Example: Grammar for CPRL

```

program = declarativePart statementPart "," .
declarativePart = initialDecls subprogramDecls .
initialDecls = ( initialDecl )* .
initialDecl = constDecl | arrayTypeDecl | varDecl .
constDecl = "const" constId ";" literal ";" .
literal = intLiteral | charLiteral | stringLiteral
         | booleanLiteral .
booleanLiteral = "true" | "false" .
arrayTypeDecl = "type" typeId "=" "array"
               "[" intConstValue "]" "of" typeName ";" .
varDecl = "var" identifiers ":" typeName ";" .
identifiers = identifier ( "," identifier )* .
typeName = "Integer" | "Boolean" | "Char" | typeId .
... (See Appendix D)

```

©SoftMoore Consulting

Slide 14

14

Lexical Versus Structural Grammars

- Lexical Grammar (handled by scanner)
 - simple rules to express the terminal symbols (tokens) in the grammar for the parser
 - based on regular expressions
 - It is possible to express the syntax for each symbol in a single rule.
 - Example:


```

identifier = letter ( letter | digit )* .
letter = [A-Za-z] .
digit = [0-9] .

```
- Syntax or structural grammar (handled by parser)
 - more complex rules that describe the structure of the language (recursive definitions allowed)
 - Example:


```

assignmentStmt = variable ":" "=" expression ";" .

```

©SoftMoore Consulting

Slide 15

15

Lexical Versus Structural Grammars (continued)

- Example
 - The scanner handles all identifiers.
 - The parser treats an identifier as if it were a terminal symbol.

©SoftMoore Consulting

Slide 16

16

Augmenting Rule

- Grammars often use an augmenting rule to ensure that all input is matched.
- The augmenting rule may be explicit or simply "understood".
- Example:


```

system = program <EOF> .
where <EOF> represents "end of file".

```

Ensures that a "complete" program is followed only by an end-of-file marker.

©SoftMoore Consulting

Slide 17

17

Alternate Rule Notations

- Use "→", ":", or simply ":" instead of "=" to separate left and right sides of rules.
- Use end of line (instead of period) to terminate rules.
- Use "{" and "}" to enclose syntax expressions that can be repeated 0 or more times. Similarly, use "[" and "]" to enclose optional syntax expressions.
- Enclose nonterminal symbols in "<" and ">" and omit quotes around terminal symbols.
- Use font changes such as bold to distinguish between terminal and nonterminal symbols.

©SoftMoore Consulting

Slide 18

18

Examples: Alternate Rule Notations

```
<program> ::= <declarativePart> <statementPart> .
```

period is a terminal symbol in the grammar, not a rule terminator

```
<initialDecls> ::= { <initialDecl> }
```

curly braces denote "zero or more"

©SoftMoore Consulting

Slide 19

19

Simple or Non-Extended Grammars

- Do not use special symbols for "alternation", "zero or more", "optional", etc.
- An extended grammar rule that uses alternation is expressed as multiple rules.
- The right side of a rule may be the empty string, denoted as λ .
- An extended grammar rule that uses "zero or more" or "optional" is expressed using recursion.

©SoftMoore Consulting

Slide 20

20

Simple Grammar Examples

- Example 1.** The extended grammar rule

```
initialDecl = constDecl | arrayTypeDecl | varDecl .
```

 becomes three rules in a simple grammar.

```
initialDecl = constDecl .
initialDecl = arrayTypeDecl .
initialDecl = varDecl .
```
- Example 2.** The extended grammar rule

```
identifiers = identifier ( "," identifier )*
```

 can be represented as three rules in a simple grammar.

```
identifiers = identifier identifiersTail .
identifiersTail = "," identifiers .
identifiersTail =  $\lambda$  .
```

©SoftMoore Consulting

Slide 21

21

Syntax Diagrams

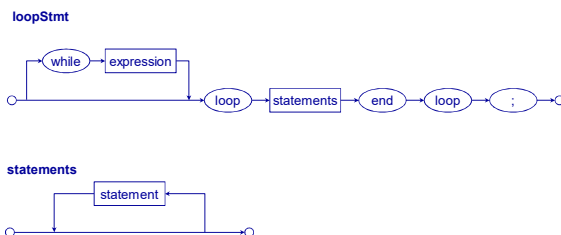
- Syntax diagrams (a.k.a., railroad diagrams) provide a graphical alternative to textual representations for a grammar.
- Textual representations for a grammar are more easily processed by compiler tools.
- Syntax diagrams are more easily understood by humans.
- Basic idea**
 - Directed graph.
 - Each diagram has an entry point and an end point.
 - Diagram describes possible paths between these two points by going through other nonterminals and terminals.
 - Terminals are represented by round boxes.
 - Nonterminals are represented by square boxes.

©SoftMoore Consulting

Slide 22

22

Example: Syntax Diagrams



©SoftMoore Consulting

Slide 23

23

Use of Naming Conventions for Nonterminal Symbols

- Many language designers use naming conventions for nonterminal symbols to convey contextual or semantic information in the grammar – roughly equivalent to a comment.
- Example 1:**

```
functionCall = funcId ( actualParameters )? .
funcId = identifier .
```

 This is equivalent to the following:

```
functionCall = identifier ( actualParameters )? .
```
- Example 2:**

```
loopStmt = ( "while" booleanExpr )? "loop" statements
           "end" "loop" ";" .
booleanExpr = expression .
```

©SoftMoore Consulting

Slide 24

24

Grammar Transformations: Substitution of Nonterminal Symbols

- Suppose we have a rule of the form

$$N = X .$$
 where the rule is nonrecursive and is the only rule for the nonterminal N.
- We can substitute X for every occurrence of N in the grammar, thereby eliminating the nonterminal N.
- A language designer may elect to leave the rule in the grammar; e.g.,

$$\text{booleanExpr} = \text{expression} .$$

©SoftMoore Consulting

Slide 25

25

Grammar Transformations: Left Factorization

- Suppose that a rule has alternatives of the form

$$X Y \mid X Z$$
- We can replace these alternatives by the following equivalent expression:

$$X (Y \mid Z)$$

©SoftMoore Consulting

Slide 26

26

Grammar Transformations: Elimination of Left Recursion

- Suppose that a rule has the form

$$N = X \mid N Y .$$
 where X and Y are arbitrary expressions.
- A rule of this form is said to be **left-recursive**.
- We rewrite this rule to obtain an equivalent rule that is not left-recursive.

$$N = X (Y)^* .$$

©SoftMoore Consulting

Slide 27

27

Example: Grammar Transformations

- Original Grammar

$$\begin{aligned} \text{identifier} &= \text{letter} \\ &\mid \text{identifier letter} \\ &\mid \text{identifier digit} . \end{aligned}$$
- Left factor

$$\begin{aligned} \text{identifier} &= \text{letter} \\ &\mid \text{identifier (letter \mid digit)} . \end{aligned}$$
- Eliminate left recursion

$$\text{identifier} = \text{letter (letter \mid digit)}^* .$$

©SoftMoore Consulting

Slide 28

28

Grammars versus Languages

- Understand the distinction between a language and a grammar.
- Different grammars can generate (define) the same language.

©SoftMoore Consulting

Slide 29

29

Derivation

- Systematically apply the rules one at a time, beginning with the start symbol.
- Grammar:

$$\begin{aligned} \text{expr} &= \text{expr op expr} \mid \text{id} \mid \text{intLit} . \\ \text{op} &= "+" \mid "*" . \end{aligned}$$
 Show that "2 + 3 * x" is valid.
- Left-most derivation:

$$\begin{aligned} \text{expr} &\Rightarrow \text{expr op expr} \\ &\Rightarrow \text{intLit op expr} \\ &\Rightarrow \text{intLit + expr} \\ &\Rightarrow \text{intLit + expr op expr} \\ &\Rightarrow \text{intLit + intLit op expr} \\ &\Rightarrow \text{intLit + intLit * expr} \\ &\Rightarrow \text{intLit + intLit * id} \end{aligned}$$

©SoftMoore Consulting

Slide 30

30

Parse Trees

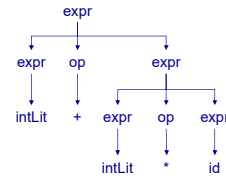
- A **parse tree** (a.k.a. **syntax tree**) of a grammar G is a labeled tree such that
 - the leaves (terminal nodes) are labeled by terminal symbols
 - the interior nodes (nonterminal nodes) are labeled by nonterminal symbols
 - the children of an interior node N correspond in order to a rule for N
- A parse tree provides a graphical illustration of a derivation.

©SoftMoore Consulting

Slide 31

31

Example: Parse Tree (from previous derivation)



©SoftMoore Consulting

Slide 32

32

Some Grammars Have Problems

- Using the same grammar, let's perform a different left-most derivation of "2 + 3 * x" by choosing a different alternative for expr in the beginning.
- Left-most derivation:


```

      expr => expr op expr
            => expr op expr op expr
            => intLit op expr op expr
            => intLit + expr op expr
            => intLit + intLit op expr
            => intLit + intLit op expr
            => intLit + intLit * expr
            => intLit + intLit * id
      
```

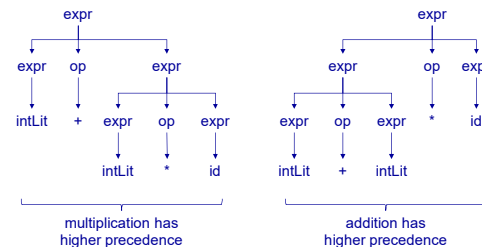
©SoftMoore Consulting

Slide 33

33

Some Grammars Have Problems (continued)

Consider parse tree for first derivation
versus parse tree for second derivation.



©SoftMoore Consulting

Slide 34

34

Some Grammars Have Problems (continued)

- **Ambiguous grammar** – some legal phrase has more than one parse tree.

©SoftMoore Consulting

Slide 35

35

Specifying Operator Precedence

- Operator precedence refers to the relative priority of operators
- Two approaches to specifying operator precedence
 1. Within the grammar
 2. Use an additional specification mechanism (e.g., a table) separate from grammar
- We will use the first approach in our definition for CPRL, but the second approach is supported by some compiler tools (e.g., yacc)

©SoftMoore Consulting

Slide 36

36

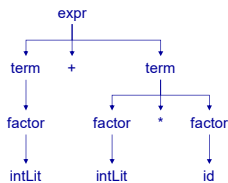
Specifying Operator Precedence Within the Grammar

- Grammar


```

      expr = term ( "+" term ) * .
      term = factor ( "*" factor ) * .
      factor = id | intLit
      
```

- Parse tree



Note that "*" has higher precedence than "+" in this grammar.

©SoftMoore Consulting

Slide 37

37

Associativity

- Specifies the evaluation order of operators with the same precedence when there are no parentheses.
- Example 1: CPRL operators + and - are at the same precedence level and are left associative.
8 - 3 + 2 is evaluated as (8 - 3) + 2
- Example 2: Some languages (not CPRL) use ^ as an exponentiation operator, and exponentiation is usually defined to be right associative.
2^2^3 is evaluated as 2^(2^3)

©SoftMoore Consulting

Slide 38

38

Abstract Syntax Trees

- Similar to a parse tree but without extraneous nonterminal and terminal symbols
- Example 1: For expressions, we could omit all the additional nonterminals introduced to define precedence (relation, simpleExpr, term, factor, etc.). All binary expressions would retain only the operator and the left and right operands.



©SoftMoore Consulting

Slide 39

39

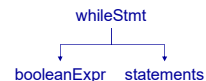
Abstract Syntax Trees (continued)

- Example 2: Consider the following grammar for a while statement:

```

whileStmt = "while" booleanExpr
           "loop" statements "end" "loop" ";" .
  
```

- Once a while statement has been parsed, we don't need to retain the terminal symbols. The abstract syntax tree for a while statement would contain only booleanExpr and statements.



©SoftMoore Consulting

Slide 40

40

Context-Free Grammar for Context-Free Grammars

```

grammar = ( rule )+ .
rule = identifier "=" syntaxExpr "." .
syntaxExpr = syntaxTerm ( "|" syntaxTerm )* .
syntaxTerm = ( syntaxFactor )+ .
syntaxFactor = identifier | terminalSym
              | "(" syntaxExpr ")" ( multChar )? .
multChar = "*" | "+" | "?" .
identifier = letter ( letter | digit )* .
terminalSym = "\"" ( terminalChar | escapedChar )* "\"" .
terminalChar = [ !#-\\[\]~ ] .
// any graphic ASCII character except backslash and quote
escapedChar = "\\" ( "\"" | "\\\" ) .
letter = [A-Za-z] .
digit = [0-9] .
  
```

©SoftMoore Consulting

Slide 41

41