CS 280
Programming Language Concepts

Spring 2025

Describing Syntax
Chapter 3

Outline

- Overview of Short Assignment 4
- Introduction to Language Definitions
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Regular Grammars
- Context-Free Grammars (CFG)
  - Derivations
  - Ambiguity in Grammars
  - Extended BNF

Short Assignment 4

- Objective of the SA 4:
  - The objective of the program is to recognize and count the occurrences of words of three types of words (as defined in SA 3).
  - See the posted problem statement on Canvas.
- Overview of the SA 4
  - The program accepts one or more command line arguments for a file name and optional input flags.
    - Reads from the file words until the end of file.
  - Program recognizes and counts the number of occurrences of each word of three types of words. Those are:
    - *Special Words*, *Keywords*, and *Identifiers*.
  - The program prints out each word of a category and the number of its occurrences in the input file, based on a specified command line argument for that category.

Short Assignment 4: Example

- An example of a test case is shown next. Given the following input file,

```
begin formatting a paragraph with italic form @
An International Standard Book Number (ISBN) is a code of
%10 characters, referred to in this case as the class $ ISBN-10 ,
separated by dashes such as private number $0-7637-0798-8
$End the formatting
begin
If the ISBN number starts with ISBN-10 it consists of four parts:
group code, a publisher code, a code that uniquely identifies the
book among those published by
a particular publisher, and a check character.
The check character is used to validate an ISBN. For the ISBN 0-
7637-0798-8, the group code is 0,
which identifies the book as one from an English-speaking country.
While the %publisher code @_ 7637 is for @Jones and @Bartlett
Publishers
```

Short Assignment 4: Example (Cot'd)

- The printed output with "-kw" and "sp" flags is as follows:

```
Invalid Special Word: $0-7637-0798-8
Total Number of Lines: 13
Number of Words: 118
Number of Special Words: 8
Number of Identifiers: 84
Number of Keywords: 9

List of Special Words and their number of occurrences:
$: 1
$End: 1
%10: 1
%publisher: 1
@: 1
@Bartlett: 1
@Jones: 1
@_: 1

List of Keywords and their number of occurrences:
begin: 2
case: 1
class: 1
for: 2
if: 1
private: 1
while: 1
```

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
  - The syntactic definition includes the forms for writing its language elements, such as expressions, statements, program units, etc.
- **Semantics:** the meaning of the expressions, statements, and program units
  - The semantics describe the meaning of the language elements (i.e., operational effects) when the program is under the execution of the computer.
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)
  - Example: while statement in Java

    while (condition) statement

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet (i.e., a finite set of symbols)
  - For example,

    index = 2 * count + 17;
- A *language* is a set of sentences
- A *lexeme* is the lowest level of a syntactic unit of a language (e.g., *, index, count, 17, `sum, begin`)
  - Formal descriptions of the syntax of programming Languages often do not include descriptions of the lowest-level syntactic units.
  - Lexemes are partitioned into groups. Each group is given a token or a name.
    - A *token* is a category of lexemes (e.g., identifier)
    - The process of recognizing lexemes is called Lexical Analysis

Example: Lexemes and Tokens

index = 2 * count + 17;

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

The General Problem of Describing Syntax

- **Recognizers**
  - A recognition device, R, reads input strings over the alphabet, Σ, of the language, L, and decides whether the input strings belong to the language L.
  - Example: syntax analysis part of a compiler acting as a filter that accepts correct sentences of the language only.

    - Detailed discussion of syntax analysis appears in Chapter 4

The General Problem of Describing Syntax

- **Generators**
  - A device that generates sentences of a language.
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator
  - Formal language-generation mechanisms, usually called grammars, are commonly used to describe the syntax of programming languages.
  - A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts. Therefore, a grammar is usually thought of as a language generator.

Formal Methods of Describing Syntax

- String patterns (i.e. tokens or lexemes) can be specified using three basic formal rules:
  - Concatenation,
  - Alternation: Choice among a finite number of alternatives), and
  - Kleene closure: Repetition an arbitrary number of times
  - Recursion: creation of a construct from simpler instances of the same construct.
- Any set of strings that can be defined by the first three rules are referred to as regular language.

Formal Methods of Describing Syntax

- Any set of strings that can be defined by the first three rules are referred to as regular language.
  - Regular languages are generated by *regular expressions* and recognized by scanners
- Any set of strings that can be defined if we add recursion is called a *context-free language* (CFL).
  - Context-free languages are generated by *context-free grammars* (CFGs) and recognized by parsers.

Regular Grammars

- A regular grammar is a simple scheme for using rules to represent strings to recognize.
- Regular grammars are the simplest and least powerful of the grammars.
- Regular grammars can be used to recognize strings that match a particular pattern.
  - They can't recognize all patterns in general.
  - This: $\{ a^n b^n \mid n \geq 1 \}$
    - Is not a regular language and so can't be recognized with a regular grammar
  - In other words, a regular grammar cannot balance paired items:
    - ( ), { }, begin end.

Regular Expressions

- A regular expression is a notation for expressing patterns of characters
- Regular expressions are all over CS
  - String finding in editors
  - Pattern expansion is built into command interpreters (OS UI through consoles, e.g., saying “ls*.c” in UNIX is a form of this)
- Many languages have a regex library of some form.
  - Java and Python have regex libraries.
- Some languages with string types may have regular expression matching built in.
  - Perl has reg expressions included in the language syntax.

Matching Strings to Regular Expressions

- The sequence of characters in a regular expression are matched against a string.
- A character in a regular expression is either a regular character, which must exactly match the character in the string, or a **metacharacter**, which stands for something else
  - Example: a dot ('.') in a regular expression is a metacharacter that means "matches any single character in the string"
- To specify tokens, we use the following notations of *regular expressions (as shown by the table in the next slide)*.
  - Note: Escaping a metacharacter with a backslash changes the metacharacter to its non-metacharacter meaning
    - Example, \. (backslash dot) matches the character dot, NOT the metacharacter meaning of "any character".
  - Example : a regular expression M* indicates through the * metacharacter zero or more repetitions of the expression M.

Matching Strings to Regular Expressions

RegExpr	Meaning
x	a character x
\x	an escaped character, e.g., \n
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[*characters*]	choose from the characters in []
[aeiou]	the set of vowels
[0-9]	the set of digits
. (that's a dot)	Any single character

You can parenthesize items for clarity

Matching Strings to Regular Expressions

- Example Regular Expressions for Tokens
  - identifier: [a-z_A-Z][0-9a-zA-Z]*
  - octal constant: 0[0-7]+
    - 0123 is octal const in C/C++
  - hex constant: 0x[0-9a-fA-F]+
    - 0xfd23 hex const
  - floating point number:

    [+-]?([0-9]*\.[0-9]+)e[+-][0-9]+
    - 1.e-2 is not accepted pattern
    - 1.5e2 is not accepted pattern

Matching Strings to Regular Expressions

- The metacharacters+ and * are “greedy”; they match as many characters as possible
- Matching can be automated
  - There are libraries that compile regular expressions and use them to match strings
- A tool known as lex (or flex) is designed to use regular expressions to automatically generate a lexical analyzer for a compiler/interpreter
- The matcher is a simple machine called a finite state machine or finite state automata.

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages

- Backus-Naur Form (1959)
  - Invented by John Backus to describe the syntax of Algol 58, and later modified by Peter Naur to describe Algol 60.
  - BNF is equivalent to context-free grammars

BNF Fundamentals

- BNF is a metalanguage: A language that is used to describe another language.

- In BNF, **abstractions** are used to represent classes of syntactic structures (e.g, assignment statement, if-stmt, while stamt, etc.)--they act like syntactic variables (also called *nonterminal symbols*, or just *nonterminals*).

  <assign> -> <var> = <expression>

- *Terminals* are the lexemes or tokens. They are symbols (tokens) that are not defined in terms of other terminals or nonterminals.

- A rule or production has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals.

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<assign> → <var> = <expression>
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

  - Example <assign>: total = value + rate * price;

- Definition: Context-free grammar, G, is defined as the tuple of 4 elements: G = (S, N, T, P), where
  - S: A *start symbol* which is a special element of the nonterminals of a grammar
  - T: Set of terminals
  - N: Set of Non-terminals
  - P: a finite non-empty set of rules

BNF Fundamentals (continued)

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

- Syntactic lists are described using recursion. A rule is recursive if its LHS appears in its RHS.

```
<ident_list> → ident
             | ident, <ident_list>
```

BNF Fundamentals (continued)

- A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts
- A grammar is a generative device for defining languages.
  - Sentences of the language are generated through a sequence of applications of the rules beginning from the start symbol, called derivation.
  - A derivation is a repeated application of the rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Summary-BNF

- BNF uses following notations:
  - Non-terminals enclosed in < and >.
  - Symbols without angle brackets are terminals.
  - ”→” means “is defined as” (some variants use “::=” or “:=” instead)
  - Rules written as $X \rightarrow Y$
    - X is LHS of rule and can only be a nonterminal.
    - Y is RHS of rule: Y can be
      a. a terminal, nonterminal, or concatenation of terminals and nonterminals, or
      b. a set of strings separated by alternation symbol |.
- Notation ε: Used to represent an empty string (a string of length 0).

An Example Grammar

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

- All derivations start from the start symbol: <program> in this grammar
- => symbol means derives
- First step: **<program> => <stmts>**
- Second step: <program> => **<stmts> => <stmt>**
- Third step: <program> => <stmts> => **<stmt> => <var> = <expr>**

An Example Derivation: Using Leftmost Derivation

1. **<program> → <stmt_list>**
2. **< stmt_list > → <stmt>; | <stmt> ; < stmt_list >**
3. **<stmt> → <var> = <expr>**
4. **<var> → a | b | c | d**
5. **<expr> → <term> + <term> | <term> - <term>**
6. **<term> → <var> | const**

- Derivation for this sentence: a = b + const;

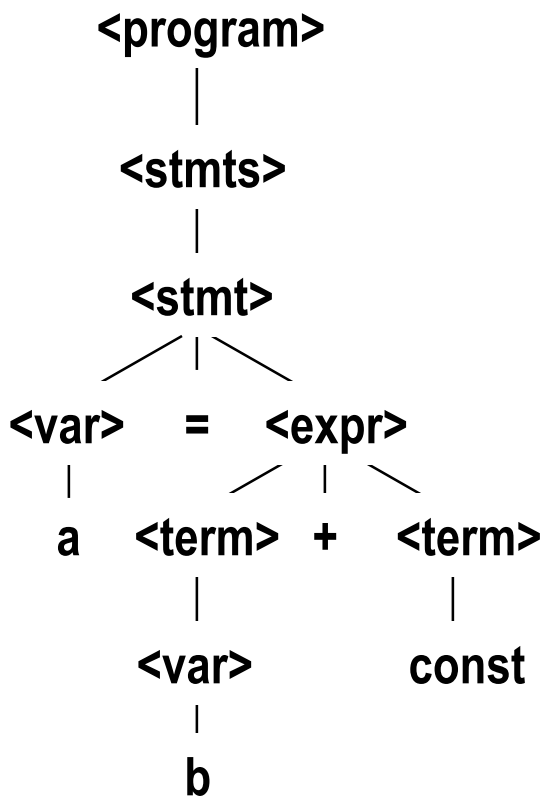
<program> => <stmt_list> => <stmt>;
=> **<var>** = <expr>;
=> a = **<expr>**;
=> a = **<term>** + <term>;
=> a = **<var>** + <term>;
=> a = b + **<term>**;
=> ***a = b + const;***

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be neither leftmost, nor rightmost.

Parse Tree

- Grammar rules represent the hierarchical syntactic structure of the language sentences.
- Parse Trees represent the derivation steps for the hierarchical structures. Each internal node of the tree is a nonterminal and each leaf is a terminal.
- Each step in the derivation is a level in the tree
- A traversal of the tree (inorder) is a representation of the expression that you parsed
- Traversal of the tree for execution is in postorder.




```
a = b + const
```

Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees.
  - Approaches to determine a grammar is ambiguous include:
    - If the grammar generates a sentence with more than one leftmost derivation.
    - If the grammar generates a sentence with more than one rightmost derivation.

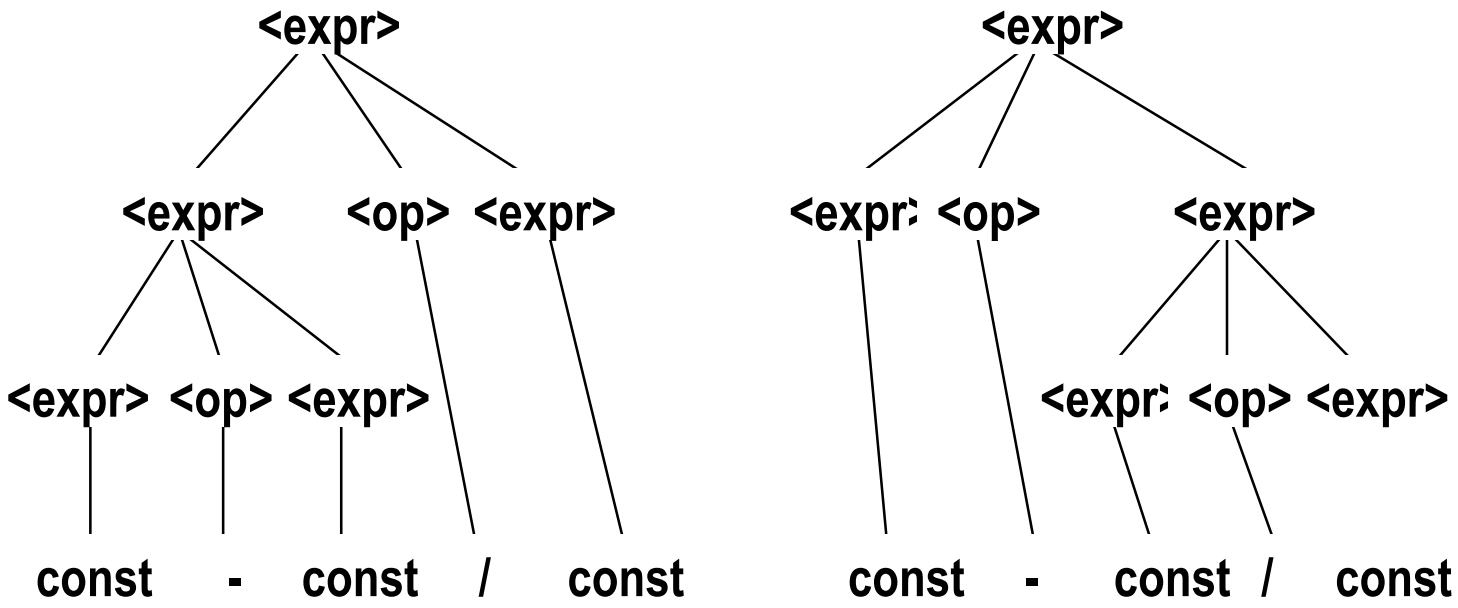
Ambiguity in Grammars

- Example: An Ambiguous Expression Grammar

  <expr> → <expr> <op> <expr> | const

  <op> → / | -

  - Using two different leftmost derivations

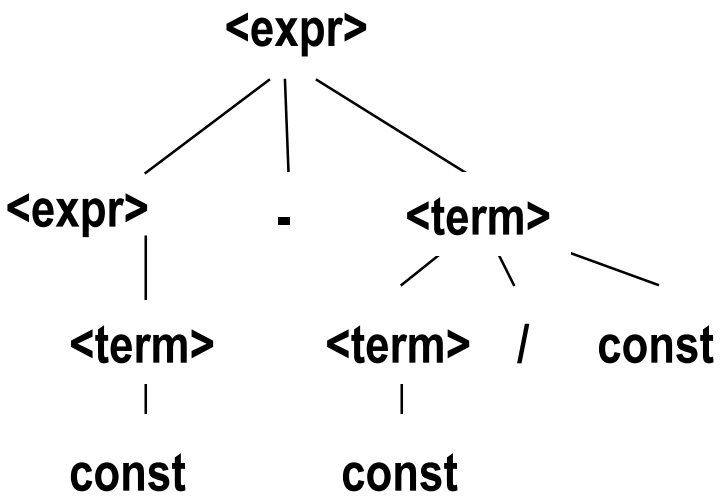


Ambiguity in Grammars

- Example: **An Unambiguous Expression Grammar**
  - If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity. We can rewrite the grammar such that:
    - A separate nonterminal symbol term is used as an operand to '-'.
    - <expr> always generates '-' at higher level of the tree than the generation of '/' by the term nonterminal.

```
<expr> → <expr> - <term> |
         <expr> + <term> | <term>
<term> → <term> / const |
         <term> * const | const

Sentence:
  const – const/const
```



Ambiguity in Grammars

- Associativity of Operators
  - Operator associativity can also be indicated by a grammar rule.
  - Ambiguous grammar example:

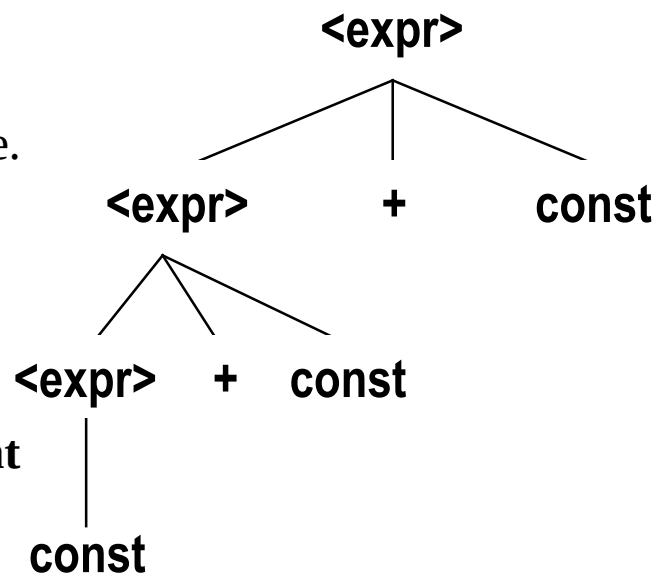
```
<expr> -> <expr> + <expr> |
<expr> - <expr> | const
```

  - The grammar rule is rewritten to be left recursive. **The left recursion specifies left associativity.**

```
<expr> -> <expr> + const |
          <expr> - const | const
```

  - **Note that right recursion specifies right associativity. Example:**

<Assign>-> <var> = <Assign> | <Expr>



Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call>->ident (<expr_list>) | ident`

  `<proc_call> -> ident [(<expr_list>)]`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (*|/|%) <factor> | factor`

- Repetitions (0 or more) are placed inside braces { }
  - `<term> → <factor> {(*|/|%) <factor>}`

    `<ident> → letter {letter|digit}`

- Terminals that are grammar symbols ('[' for instance) are enclosed in quotes (‘’).

BNF and EBNF

- BNF

```
<expr> → <expr> + <term> | <expr> - <term>
         | <term>
<term> → <term> * <factor> | <term> / <factor>
         | <factor>
<factor> → <exp> ** <factor> |<exp>
<exp> → (<exp>) | id
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> {** <exp>}
<exp> → (<exp>) | id
```

- There is a standard for EBNF, ISO/IEC 14977:1996 but it is rarely used.

BNF and EBNF

- **Note**
  - In BNF, <expr> → <expr> + <term> clearly specifies the + operator to be left associative.
  - In EBNF, <expr> → <term> {(+ | -) <term>} does not imply the direction of associativity
  - The problem of associativity has been overcome in syntax analyzers by enforcing the correct associativity.
- Recent Variations in EBNF
  - Alternative RHSs are put on separate lines.
  - NonterminIs begin with uppercase letters (discard <>)
  - Use of a colon instead of =>
  - Use of $_{opt}$ for optional parts
  - Use of `oneof` for choices

Conversion from EBNF to BNF and Vice Versa

- BNF to EBNF:

  (i) Look for recursion in grammar:

  A ::= a | B ⇒ EBNF rule: A ::= { a } B

  (ii) Look for common string that can be factored out with grouping and options.

  A ::= a B | a ⇒ EBNF rule: A := a [B]

- EBNF to BNF:

  (i) Options: []

  A ::= a [B] C ⇒ BNF rules: A' ::= a N C, N ::= B | ε

  (ii) Repetition: {}

  A ::= a { B1 B2 ... Bn } C ⇒ BNF rules: A' ::= a N C, N ::= B1 B2 ... Bn N | ε