Military Techincal Academy

Principles of Programming Language

Syntax

Introduction

- Languages have a “grammar”, i.e., rules
- Programming languages have a grammar as well
- Designers design them
- Users learn and use them
- This lecture: **How to describe them**

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

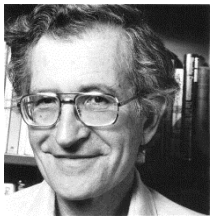
- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

Formal Definition of Languages

- **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

BNF and Context-Free Grammars

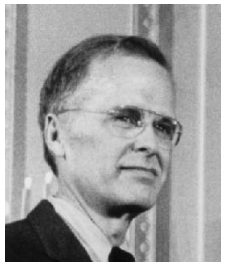
- Context-Free Grammars



  - 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

BNF and Context-Free Grammars

- Backus-Naur Form (1959)



  - Invented by John Backus to describe Algol 58
  - Modified by Peter Naur

  - BNF is equivalent to context-free grammars

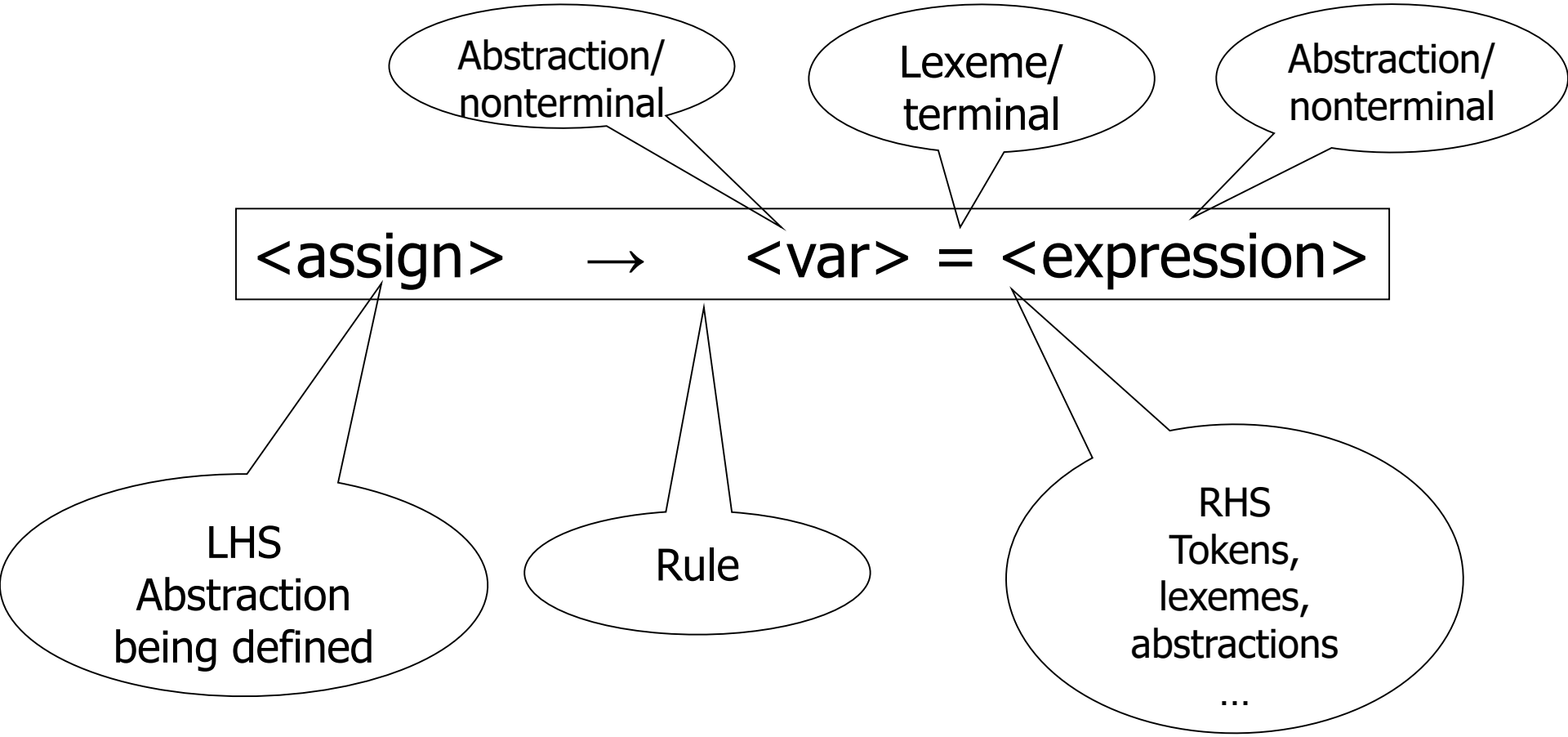
BNF Fundamentals

- A metalanguage is a language that is used to describe another language
- BNF is a metalanguage for programming language

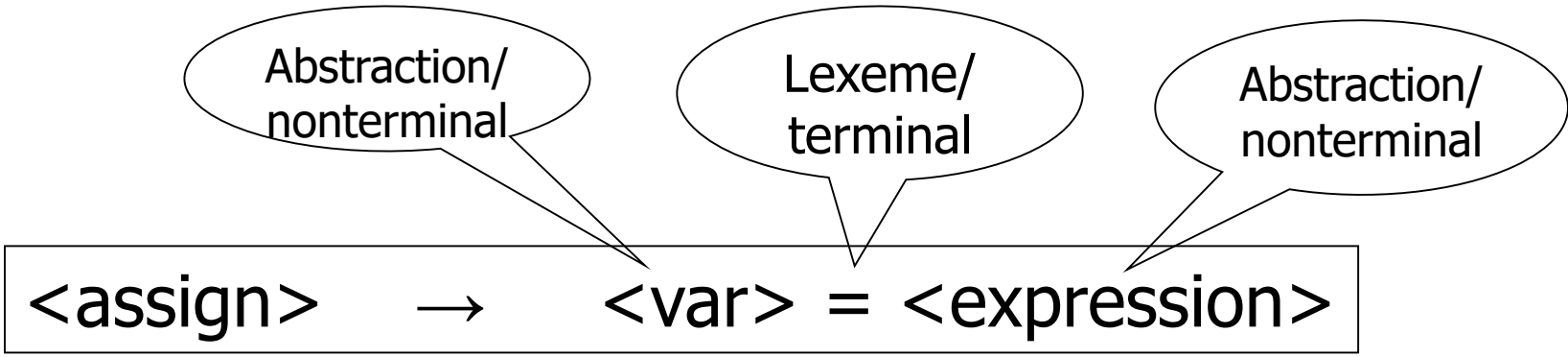
BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures
- Abstraction is called *nonterminals*
- *Terminals* are lexemes or tokens
- A rule 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
- Nonterminals are often enclosed in angle brackets

BNF Fundamentals



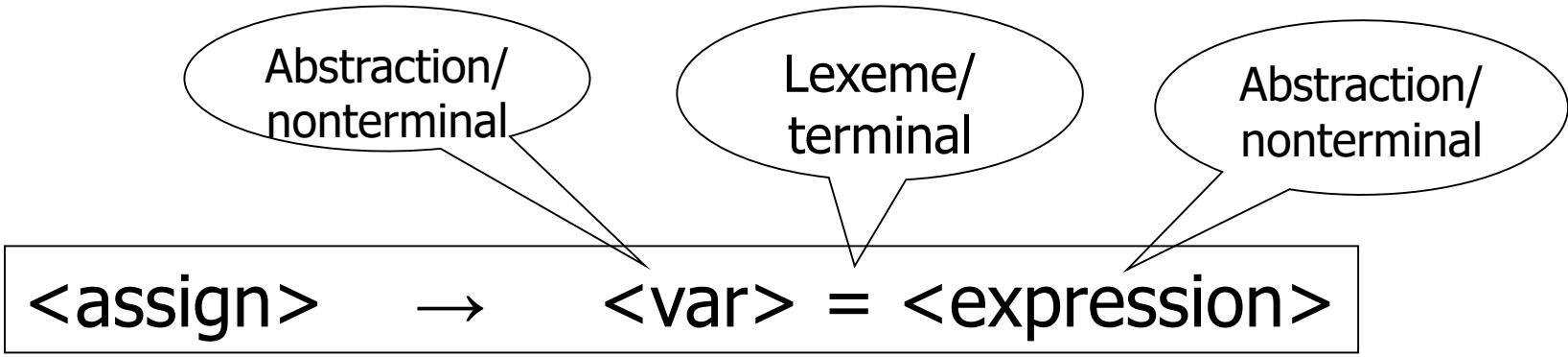
BNF Fundamentals



An example sentence whose syntactic structure is described by this rule:

Total = subtotal1 + subtotal2

BNF Fundamentals

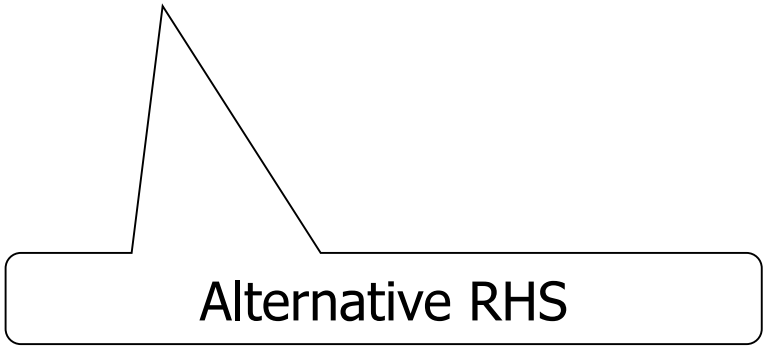


A BNF description, or grammar, is simply a collection of rules

BNF Rules

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

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



Describing Lists

- Syntactic lists are described using recursion

<ident_list> → ident | ident, <ident_list>

Grammar and Derivation

- A grammar is a generative device for defining a language
- Sentences are generated through a sequence application s of the rules, beginning with a special nonterminal of the grammar call the **start symbol**
- A sentence generation is called a derivation.
- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

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

An Example: Grammar

EXAMPLE 3.1 — A Grammar for a Small Language

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

An Example: A Derivation

<program> => begin <stmt_list> end
=> begin <stmt>; <stmt_list> end
=> begin <var> = <expression>; <stmt_list> end
=> begin A=<expression>; <stmt_list> end
=> begin A = <var> + <var>; <stmt_list> end
=> begin A = B + <var>; <stmt_list> end
=> begin A = B + C; <stmt_list> end
=> begin A = B + C; <stmt> end
=> begin A = B + C; <var> = <expression> end
=> begin A = B + C; B = <expression> end
=> begin A = B + C; B = <var> end
=> begin A = B + C; B = C end

An Example: A Derivation

<program> => begin <stmt_list> end
=> begin <stmt>; <stmt_list> end
=> begin <var> = <expression>; <stmt_list> end
=> begin A=<expression>; <stmt_list> end
=> begin A = <var> + <var>; <stmt_list> end
=> begin A = B + <var>; <stmt_list> end
=> begin A = B + C; <stmt_list> end
=> begin A = B + C; <stmt> end
=> begin A = B + C; <var> = <expression> end
=> begin A = B + C; B = <expression> end
=> begin A = B + C; B = <var> end
=> begin A = B + C; B = C end

The replaced nonterminal is always the leftmost nonterminal in the previous sentential form – **leftmost derivation**

An Example: A Derivation

<program> => begin <stmt_list> end
=> begin <stmt>; <stmt_list> end
=> begin <var> = <expression>; <stmt_list> end
=> begin A=<expression>; <stmt_list> end
=> begin A = <var> + <var>; <stmt_list> end
=> begin A = B + <var>; <stmt_list> end
=> begin A = B + C; <stmt_list> end
=> begin A = B + C; <stmt> end
=> begin A = B + C; <var> = <expression> end
=> begin A = B + C; B = <expression> end
=> begin A = B + C; B = <var> end
=> begin A = B + C; B = C end

Stop when the sentential form contains no nonterminals

An Example: A Derivation

- By choosing alternative RHSs of rules with which to replace nonterminals in the derivation, different sentences in the language can be generated.
- By exhaustively choosing all combinations of choices, the entire language can be generated.
- This, like most others, is infinite. It is impossible to generate all the sentences in the language in finite time

An Example: Grammar

EXAMPLE 3.2	A Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

This grammar describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses.

An Example: A Derivation

- The statement

  A = B*(A+C)

is generated by leftmost derivation:

<assign> =><id>=<expr>
=>A=<expr>
=>A=<id>*<expr>
=>A=B*<expr>
=>A=B*(<expr>)
=>A=B*(<id>+<expr>)
=>A=B*(A+<expr>)
=>A=B*(A+<id>)
=>A=B*(A+C)

Parsing

- Latin – Quae pars orationis?
- English – Which part of speech?

- Parsing is the process of breaking down sentences
  - In natural languages
  - In programming languages
  - In command and control languages

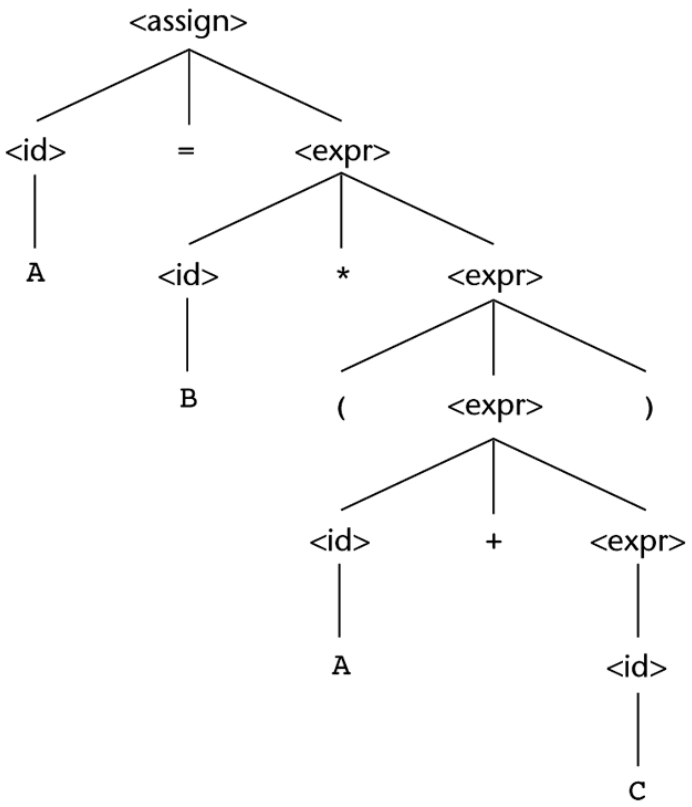
Parse Tree

- A parse tree shows how a string of terminals can be generated from the rules of the grammar
- Root is labeled with the (nonterminal) start symbol
- Each leaf is labeled with a terminal (or a string of terminals)
- The label of a non-leaf node is a non-terminal

Parse Tree

Figure 3.1

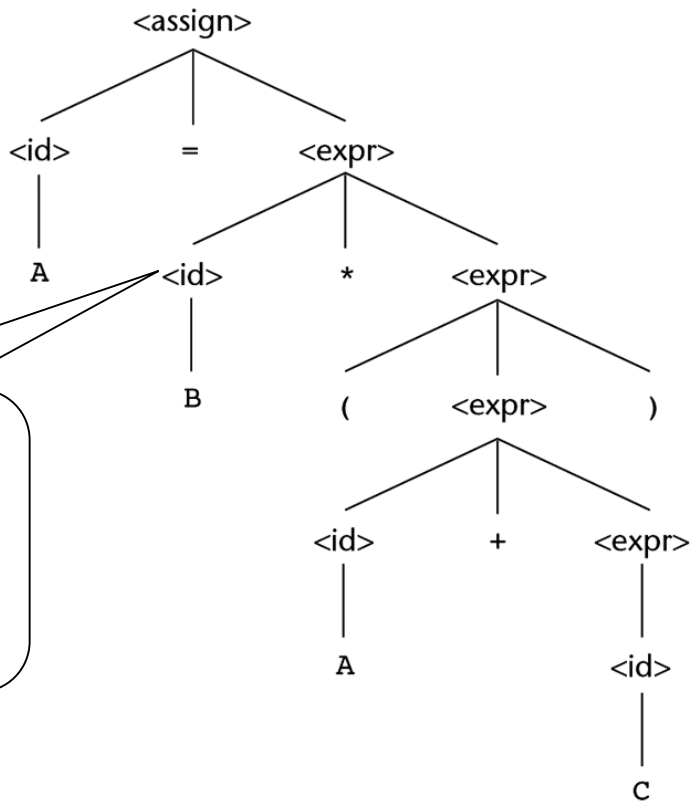
A parse tree for the simple statement
`A = B * (A + C)`



Parse Tree

Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`

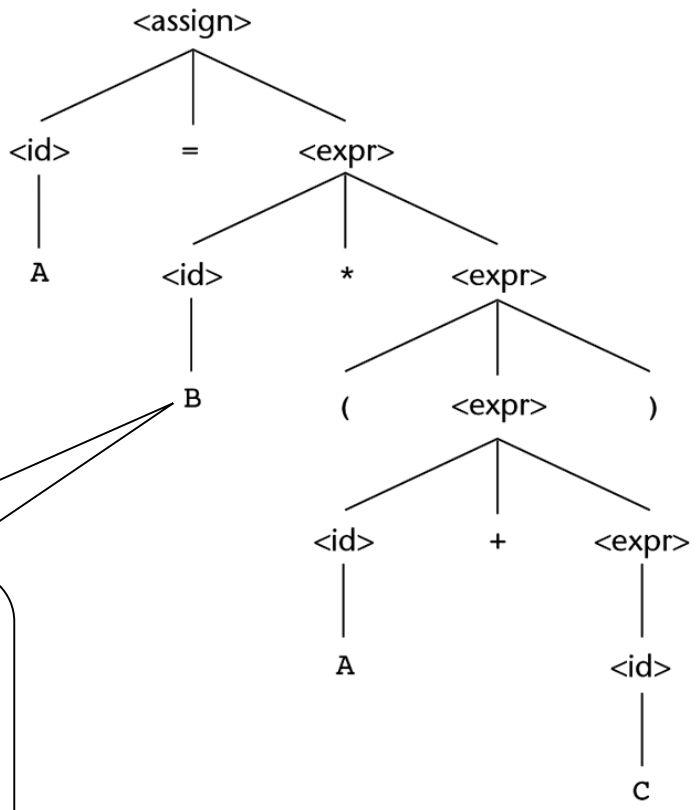


Every internal node is labeled with a nonterminal symbol

Parse Tree

Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`

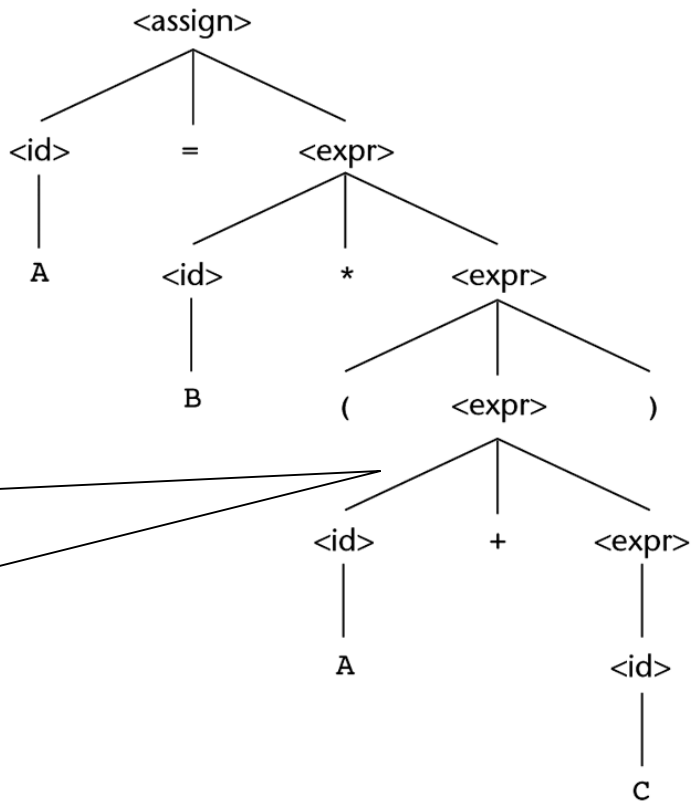


Every leaf is labeled with a terminal symbol

Parse Tree

Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`



Every sub-tree describes one instance of an abstraction in the sentence

Ambiguity in Grammars

- ***Example grammar***

<expr> → <expr> + <expr> | <expr> * <expr> | (<expr>)
| <number>
<number> → <number> <digit> | <digit>
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Given 234 we have different derivations

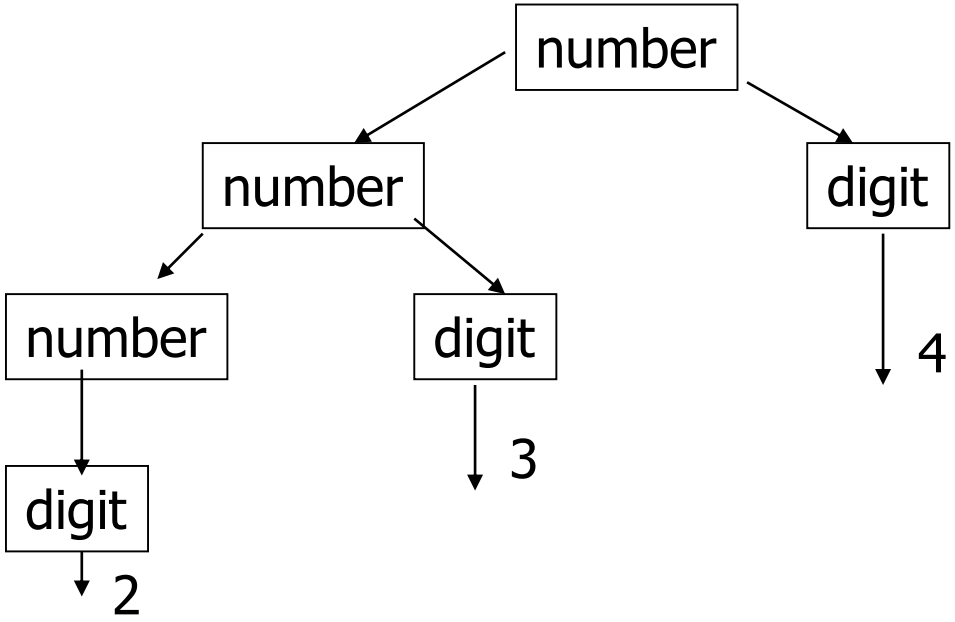
number => <number><digit>
=> <number> 4
=> <number><digit> 4
=> <number> 3 4
=> <digit> 3 4
=> 234

number =><number>< digit>
=> <number>< digit> <digit>
=> <digit>< digit>< digit>
=> 2 <digi><t digit>
=> 2 3 <digit>
=> 234

Ambiguity in Grammars

- ***Example grammar***

- However the parse tree is the same in either case



Ambiguity in Grammars

- Two different derivations can lead to the same the parse tree, this is good because the grammar is unambiguous

- However, it isn't always the case.

Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

Ambiguity in Grammars

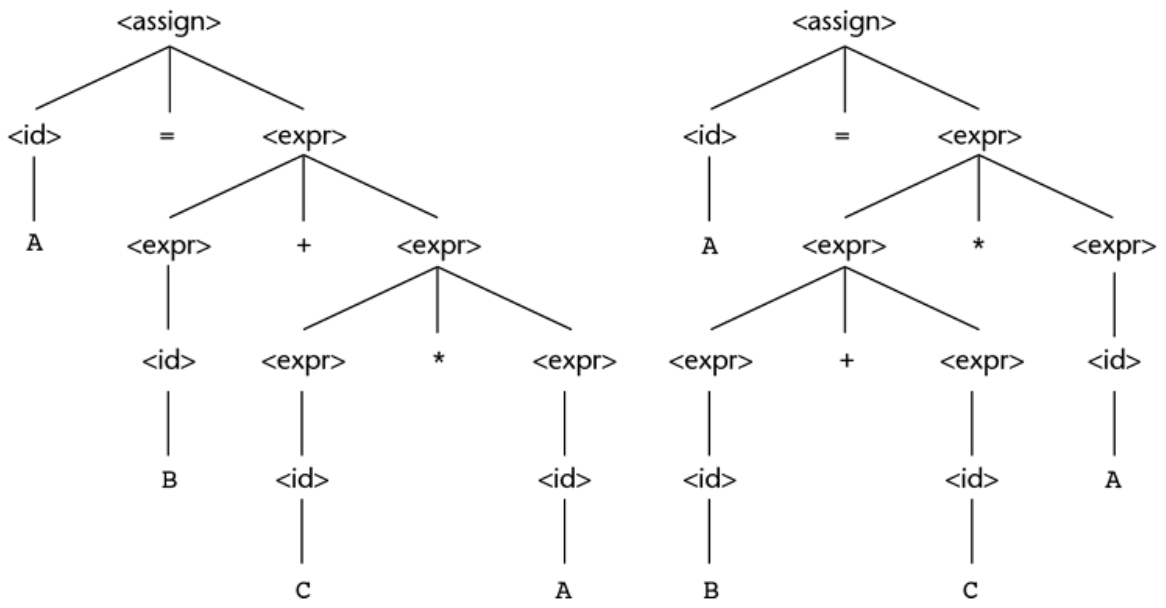
EXAMPLE 3.3	An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Ambiguity in Grammars

Figure 3.2

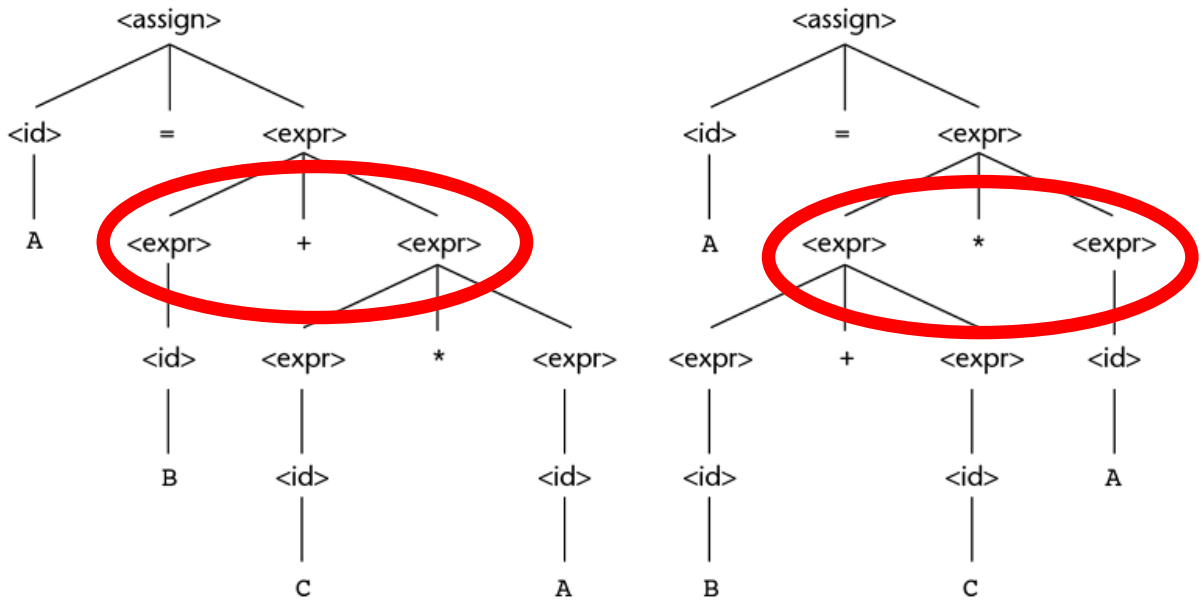
Two distinct parse trees for the same sentence,
`A = B + C * A`



Ambiguity in Grammars

Figure 3.2

Two distinct parse trees for the same sentence,
`A = B + C * A`



Removing Ambiguity

- A grammar that produces different parse trees depending on derivation order is considered *ambiguous*
- We can try to revise the grammar and introduce a disambiguating rule to establish which of the trees we want
- In the previous example we want multiplication to take precedence over addition, thus we tend to write a special grammar rule that establishes a precedence cascade to force the * at the lower point in the tree

Operator Precedence

x + y*z

- One semantic issue is the order of evaluation
- This can be resolved by assigning different precedence levels
- * is assigned a higher precedence than + (by the designer)
- Multiplication will always be done first, regardless of the order of appearance of the two operators

Operator Precedence

- An operator is generated lower in the parsing tree indicates it has precedence over an operator higher up in the tree

- In the grammar, introduce more nonterminals and new rules
  - <expr> (defines +, -)
  - <term> (defines *, /)
  - <factor> (defines numbers, unary -)

Operator Precedence

EXAMPLE 3.3 An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Force + to the top of the passing tree

EXAMPLE 3.4 An Unambiguous Grammar for Expressions

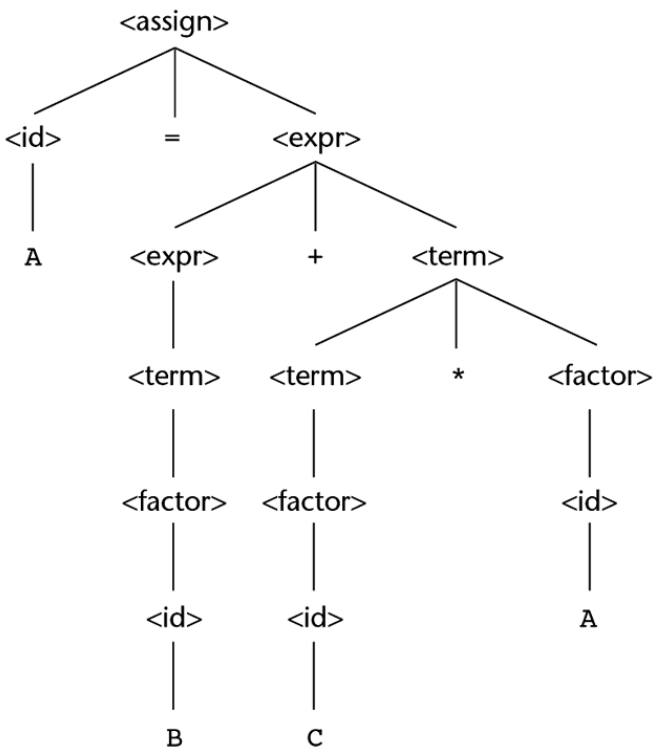
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

* will be lower in the parsing tree, because it is farther from the start symbol than + in every derivation

Operator Precedence

Figure 3.3

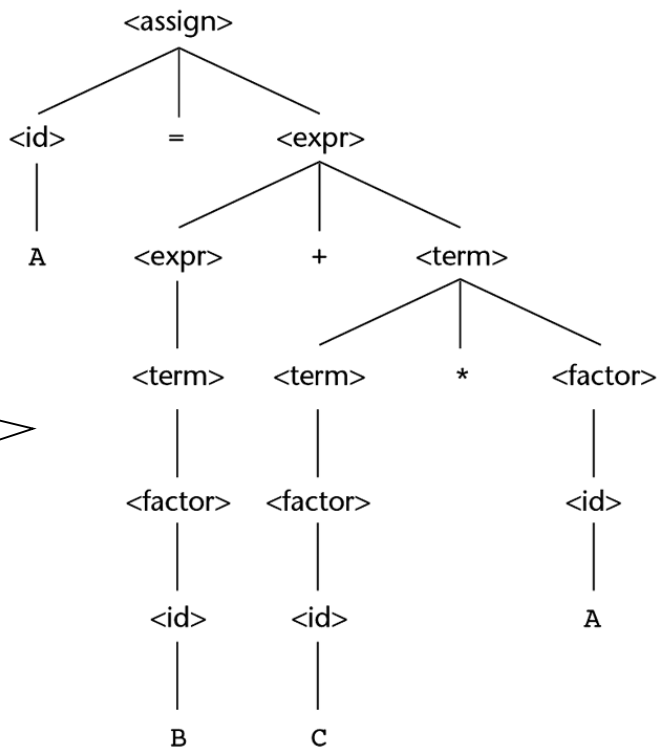
The unique parse tree for `A = B + C * A` using an unambiguous grammar



Operator Precedence

Figure 3.3

The unique parse tree for `A = B + C * A` using an unambiguous grammar



Leftmost or Rightmost derivation will lead to the same parsing tree

Associativity of Operators

- When an expression includes two operators that have the same precedence, A/B*C, a semantic rule is required to specified which should have precedence – this rule is **associativity**

Associativity of Operators

- Left Recursive – LHS appearing in the beginning of its RHS – specifies left associativity

EXAMPLE 3.4 An Unambiguous Grammar for Expressions

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → ( <expr> )
         | <id>
```

+ and * are left associative

Associativity of Operators

- Right Recursive – RHS appearing in the right end of the RHS – specifies right associativity

- Exponentiation should is right associative

<factor> → <exp>**<factor> | <expr>

<exp> → (<expr>) | id

Could be used to describe exponentiation as a right associative operator

Extended BNF

- Optional parts are placed in brackets [ ]

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

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

  <term> → <term> (+|-) const

- Repetitions (0 or more) are placed inside braces { }

  <ident> → letter {letter|digit}

Extended BNF

EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

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

EBNF:

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

Semantics Overview

- Specifying the semantics of a programming language is a much more difficult task than specifying syntax.
- We need formal semantics to define all the props of a language that are not specified with a BNF (declaration before use, some type issues, etc.)

- Three approaches
  - Reference Manual Approach (in English as precise as we can make things)
  - Define a translator- see what the language does by experimentation
    - Seems ridiculous but is this how HTML/CSS/JS is often dealt with, the popular browser(s) being the translators?
  - Formal definition – very precise, but complex

Semantics Overview

- Specifying the semantics of a programming language is a much more difficult task than specifying syntax.
- We need formal semantics to define all the props of a language that are not specified with a BNF (declaration before use, some type issues, etc.)

- Three approaches
  - Reference Manual Approach (in English as precise as we can make things)
  - Define a translator- see what the language does by experimentation
    - Seems ridiculous but is this how HTML/CSS/JS is often dealt with, the popular browser(s) being the translators?
  - Formal definition – very precise, but complex

Semantics Overview

- The advantages of the formal definition is that it is so precise that programs can be proven correct and translators validated to produce the defined behavior.

- Formal definitions for semantics have not met with complete acceptance, multiple approaches are pushed and many are not well understood and most not used with common languages (at least not initially).

Semantics Overview

- Three methods for formal semantics
  - Operational Semantics – defines a language by describing its actions in terms of operations on an actual or hypothetical machine
  - Denotational semantics – uses mathematical functions on programs to specify semantics
  - Axiomatic semantics – applies mathematical logic to language definition. Assertions or predicates are used to describe desired outcomes and initial assumptions. Constructs transform new assertions out of old ones reflecting the action of the construct. These transforms can prove the desired outcome follows from the initial conditions (a correctness proof)

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
  - To use operational semantics for a high-level language, a virtual machine in needed
  - A hardware pure interpreter would be too expensive to create

Semantics

- A software pure interpreter also has problems:
- A possible alternative: A complete computer simulation

- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer

- Evaluation of operational semantics:
- Good if used informally (language manuals, etc.)
- Extremely complex if used formally (e.g., VDL –Vienna Definition Language)

Semantics

- Axiomatic Semantics
- Based on formal logic (first order predicate calculus)
- Original purpose: formal program verification
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)

Semantics

- Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- The process of building a denotational spec for a language (not necessarily easy):
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

Semantics

- The difference between denotational and operational semantics:
- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, they are defined by rigorous mathematical functions

Semantics

- Evaluation of denotational semantics:
  - Can be used to prove the correctness of programs
  - Provides a rigorous way to think about programs
  - Can be an aid to language design
  - Has been used in compiler generation systems
  - Probably way beyond what most folks will get involved with

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- Three primary methods of semantics description
  - Operation, axiomatic, denotational