Principles of Programming Languages

Topic: Formal Languages, part A

Review

- **See Sakai for**
  - **Syllabus**
  - **Rules and Procedures for Exams**
  - **Exam Schedule**
- **This class will teach you**
  - **some new ways of thinking about programs (new *paradigms*)**
  - **some common principles underlying most languages**

Review

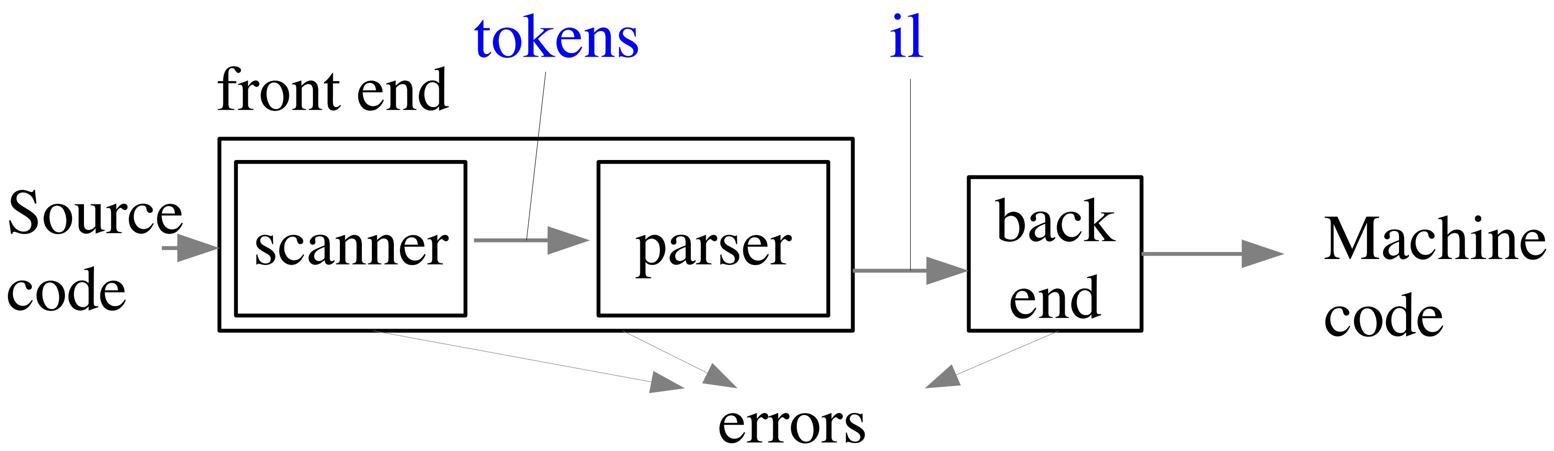
- **A programming language should:**
  - **be as easy as possible for people to read, write, and learn**
  - **be as easy as possible for a compiler to compile into efficient machine code or an interpreter to execute**
  - **be as powerful as possible *for the task at hand:***

    scale, novelty, paradigm, …

**"The best chainsaw is no good at cutting paper.
The best scissors are no good at cutting logs."**

Traditional Two-Pass Compiler

- **Pass: read and write entire program**



il = intermediate language

Scanner

- **Maps characters → tokens**
  - **Tokens: basic unit of syntax**
  - **E.g., x = x + y; becomes**

    <id, x> <operator, assign> <id, x> <operator + > <id, y>
  - **Typical token types:**

    number, id, operator (e.g., +), keyword (e.g., do, else)

Parser

- **Parse: determine the grammatical structure of a sequence of tokens**
- **Grammar: set of rule like**

  assignment → variable '=' expression ';'
- **Analogy with grammars of human languages.**
  - **e.g., English: Sentence → Subject Verb Object**

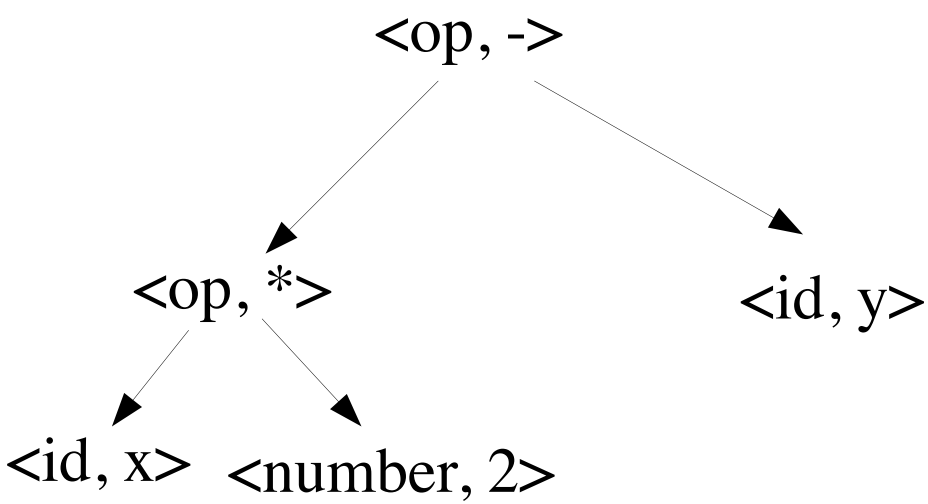
    The dog bit Bob

    Bob bit the dog ← different meaning

    Dog Bob the bit ← meaningless

Abstract Syntax Tree

For: x * 2 - y



Defining a Language

- **To define a computer language we need to say**
  - **What are the parts of the text of the program and how are they related: *syntax***
  - **What are the parts of the behavior specified by the program and how are they related: *semantics***
- **Syntactic structure should mirror semantic structure**

Text and Behavior

- **Text of the program**

  a = b + c ;

- **Behavior of the program**

  calculate:
  add the value of variable b
  to the value of variable c

  and store the result in variable a

Parts

- Text of the program

a = b + c ;

- Behavior of the program

calculate:
add the value of variable b
to the value of variable c

and store the result in variable a

Defining a language

- **Semantics:**
  - several formal approaches but in practice we just use English to explain the meaning
- **Syntax:**
  - tokens defined by a Regular Expression or Finite State Automaton
  - Larger scale structure defined by a Context Free Grammar

Formal Languages

For now, we will use the following terms in a special technical sense:

- Alphabet: a set of symbols
  {a, b} or {if, while, for, …}
- String: a sequence of symbols from the alphabet
  abbab
- Language: a set of strings
  - {ab, abab, ababab, …}
  - Each string: finite length
  - Set: possibly infinite number of strings

Specifying a Language

- **Languages can be infinite sets so we can't just list all strings in the language, must describe the set**
  - **"any string of alternating 'a' and 'b' that starts with 'a' and ends with 'b' "**
- **Formalisms for specifying a language**
  - **Grammars**
  - **Regular Expressions**
  - **Automata**

Grammar

- **A grammar is a formalism for specifying a language**
  - **A set of *Terminal Symbols*, the alphabet of the language**

    a, b
  - **A set of *Non-terminal Symbols*, the grammatical categories of the language, one of which is the "start" symbol**

    MultiAB, AB
  - **A set of *Rule*s of the form**
    Non-terminal* => *sequence of Symbols

    MultiAB => MultiAB   AB

    MultiAB => AB

    AB => a b

Grammar Rules

- **Another term for Rule is *Production***
- **A rule is of the form**

  Left-side* => *Right-side

  where *Left-side* is a single Non-terminal,

  and *Right-side* is a sequence of Terminals and/or Non-terminals

  Assignment => Variable '=' Expression ';'

Grammar

- **Usually we just give the Rules, since you can see from them what the Terminals and Non-terminals are.**

Grammar GAB:

start symbol

MultiAB => MultiAB   AB

MultiAB => AB

AB => a b

Grammar for (Part of) English

<S> => <NP> <VP>

<NP> => <Name> | <Det> <Noun>

<VP> => <Verb> | <Verb> <NP>

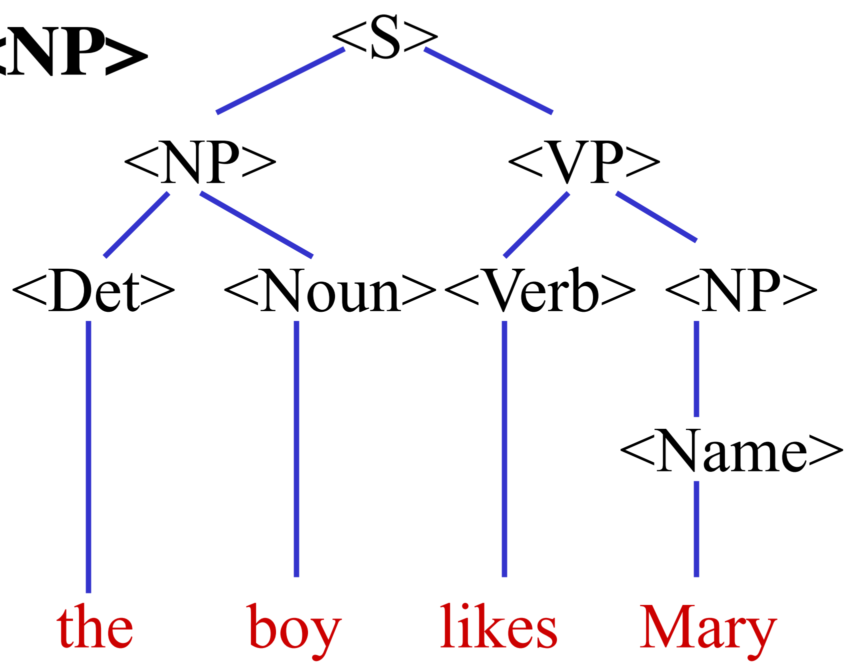
<Name> => John | Mary

<Det> => a | the

<Det> => some | every

<Noun> => boy | girl

<Verb> => runs | likes



The Language of a Grammar

Initialize a list of symbols to be just the Start Symbol

Repeatedly:

- Find the first Non-terminal in the list
- Find a rule whose Left-hand side is this non-Terminal
- Replace the Non-terminal with the Right-hand side of that rule

Until the list contains only Terminals

The Language of a Grammar

Grammar GAB:

MultiAB => MultiAB AB

MultiAB => AB

AB => a b

Derivation of a b a b from GAB

MultiAB

MultiAB AB

AB AB

a b AB

a b a b

The Language of a Grammar

- **Proof that a b a b a b is in L(GAB):**

  MultiAB

  MultiAB AB

  MultiAB AB AB

  AB AB AB

  a b AB AB

  a b a b AB

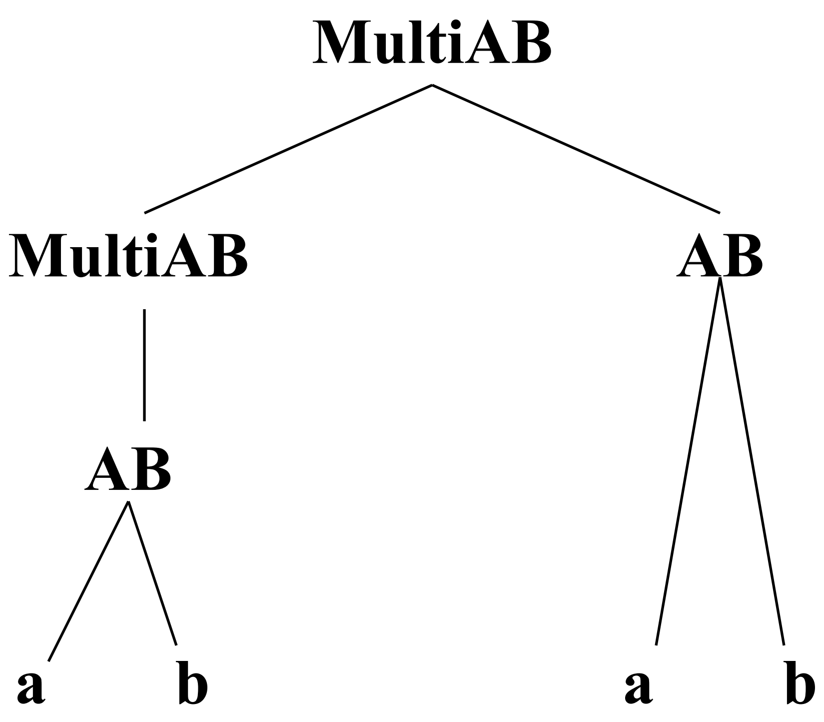
  a b a b a b

Parse Trees

- **The order in which Non-terminals are replaced does not matter**

MultiAB	**MultiAB**
MultiAB AB	**MultiAB AB**
AB AB	**MultiAB a b**
a b AB	**AB a b**
a b a b	**a b a b**

Parse Trees



Each internal node is a Non-terminal; its children make up the right-hand side of one of the productions for that Non-terminal.

Derivation in a Grammar

Question: Write a leftmost derivation of aabb in G2?

G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b

Answer:

<Stmt> => <A> <B>

=> a<A> <B>

=> aa <B>

=> aa <B>b

=> aa bb

Derivation in a Grammar

Question: Write a leftmost derivation of baa in G2?

G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b

Derivation in a Grammar

Question: Write a leftmost derivation of baa in G2?

Answer:

Impossible!

No way to get b left of a

G2:

<Stmt> => <A> | <A> <B>

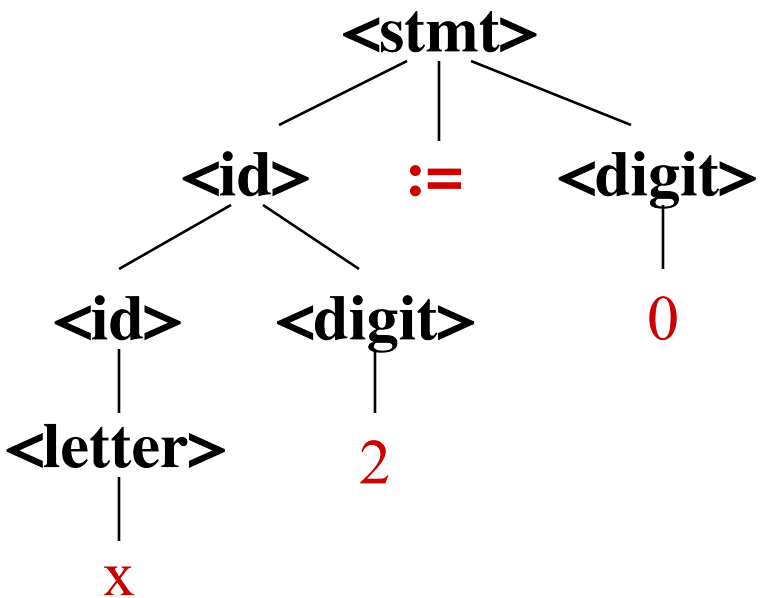
<A> => a | a <A>

<B> => b | <B> b

Derivation vs Parse

- **Derivation: <start> => … => list of Terminals**
- **Parse: list of Terminals => … => <start>**

Parse Trees



Each internal node is a nonterminal; its children make up the right-hand side of one of the productions for that nonterminal.

Parse Trees

Draw a parse tree for
aabb in G2?

G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b

Parse Trees

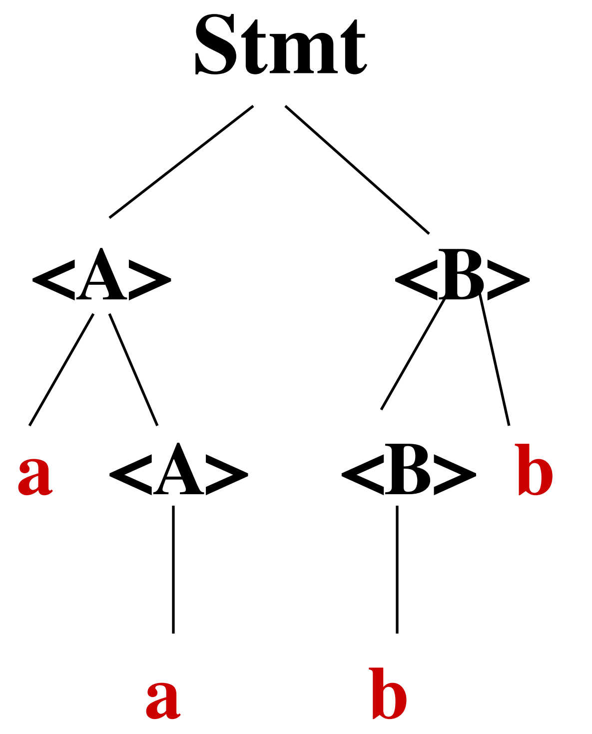
Draw a parse tree for
aabb in G2?

G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b



Grammars are not Unique

- **Consider a grammar G:**

  <stmt> ::= <ident> := <digit>

  <ident> ::= <letter>

  <ident> ::= <ident> <letter>

  <ident> ::= <ident> <digit>

  <letter> ::= a | b | c | ... | x | y | z

  <digit> ::= 0 | 1 | … | 8 | 9

- **Consider a grammar G':**

  <stmt> ::= <ident> := <digit>

  <ident> ::= <letter>

  <ident> ::= <ident> <char>

  <char> ::= <letter> | <digit>

  <letter> ::= a | b | c | ... | x | y | z

  <digit> ::= 0 | 1 | … | 8 | 9

The grammar G' generates the same language as G, but it has different parse trees.

G vs G'

- **G**

  …

  <identifier> ::= <letter> | <identifier> <letter> |
  <identifier><digit>

  …

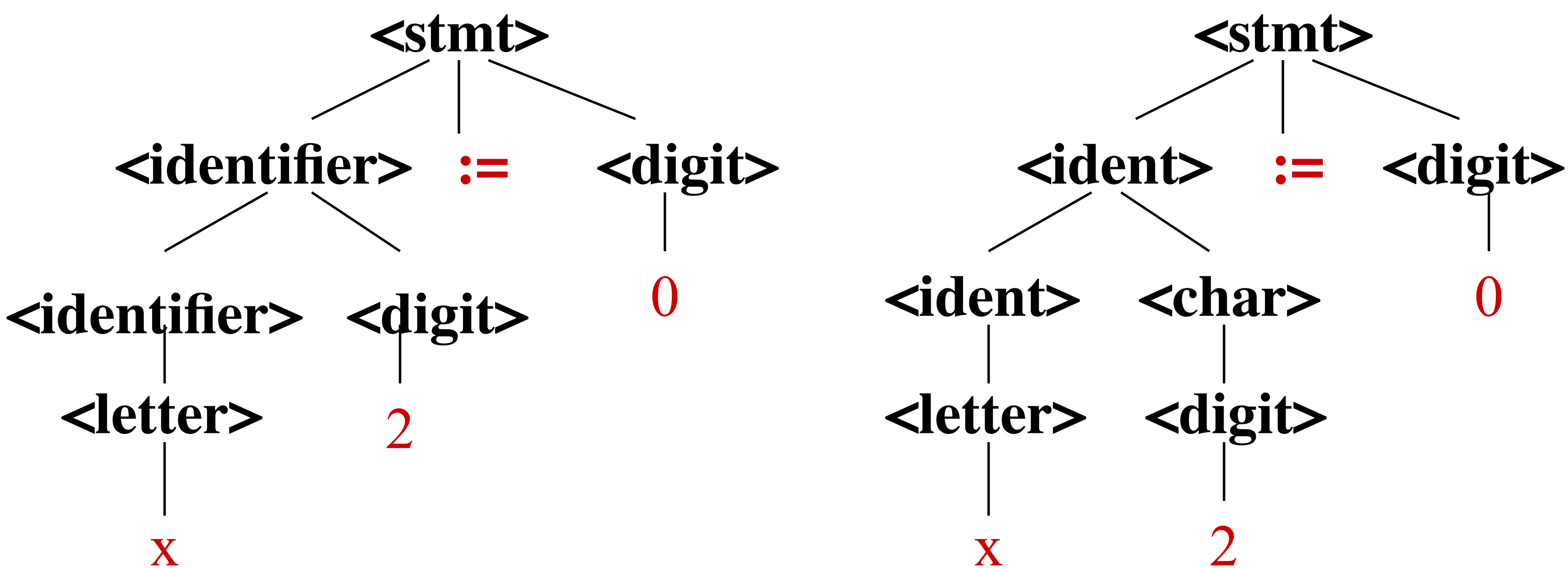
- **G': …**

  <ident> ::= <letter> | <ident> <char>

  <char> ::= <letter> | <digit>

  …

Grammars are not Unique

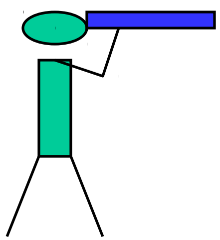


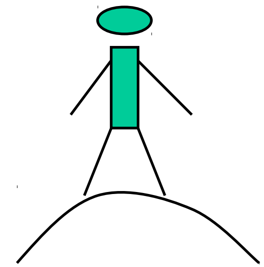
Parse Tree for G

Parse Tree for G’

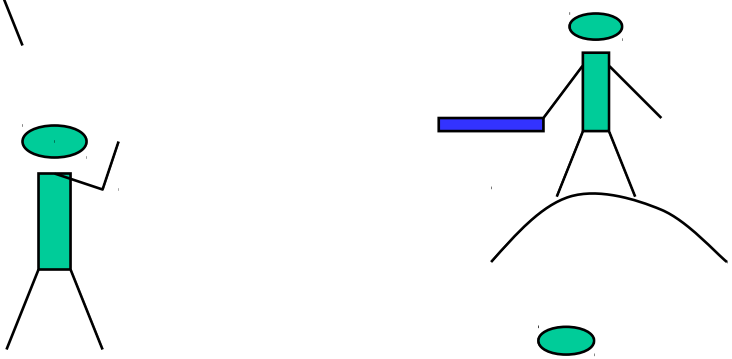
Ambiguity within a grammar

- John saw the man on the hill with the telescope
  - Who had the telescope?
  - John

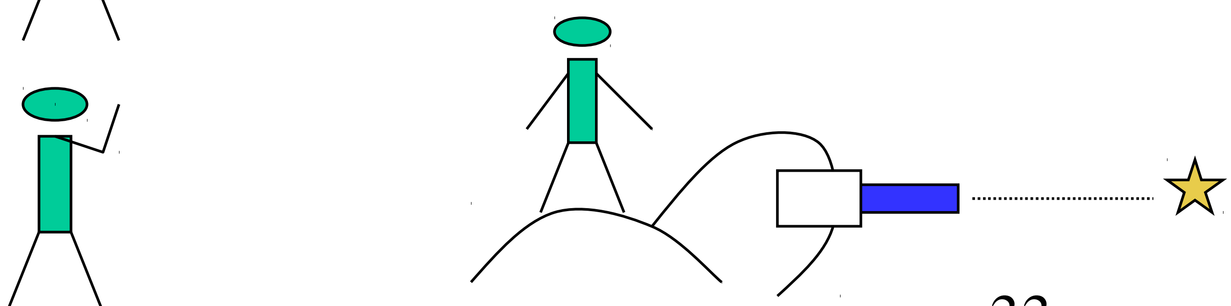




  - The man



  - The hill



Arithmetic Expressions

Here is a grammar for arithmetic expressions:

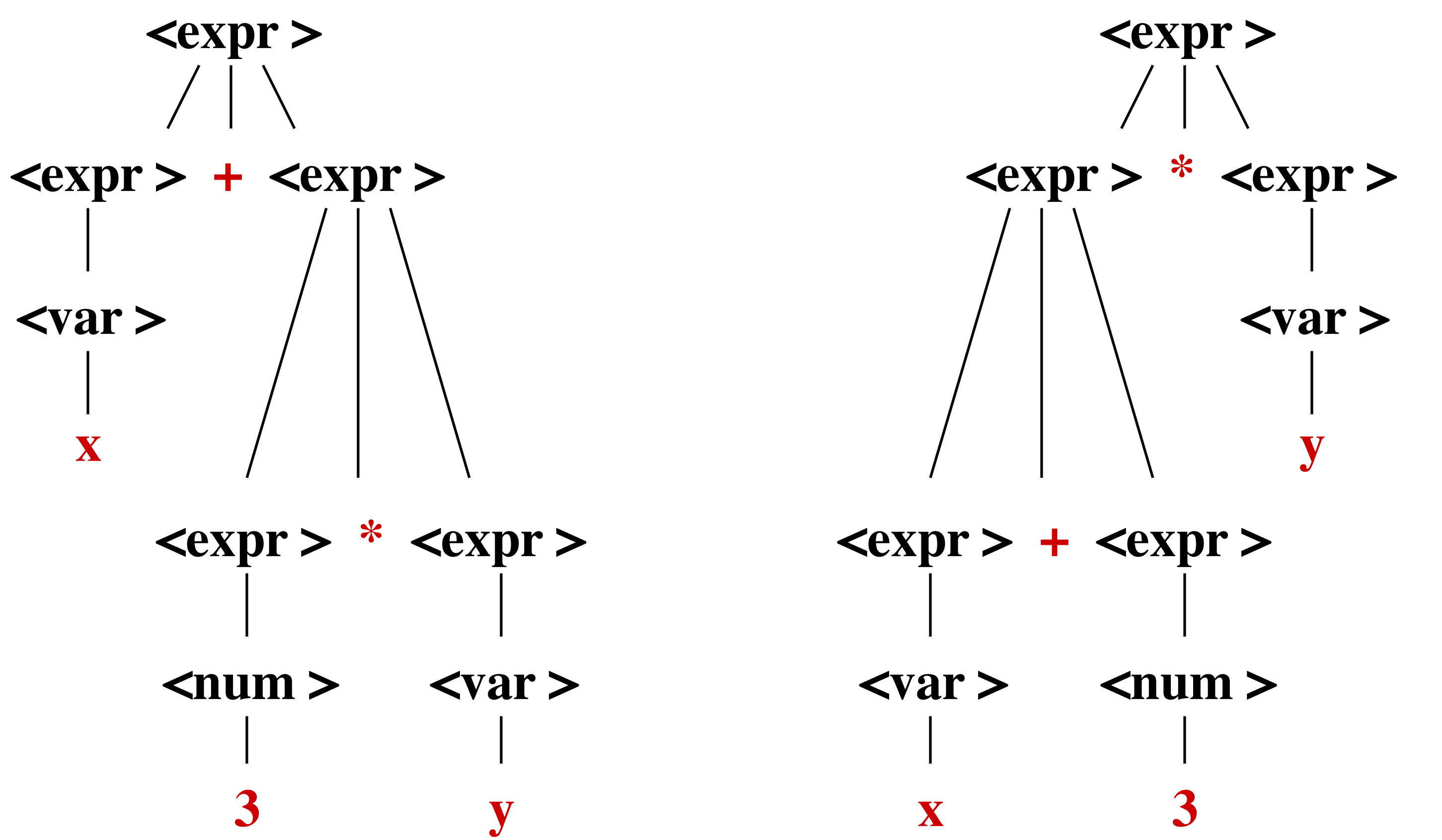
<expr> ::= <expr> + <expr> | <expr> - <expr> |
<expr> * <expr> | <expr> / <expr> |
<var> | <num>

<var> ::= a | b | c | ... | x | y | z

<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Using this grammar, how would we parse: x + 3 * y ?

Two Parse Trees



Two Parse Trees

- in a PL we want to base meaning on parse so
- ambiguous parse -> ambiguous meaning -> ☹

Precedence

Modify the grammar to add precedence:

<expr> ::= <expr> + <expr> | <expr> - <expr> | <term>

<term> ::= <term> * <term> | <term> / <term> | <factor>

<factor> ::= <var> | <num> | (<expr>)

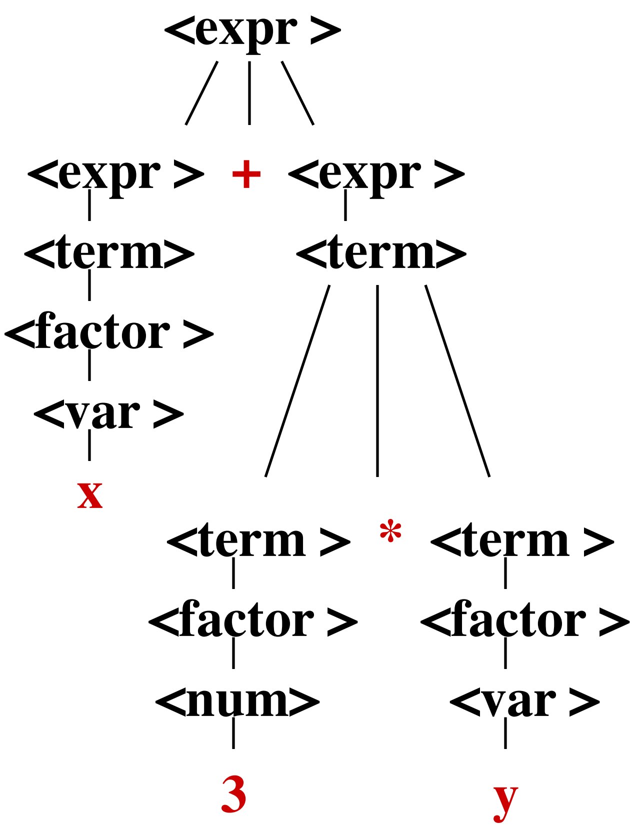
<var> ::= a | b | c | ... | x | y | z

<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

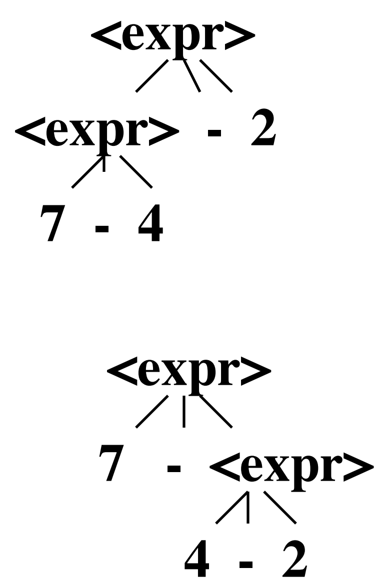
Using this grammar, how would we parse: x + 3 * y ?

Using this grammar, how would we parse: 7 - 4 - 2 ?

Only One Parse Tree



But there are two parse trees for the second example:



Associativity

Modify the grammar to add associativity:

<expr> ::= <expr> + <term> | <expr> - <term> |
<term>

<term> ::= <term> * <factor> | <term> / <factor> |
<factor>

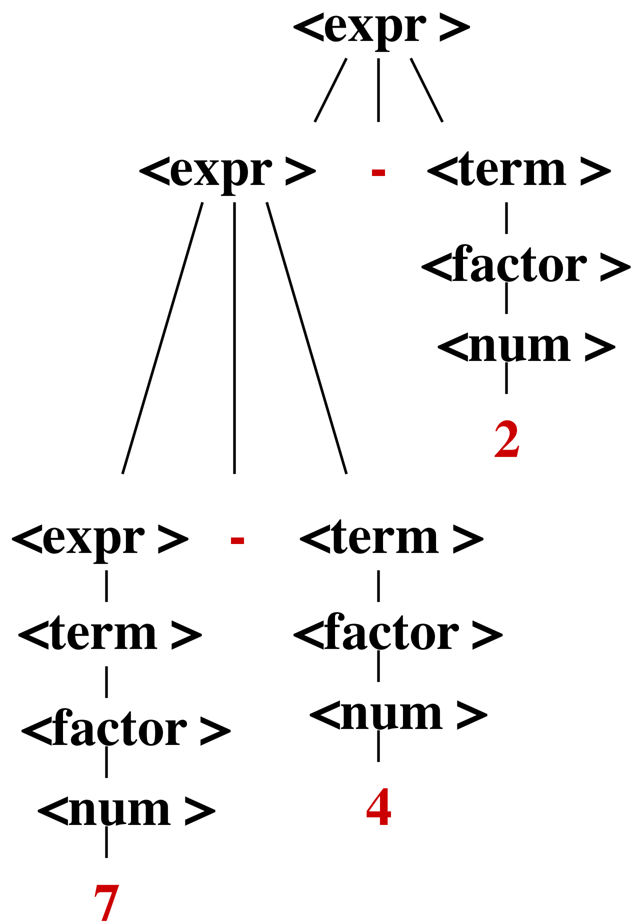
<factor> ::= <var> | <num> | (<expr>)

<var> ::= a | b | c | ... | x | y | z

<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Using this grammar, how would we parse: 7 - 4 - 2 ?

Only One Parse Tree



Solution

- **Solution: encode precedence & associativity in grammar**
  - **non-terminal for each level of precedence**

    + - <term>

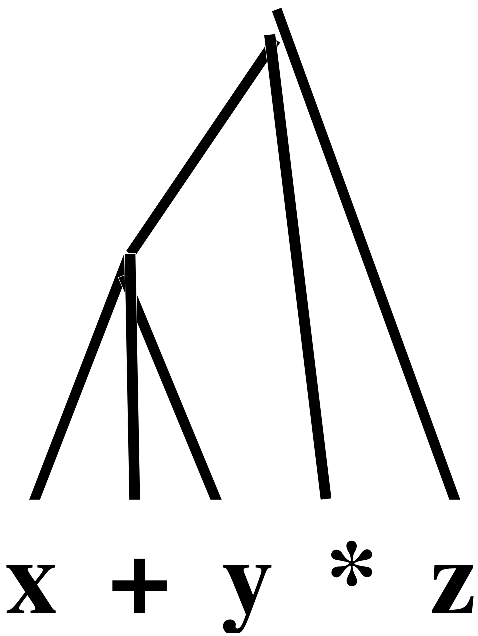
    *** / <factor>**
  - **for each level of precedence:**

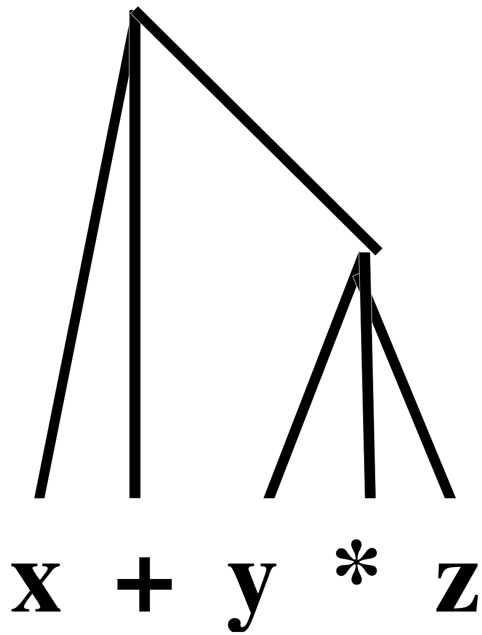
    <nt1> ::= <nt2> (nt2 higher precedence)

    <nt1> ::= <nt1> + <nt2> (left associative)

How it works

- **Higher in the tree means lower precedence**





`(x+y)*z` **`x+(y*z)`**

How it works: Precedence

NT1 => NT2 | NT1 + NT2 (NT1 is Start Symbol)

NT2 => NT3 | NT2 * NT3

NT3 => Number

To get 1+2*3:

Can only get + from an NT1 so start

NT1

NT1 + NT2

How it works: Precedence

NT1 => NT2 | NT1 + NT2 (NT1 is Start Symbol)

NT2 => NT3 | NT2 * NT3

NT3 => Number 1+2*3

NT1 + NT2 * is right of +, so must come from here

NT1 + NT2 * NT3

How it works: associativity

NT1 => NT2 | NT1 + NT2 (NT1 is Start Symbol)

NT2 => NT3 | NT2 * NT3

NT3 => Number 1+2+3

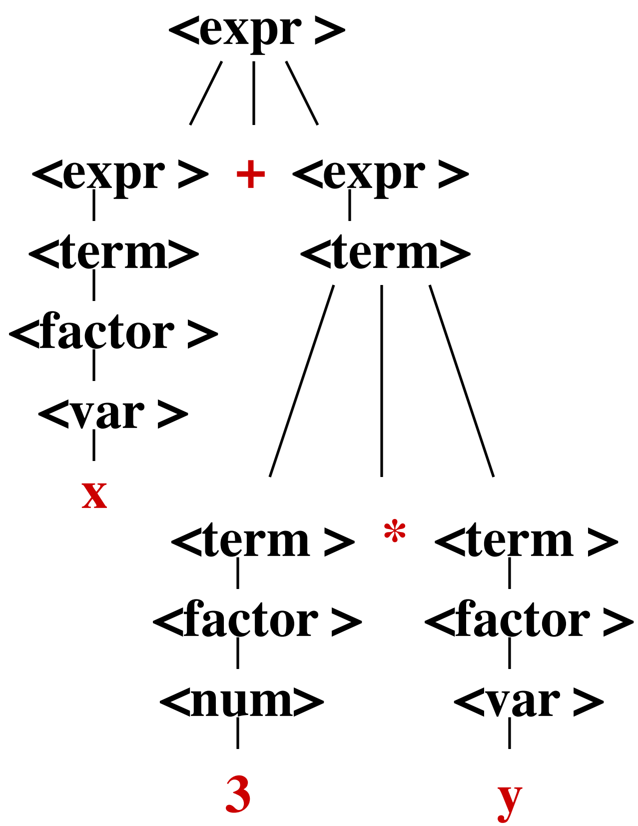
Can only get + from an NT1

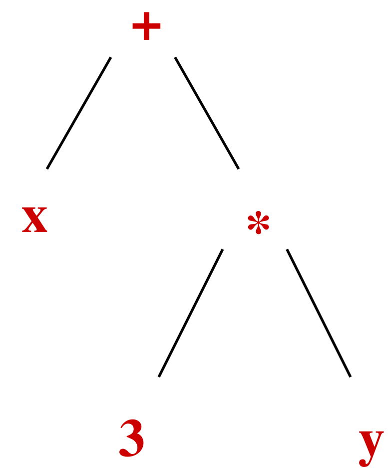
NT1

NT1 + NT2

NT1 + NT2 + NT2

Concrete vs. Abstract Syntax





Abstract Syntax

Types of Grammars

- **Grammars can be classified into types**
  - **Type is based on form of rules**
  - **Different types -> parsing is a harder / easier computation**

Types of Grammars

- **Context Free Grammars:**
  - **Every rule has a single nonterminal on the left-hand side: <A> =>…**
  - **Disallowed: <X> <A> => <X> a**
- **Regular Grammars:**
  - **Rules all take the forms:**
    <A> => c or <A> => <B> c ***(left-linear)***
  - **Or rules all take the forms:**
    <A> => c or <A> => c <B> ***(right-linear)***
  - **Disallowed: S => a S b**
  - **Cannot generate the language { $a^n b^n$ | n = 1,2,3, ... }**

Types of Grammars

- **Context Free Grammars (CFGs) are used to specify the overall structure of a programming language:**
  - **if/then/else, ...**
  - **brackets: ( ), { }, begin/end, ...**
- **Regular Grammars (RGs) are used to specify the structure of tokens:**
  - **identifiers, numbers, keywords, ...**
- **RGs are a subset of CFGs**

Extended BNF (EBNF)

A language for defining the grammar of a language.

Write nonterminals as usual. (Variant: Write them with initial capital letters, or using a different font.)

Use additional *metasymbols*, as shortcuts:

- {…} means repeat the enclosed text zero or more times
- […] means the enclosed text is optional
- (…) is used for grouping, usually with the alternation symbol, e.g., (… | ...).

If { }, [ ], or ( ) are used as terminal symbols in the language being defined, then they must be quoted. (Variant: They must be underlined.)

Extended BNF (EBNF)

Examples:

<expr> ::= <term> { (+ | -) <term> }

<term> ::= <factor> { (* | /) <factor> }

<factor> ::= <var> | <num> | ‘(’ <expr> ‘)’

<if-stmt> ::= if <expr> then <stmt> [else <stmt>]

<identifier> ::= <letter> { (<letter> | <digit>) }

Formal Language Theory

- **Offers a way to describe computation problems formulated as language recognition problems**
  - **Enables proofs of relative difficulty of certain computational problems**
- **Provides a mechanism to aid description of programming language constructs**
  - **Regular expressions ~ PL tokens (e.g. real numbers, keywords)**
  - **Context-free grammars ~ PL statements**

Specifying a Language

- Formalisms for specifying a language
  - Grammars
  - Regular Expressions
  - Automata

Regular Expressions

- Formalism for describing simple PL constructs
  - reserved words
  - identifiers
  - numbers
- Simplest sort of grammatical structure
- Defined recursively
- Actually are expressions on languages, ie on sets of strings

Regular Expressions

	RE Notation	**Language**
	an empty RE	**{ }**
symbol a	**a**	**{a}**
null symbol	**ε**	**{ε}**
R,S regular exprs	**R | S**	$L_R \cup L_S$
	a|b (alternation)	***{a,b}***
R,S regular exprs	**RS**	$L_R L_S$
	ab (concatenation)	***{ab}***

Regular Expressions

RE Notation	Language
a	{a}
b	{b}
ab	{ab}
a | b	{a, b}
ab | ac	{ab, ac}
(a | b)(c | d)	{ac, ad, bc, bd}
(abc | ε) d	{abcd, d}

Regular Expressions

	RE Notation	Language
R regular expr	R^*	$\{\varepsilon\} \cup L_R \cup L_R L_R \cup L_R L_R L_R \ldots$
	a^*	*$\{\varepsilon, a, aa, aaa, \ldots\}$*
R regular expr	R^+	$L_R \cup L_R L_R \cup L_R L_R L_R \ldots$
	a^+	*$\{a, aa, aaa, \ldots\}$*

Note: $\varepsilon\, a = a\, \varepsilon = a$

Precedence is + * concatenation |

high to low

(all are left associative operators)

Regular Expressions

RE Notation	Language
a*	{ε, a, aa, aaa, …}
ab*	{a, ab, abb, abbb, …}
(ab)*	{ε, ab, abab, ababab, …}
(a | b)*	{ε, a, b, aa, ab, ba, bb, …}
a+	{a, aa, aaa, …}
ab+	{ab, abb, abbb, …}