Language Description: Syntactic Structure

Expression Notations

Abstract Syntax Trees

Lexical Syntax

Context-Free Grammars

Grammars for Expressions

Variants of Grammars

Preface

- clear and complete descriptions
- formal syntax and informal semantics
- *syntax*: how programs in the language are built up
- *semantics*: what programs mean
- The same syntax has different semantics in different parts of the world.
  - Example: *DD/DD/DDDD*
    - 01/02/2001
    - January 2, 2001 or February 1, 2001 ?

Preface

- abstract syntax: captures intent, independent of notation
  - Example:
    - a+b, (+ a b), ADD a TO b
    - intent: apply the operator + to the operands a and b
    - written representation is different in each case
    - abstract syntax is the same

2.1 Expression Notations

- Programming languages use a mix of *infix*, *prefix*, and *postfix* notations.
  - Example:
    - infix: a+b
    - prefix: +ab
    - postfix: ab+
- An expression can be enclosed within parentheses without affecting its value.
  - expression *E* has the same value as (*E*)
- *parenthesis-free*
  - *prefix*
  - *postfix*

2.1 Expression Notations

- *Prefix* Notation
  - a constant or a variable: itself
  - an operator **op** to subexpressions *E*1 and *E*2:

    op *E*1 *E*2
  - advantage: easy to decode during a left-to-right scan of an expression
  - Example:
    - sum of *x* and *y*: + *x* *y*
    - product of + *x* *y* and *z*: * + *x* *y* *z*
    - * + 20 30 60
    - * 20 + 30 60

2.1 Expression Notations

- *Prefix* Notation (cont)
  - *arity:* the number of operands of an operator
  - **op**^{k} of arity k>=0 to E_1, E_2, …, E_k: **op**^{k} E_1 E_2 … E_k
  - during a left-to-right scan, the *i*th expression to the right of **op**^{k} is the *i*th operand of **op**^{k}, for 1<=*i*<=*k*
  - *read*(*x*) and *max*(*x*, *y*): a variant of prefix notation
    - allows operators to take a variable number of arguments
    - Example:
      - *write*(*root*), *write*(*root*, *a*, *b*, *c*)
  - Lisp: (*read x*), (*max x y*)

2.1 Expression Notations

- *Postfix* Notation
  - a constant or a variable: itself
  - an operator **op** to subexpressions *E*1 and *E*2:

    *E*1 *E*2 **op**
  - advantage: with the help of a *stack* data structure
  - Example:
    - sum of *x* and *y*: *x y* +
    - product of *x y* + and *z*: *x y* + *z* *
    - 20 30 + 60 *
    - 20 30 60 + *

2.1 Expression Notations

- *Infix* Notation: Precedence and Associativity
  - advantages: familiar, easy to read
  - decode *a+b*c*
    - sum of *a* and *b*c*?
    - product of *a+b* and *c*?
    - solved by *precedence* and *associativity*
  - an operator at a higher *precedence level* takes its operands before an operator at a lower precedence level
  - precedence: * > +
    - the operands of *: *b* and *c*
    - the operands of +: *a* and *b*c*

2.1 Expression Notations

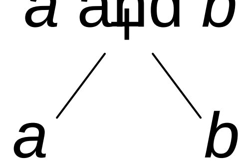
- *Infix* Notation (cont)
  - precedence:
    - *, / > +, -
    - * and /: at the same level
    - + and -: at the same level
  - with the same precedence: *from left to right*
  - 4 – 2 – 1 = (4 – 2) – 1 = 2 – 1 = 1
  - left associative
    - if subexpressions containing multiple occurrences of the operators are grouped *from left to right*
  - +, -, *, /: left associative
  - $b * b - 4 * a * c = (b * b) - ((4 * a) * c)$

2.1 Expression Notations

- *Infix* Notation (cont)
  - *right associative*: grouped *from right to left*
  - Example: exponentiation
    - $2^{3^4} = 2^{\left(3^4\right)} = 2^{81}$
- *Mixfix* Notation
  - operations specified by a combination of symbols do not fit neatly into the prefix, infix, postfix notations
  - Example: **if** *a*>*b* **then** *a* **else** *b*
  - when symbols or keywords appear interspersed with the components of an expression, the operation will be said to be in *mixfix* notation

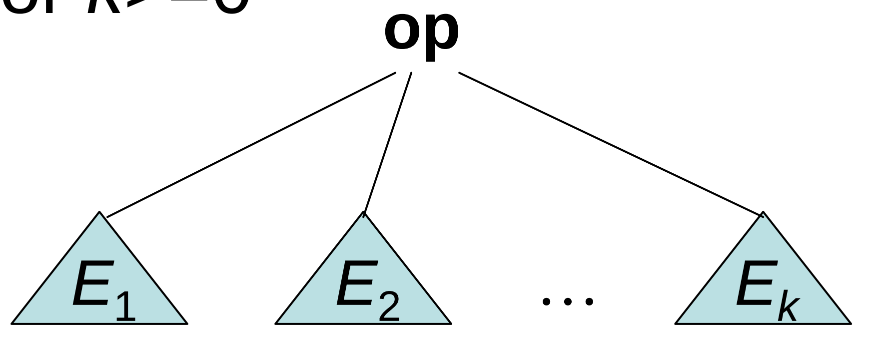
2.2 Abstract Syntax Trees

- *abstract syntax* of a language
  - identifies the meaningful components of each construct in the language
- prefix: + *a b*, infix: *a* + *b*, postfix: *a b* +
  - all have the same meaningful components
    - the operator: +
    - the subexpressions: *a* and *b*
  - tree representation:



2.2 Abstract Syntax Trees

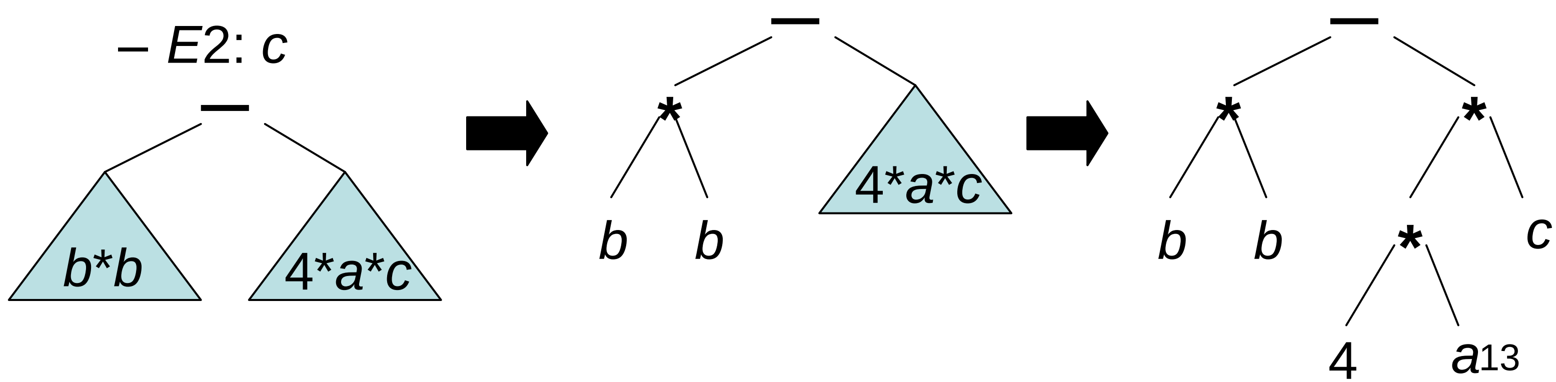
- Tree Representation of Expressions
  - applying an operator **op** to operands E_1, E_2, …, E_k, for k>=0



  - if an expression is a constant or a variable, then its tree consists of a leaf

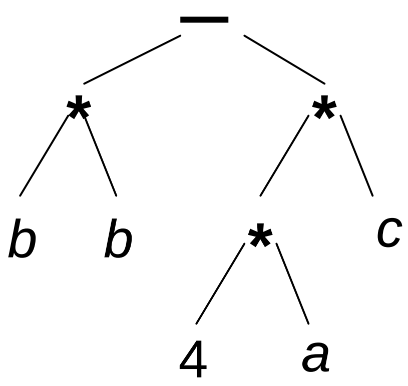
2.2 Abstract Syntax Trees

- Tree Representation of Expressions (cont)
  - Example: *b***b*-4**a***c*
    - expression is of the form: *E*1-*E*2
    - *E*1: *b***b*
    - *E*2: 4**a***c*
      - the subexpression has the form *E*1**E*2
      - *E*1: 4**a*
      - *E*2: *c*

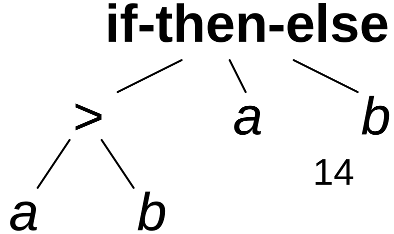


2.2 Abstract Syntax Trees

- Abstract Syntax Tree
  - show the operator/operand structure of an expression
  - show the syntactic structure of an expression independent of the notation
  - Example:
    - Prefix: – * *b b* * * 4 *a c*
    - Infix: *b* * *b* – 4 * *a* * *c*
    - Postfix: *b b* * 4 *a* * *c* * –



  - can be extended to other constructs by making up suitable operators for the constructs
    - Example: **if** *a*>*b* **then** *a* **else** *b*



2.3 Lexical Syntax

- Tokens and Spellings
  - the syntax of a programming language is specified in terms of units called *tokens* or *terminals*
  - *lexical syntax*
    - specifies the correspondence between the written *representation* of the language and the *tokens* or terminals in a grammar for the language
  - *keywords*
    - alphabetic character sequences that are treated as units in a language
    - Example: **if** and **while** are keywords in both Pascal and C
  - *Reserved Words*
    - keywords that cannot be used as names

2.3 Lexical Syntax

- Tokens and Spellings (cont)
  - *Spelling*
    - the actual character sequence used to write down an *occurrence* of a token is called the *spelling* of that occurrence
    - Example: the keyword **while** has spelling `while`
  - Using token **name** for names and token **number** for integers, the character sequence

    b * b – 4 * a * c

    is represented by the token sequence

    $\textbf{name}_b * \textbf{name}_b - \textbf{number}_4 * \textbf{name}_a * \textbf{name}_a$
  - *White space* in the form of *blank*, *tab*, and *newline* characters can typically be inserted between tokens without changing the meaning of a program

2.3 Lexical Syntax

- Tokens and Spellings (cont)
  - *comments* between tokens are ignored
    - (* and *) in Pascal
    - /* and */ in C
  - the most complex rules in a lexical syntax are typically the ones describing the syntax of *real numbers*
  - Example:

    314.E-2, 3.14, 0.314E+1, 0.314E1, .314E1

2.4 Context-Free Grammar

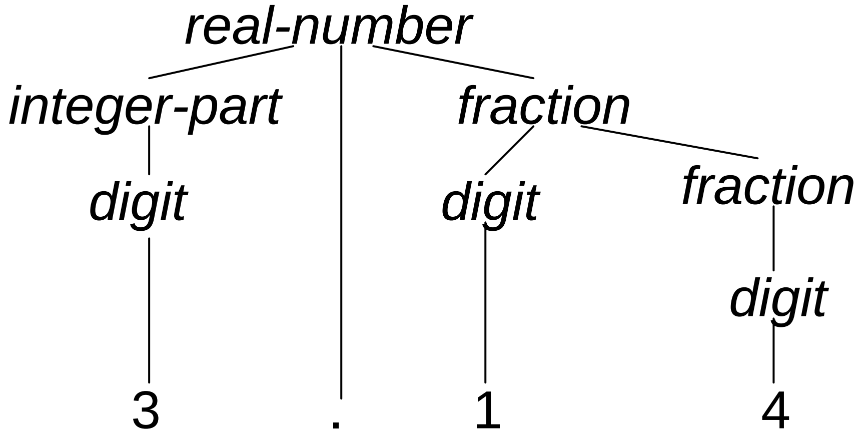
- *Context-Free Grammar*
  - a set of tokens or terminals
  - a set of nonterminals
  - a set of rules, productions
  - a starting nonterminal
- BNF: Backus-Naur Form

BNF rules for real numbers

<real-number>	::=	*<integer-part>* . *<fraction>*
<integer-part>	::=	*<digit>* | *<integer-part>* *<digit>*
<fraction>	::=	*<digit>* | *<digit>* *<fraction>*
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2.4 Context-Free Grammar

- *Parse Tree*
  - hierarchical structure
  - nonterminal
  - production
    - a rule that defines a nonterminal in terms of a sequence of terminals and nonterminal
    - each node in the parse tree is based on a production
  - Example: real number

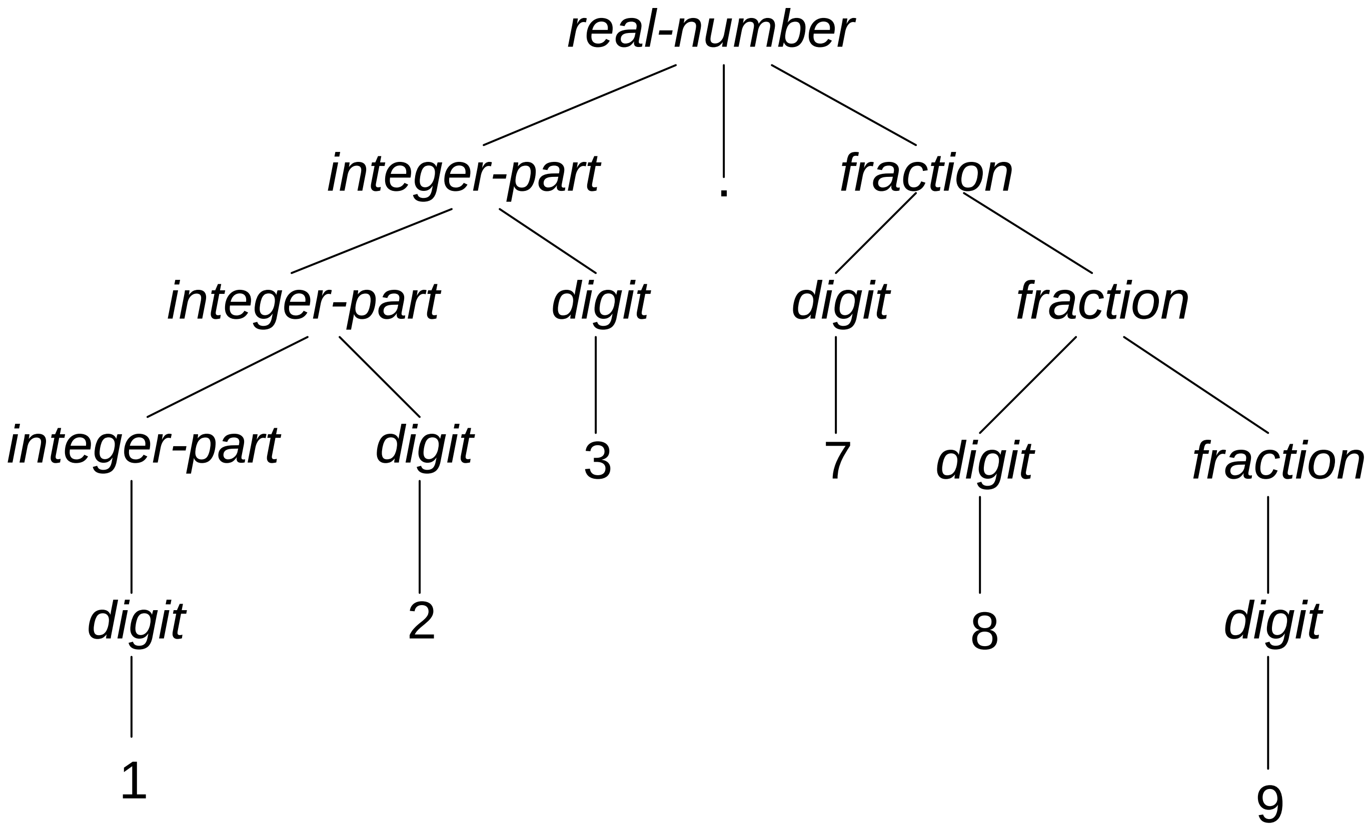


2.4 Context-Free Grammar

- *Parse Tree* (cont)
  - <*empty*>: an empty string of length 0
    - useful for specifying optional constructs
    - Example: 0 in 0.5
      - <*integer-part*> ::= <*empty*> | <*digit-sequence*>
  - each *leaf* is labeled with a terminal or <*empty*>
  - each *nonleaf* node is labeled with a nonterminal
  - the *root* is labeled with the starting nonterminal

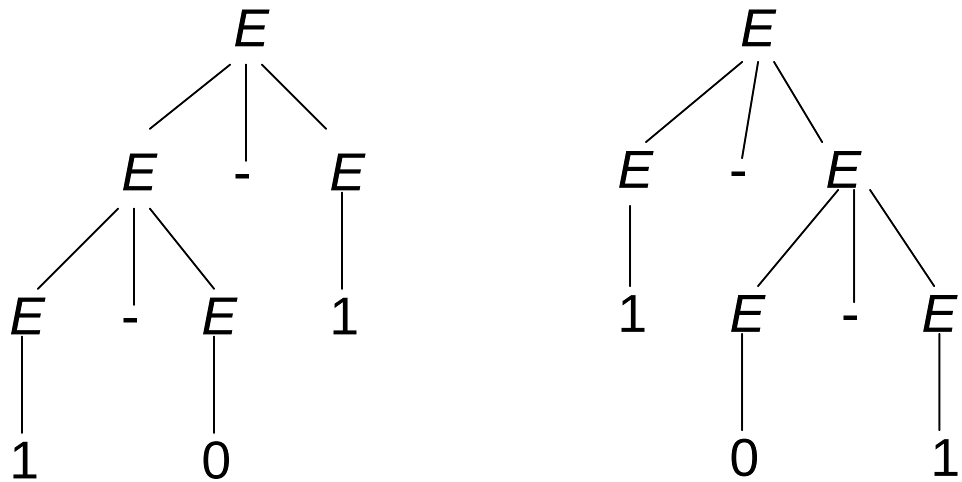
2.4 Context-Free Grammar

- parsing 123.789



2.4 Context-Free Grammar

- Syntactic Ambiguity
  - if some string in its language has *more than one* parse tree
  - Example: $E ::= E - E \mid 0 \mid 1$



2.4 Context-Free Grammar

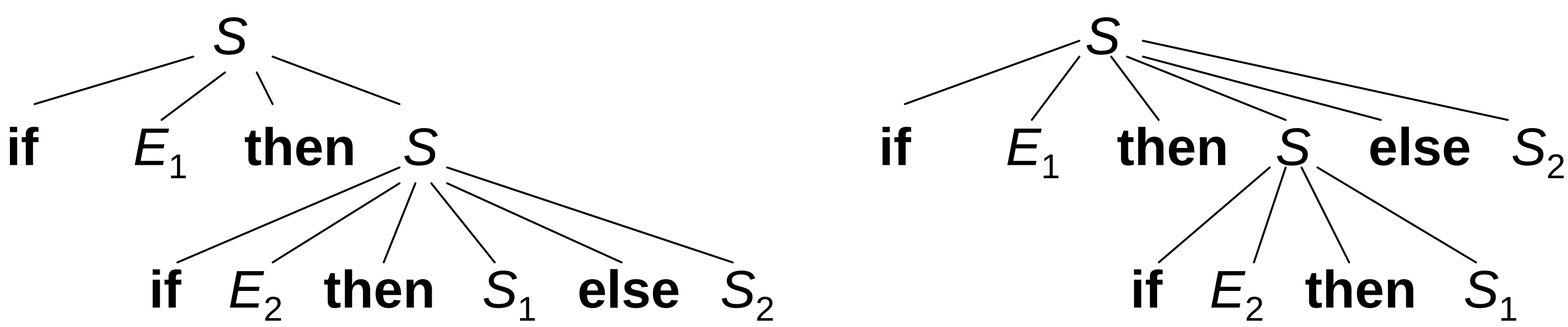
- *Dangling-Else Ambiguity*
  - production:

    S ::= **if** E **then** S

    S ::= **if** E **then** S **else** S

  - parsing:

    if E_1 **then if** E_2 **then** S_1 **else** S_2



resolved by matching an **else** with the nearest unmatched **if**

2.4 Context-Free Grammar

- *Derivations*
  - *top-down* parser
    - from the *root* of a parse tree toward the *leaves*
  - *bottom-up* parser
    - from the *leaves* of a parse tree toward the *root*
  - a derivation consists of a sequence of strings, beginning with the starting nonterminal
  - each successive string is obtained by *replacing a nonterminal by the right side of one of its production*
  - a derivation ends with a string consisting *entirely of terminals*

2.4 Context-Free Grammar

- *Derivations* (cont)
  - Example: parsing 21.89

bottom-up

real-number ::= *integer-part* . *fraction*
::= *integer-part digit* . *fraction*
::= *digit digit* . *fraction*
::= 2 *digit* . *fraction*
::= 2 1 . *fraction*
::= 2 1 . *digit fraction*
::= 2 1 . 8 *fraction*
::= 2 1 . 8 *digit*
::= 2 1 . 8 9

top-down

2.5 Grammars for Expressions

- Lists in Infix Expression
  - terms and factors
    - Example: a*b+c*d+e

      terms: a*b, c*d, e

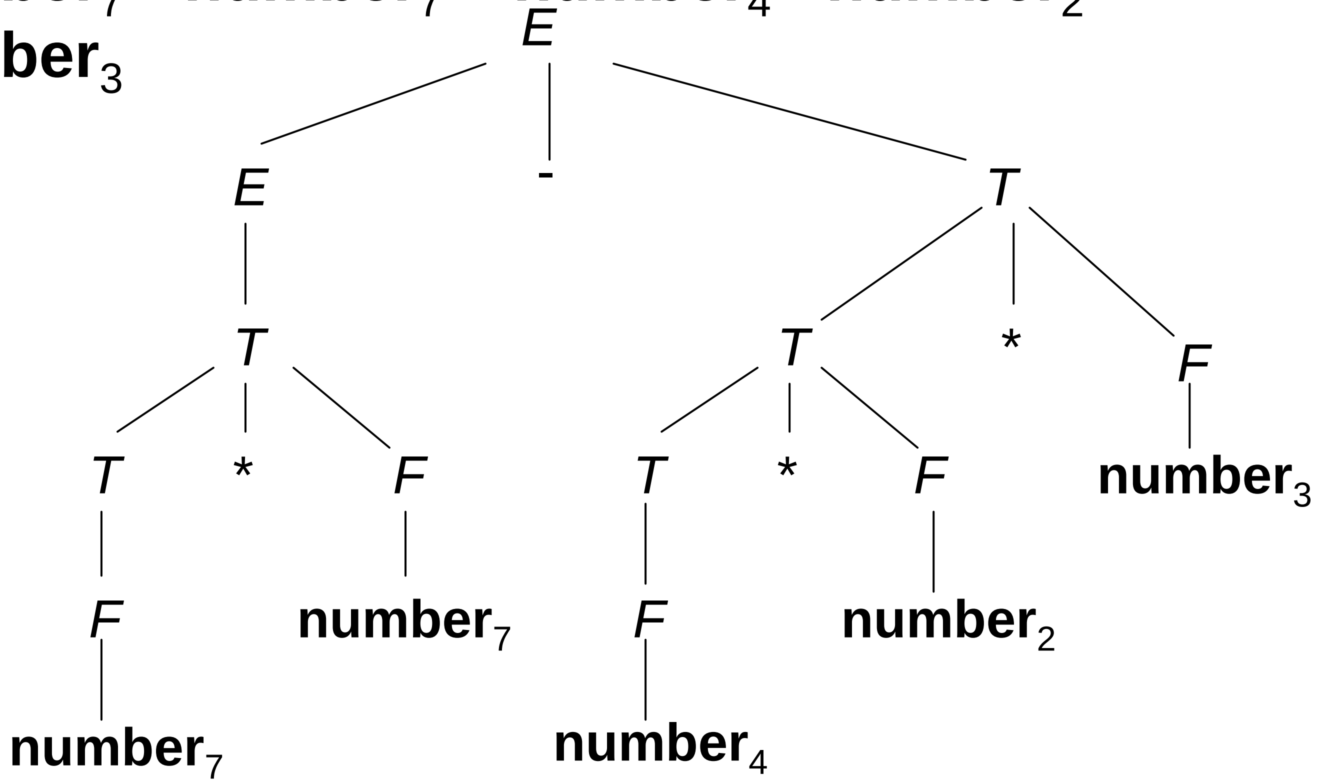
      factors: a, b for a*b
  - a grammar for arithmetic expressions

$$E ::= E + T \mid E - T \mid T$$
$$T ::= T * F \mid T \, / \, F \mid F$$
$$F ::= \textbf{number} \mid \textbf{name} \mid (E)$$

2.5 Grammars for Expressions

– parse tree:

- **number**$_7$ * **number**$_7$ – **number**$_4$ * **number**$_2$ * **number**$_3$

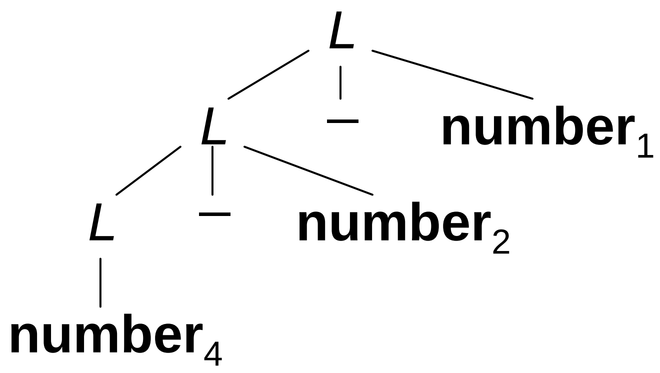


2.5 Grammars for Expressions

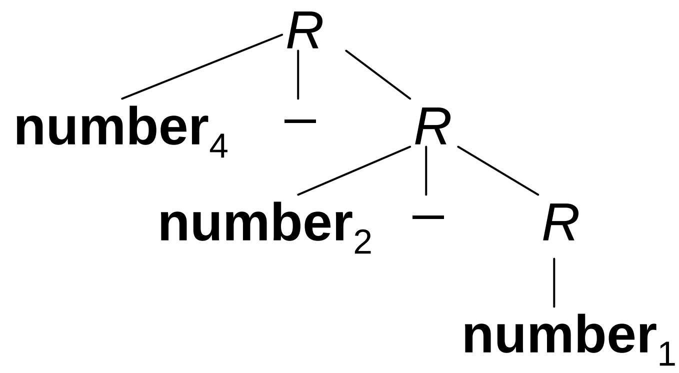
- Handling Associativity
  - – Example: 4 – 2 – 1

L ::= *L* + **number**
| *L* – **number**
| **number**

R ::= **number** + *R*
| **number** – *R*
| **number**



left associative



right associative

2.5 Grammars for Expressions

- Handling Associativity and Precedence

  A ::= E := A | E
  E ::= E + T | E – T | T
  T ::= T * F | T / F | F
  F ::= (E) | **name** | **number**

2.6 Variants of Grammars

- BNF

<*expression*> ::= <*expression*> + <*term*>
| <*expression*> - <*term*>
| <*term*>
<*term*> ::= <*term*> * <*factor*>
| <*term*> / <*factor*>
| <*factor*>
<*factor*> ::= **number**
| **name**
| (<*expression*>)

2.6 Variants of Grammars

- EBNF
  - extension of BNF
  - allows lists and optional elements to be specified
  - for convenience, not for additional capability
  - equivalent to BNF
  - { and } : zero or more
    - Example: { <*statements*> ; }

    - <*statement list*> ::= { <*statement*> ; }

      ≡

      <*statement list*> ::= <*empty*>
      | <*statement*> ; <*statement list*>

    - reduce the number of productions and nonterminal

2.6 Variants of Grammars

- EBNF (cont)
  - [and] : optional
    - Example:
    - <*real number*> ::= [<*integer part*>] . <*fraction*>

      ≡

      <*real number*> ::= <*integer part*> . <*fraction*>
| . <*fraction*>
  - | : choice
  - (and) : grouping
  - tokens & metasymbols
    - enclosing tokens within single quotes: ‘(‘

2.6 Variants of Grammars

- EBNF: <*expression*> ::= <*term*> { (+ | -) <*term*> }
  <*term*> ::= <*factor*> { (* | /) <*factor*> }
  <*factor*> ::= ‘(‘ <*expression*> ‘)’ | **name** | **number**

- BNF: <*expression*> ::= <*expression*> + <*term*>
  | <*expression*> - <*term*>
  | <*term*>
  <*term*> ::= <*term*> * <*factor*>
  | <*term*> / <*factor*>
  | <*factor*>
  <*factor*> ::= **number**
  | **name**
  | (<*expression*>)

2.6 Variants of Grammars

- Syntax Chart (syntax graph, syntax diagram)
  - graphical notation
  - equivalent BNF

2.6 Variants of Grammars

- Syntax Chart