Fundamantals

Syntax of Programming Languages

Syntax and Semantics

- Syntax
  - The symbols and rules to write legal programs
- Semantics
  - The meaning of legal programs
- Programming language implementation
  - Syntax –> semantics (computer actions)
- Example: date specification
  - Syntax
    - date ::= dd/dd/dddd   d = 0|1|2|3|4|5|6|7|8|9
  - Semantics
    - 01/02/2005 => Jan 02, 2005 (or Feb 01,2005) ?

Describing Language Syntax

- Lexical grammar
  - Spelling of words (tokens/terminals)
    - Numbers, strings, names, keywords(if, while, for, else)…
  - Formal description: regular expressions
    - Describe the composition of words
    - [a-zA-Z_][a-zA-Z0-9_]*,   [0-9]+, "while"
- Context-free grammar
  - Formal description: BNF (Backus-Naur Form)
  - Rules to compose programs from tokens
    - forStmt: "for" "(" exp ";" exp ";" exp")" stmt
  - Support variables and recursion, but cannot express context sensitive information
    - recursion does not have parameters/memories
- Why formal description?
  - Avoid miscommunication
  - Automated generation of parsers (syntax analyzers)

BNF: Expressing Context-Free Grammars

- Each BNF includes
  - A set of terminals: the words/tokens of the language
  - A set of non-terminals: variables that could be replaced with different sequences of terminals
  - A set of productions
    - Rules identifying the structure of each non-terminal
    - Each production has format A ::= B where
      - A is a single non-terminal
      - B is a sequence of terminals and non-terminals
  - A start non-terminal: the top-level syntax of the language
- Example: BNF for expressions

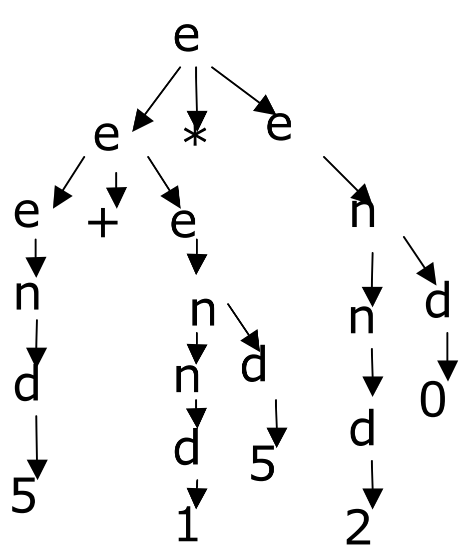
```
e ::= n | e+e | e–e | e * e | e / e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

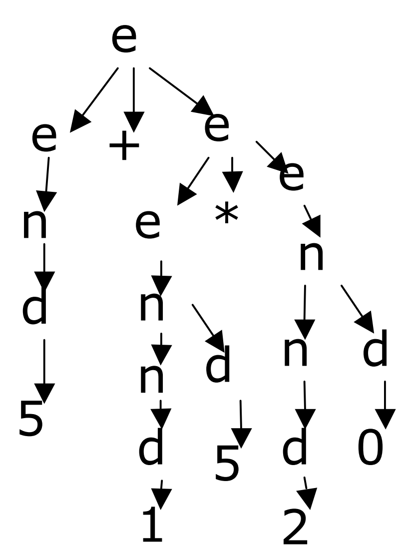
  - Non-terminals: e, n, d; start non-terminal: e
  - Terminals: 0,1,2,3,4,5,6,7,8,9

Derivations and Parse Trees

- Derivation: deriving an input string from the start non-terminal
  - Top-down replacement of non-terminals following production rules
  - One or more derivations for each valid program
- Derivations for 5 + 15 * 20
  - e=>**e*e**=>**e+e*e**=>n+e*e=>d+e*e=>**5+e*e**=>5+n*e=>5+nd*e=>5+dd*e=>5+1d*e=>**5+15*e**=>…=>**5+15*20**
  - E=>**e+e**=>…=>**5+e**=>**5+e*e**=>…=>**5+15*e**=>…=>**5+15*20**

Parse trees: graphical (tree) representation of derivations





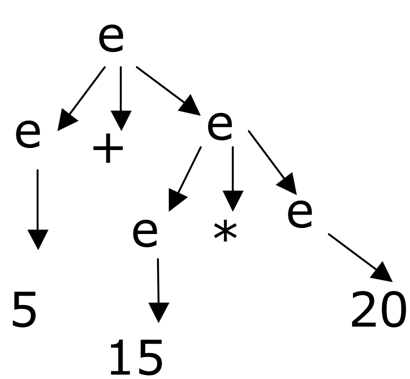
Parsing And Parse Trees

- Parsing (checking syntactical correctness)
  - Given an input program, does it have correct syntax?
    - Answer: can a parse tree be constructed for the program?
    - Top-down and bottom-up parsers
- A parse tree represents a syntactically correct program
  - To regenerate a program, read terminals from left to right
  - Interior nodes represent the structure of the input program
- A parse tree of each program satisfies
  - Each leaf node represent a terminal
  - Each non-leaf node represent a non-terminal
  - The children of each non-leaf node A, from left to right, form the right-side of a production rule for A (with A at left-side)
  - The root of the parse tree is the starting non-terminal

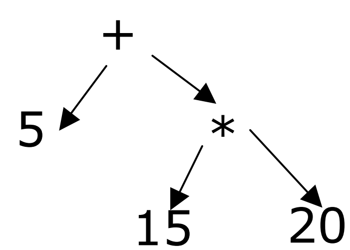
Concrete Vs. Abstract Syntax

- Concrete syntax: the syntax that programmers write
  - Example: different notations of expressions
    - Prefix + 5 * 15 20
    - Infix 5 + 15 * 20
    - Postfix 5 15 20 * +
- Abstract syntax: the internal structure of the input program recognized by compilers/interpreters
  - Identifies only the meaningful components
    - What is the operation and which are the operands ?

Parse Tree for 5+15*20

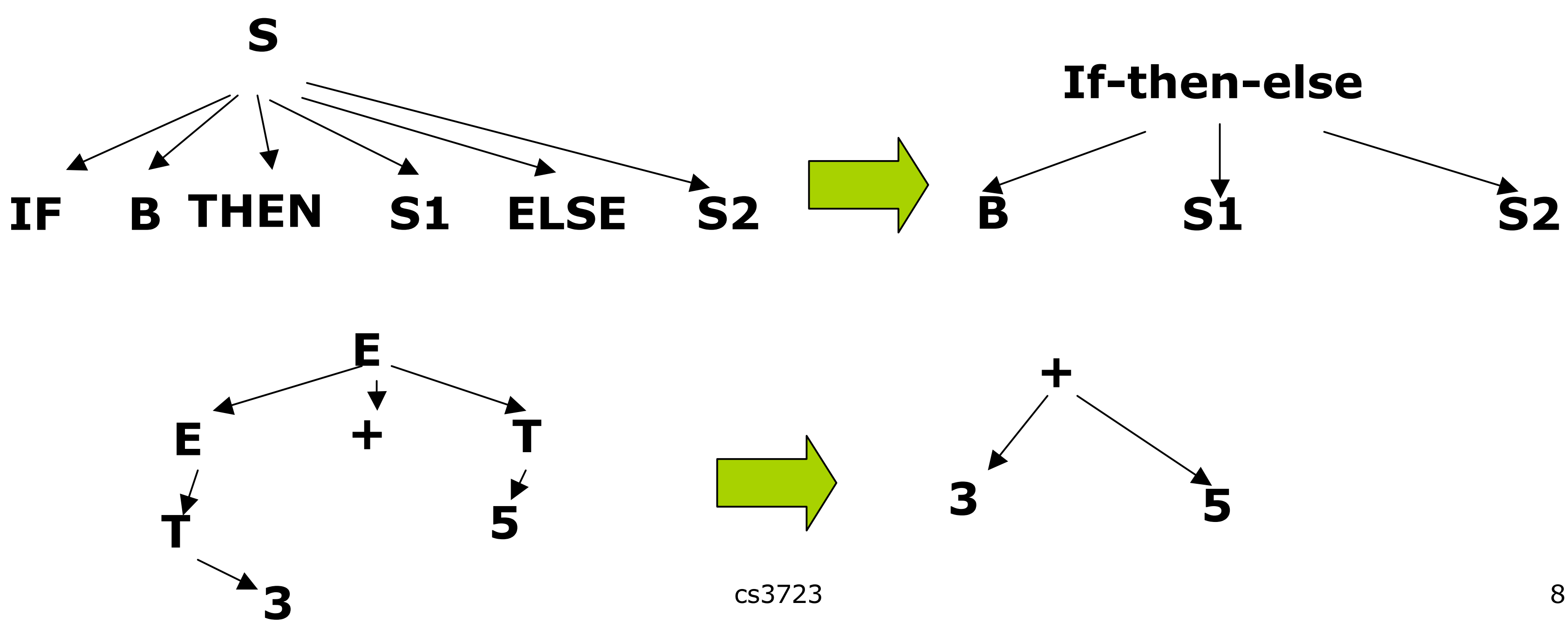


Abstract Syntax Tree for 5 + 15 * 20



Abstract Syntax Trees

- Condensed form of parse tree: internal representation of programs by compilers/interpreters
  - Operators and keywords do not appear as leaves
    - They define the meaning of the interior (parent) node
  - Chains of single productions may be collapsed



Exercises
Building Parse Trees and AST

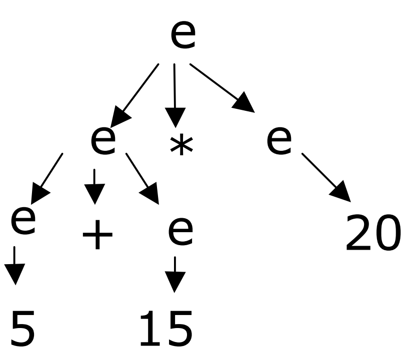
- Grammar for expressions
  - e ::= n | e+e | e–e | e * e | e / e | (e)
  - What are the terminals and non-terminals?
  - Write parse trees and ASTs for 1-1*1 and 1*(2-3+1)
- Grammar: e ::= 0 | 1 | 0e | 1e
  - What language does the grammar describe?
  - Write parse trees and ASTs for 011100

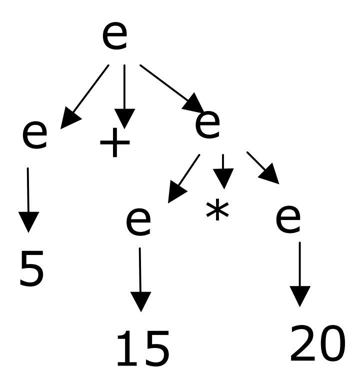
- Steps for building parse trees
  - Write down the start non-terminal
  - Pick a non-terminal in the tree, pick a production, replace the non-terminal by expanding the subtree
    - **Which production to pick? --- the one that describes the structure of the current input for the given non-terminal**

- Parse tree => AST
  - Replace each production with an operator
  - Remove useless tokens (those that don't have values)
  - Collapse chains of single productions

Ambiguous Grammars

- A grammar is syntactically ambiguous if
  - some program has multiple parse trees
    - Multiple choices of production rules during derivation
    - Result in multiple ASTs
- Consequence of multiple parse trees
  - Parse trees/ASTs are used to interpret programs
    - Multiple ways to interpret a program





Rewrite ambiguous Grammars

- Solution1: introduce precedence and associativity rules to dictate the choices of applying production rules
  - Original grammar: e ::= n | e+e | e–e | e * e | e / e
  - Precedence and associativity
    - * / >> + - all operators are left associative
  - Derivation for n+n*n
    - e=>e+e=>n+e=>n+e*e=>n+n*e=>n+n*n

- Solution2: rewrite production rules by introducing additional non-terminals
  - Alternative grammar E ::= E + T | E – T | T
    T ::= T * F | T / F | F
    F ::= n
  - Derivation for n + n * n
    - E=>E+T=>T+T=>F+T=>n+T=>n+T*F=>n+F*F=>n+n*F=>n+n*n
  - How to modify the grammar if
    - + and - has high precedence than * and /
    - All operators are right associative

Writing CFGs

- Give a CFG to describe the set of strings over {(,),[,]} which form balanced parentheses/brackets. For example
  - "()", "()()", "(()())", and "([]()[])" are in the language
  - ")(", "(()", and "([" are not in the language
- If your grammar ambiguous? If yes, prove it by giving two different parse trees for a single input. Rewrite it to be non-ambiguous

Here we are practicing programming using BNF

- **Fundamental concepts: variables (non-terminals) and recursion**
  - Define a clear meaning (in English) for each non-terminal
  - Use recursion to implement the meaning
    - Need to know how to describe a sequence of items and how to ensure an item appears some number of times
- **Ambiguity: introduce a new non-terminal for each precedence**
  - Recursive on the left if left-associative
  - Recursive on the right if right-associative

Additional exercises

- Give a context-free grammar for a small graph description language
  - Terminals: digits(`0',`1',...,`9'),`(', `)', `;' and `->'
  - Each node of the graph is represented by an integer number,
  - Each edge is represented by a pair of nodes connected with `->'
    - eg., 3->4 is an edge from node `3' to node `4'
  - Each graph description is a sequence of edges
    - Eg. (1->2; 2->5; 5->1)
- Write a parse tree and an abstract syntax tree for (1->2; 2->5; 5->1)

Additional Exercises (practice on your own)

- Give a CFG to describe the set of symmetric strings over {a,b}
- Give a CFG to describe the set of strings over {a,b} that have the same numbers of a's and b's?
- Give a CFG for the syntax of regular expressions over {0,1} . For example
  - "0|1", "0*", (01|10)* are in the languages
  - "0|" and "*0" are not in the language
- Can you give a CFG to describe the set of strings that have the format xx, where x is an arbitrary string over {a,b}