Backus naur form questions 2017 pdf free pdf

Backus naur form questions 2017 pdf free pdf

You need to enable JavaScript to access Isaac Computer Science. $\begingroup$ I'm pretty new to CFG, and I've tried out some samples as well. As of now I'm trying this for-loop to write it in CFG using BNF: Start{ int z = 20; for(int b = 0; b < 5; b++) { //some statement } } Attempted BNF: ::= Start ::= | ; | | ::= ::= char | short | int | long | float | double | string ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | x | y | z ::= = ::= ::= | | | ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ::= + | - | * | / The above CFG isn't complete and I wanted to make sure that I'm doing it correctly. Am I heading towards the right path? Should it be written as above or have I missed anything? Any help could be appreciated. $\endgroup$ 4 You can't perform that action at this time. You signed in with another tab or window. Reload to refresh your session. You signed out in another tab or window. Reload to refresh your session. Something went wrong. Wait a moment and try again. 1 Syntax and Backus Naur Form4/23/2017 Syntax and Backus Naur Form How BNF and EBNF describe the grammar of a language. Parse trees, abstract syntax trees, and alternatives to BNF. 2 BNF learning goals Know the syntax of BNF and EBNFBe able to read and write BNF/EBNF rules Understand... how the order of productions effects precedence of operations effect of left recursion and right recursion on association (left-right order of evaluation) what is ambiguity and its consequences 3 Backus Naur Form Backus Naur Form (BNF): a standard notation for expressing syntax as a set of grammar rules. BNF was developed by Noam Chomsky, John Backus, and Peter Naur. First used to describe Algol. BNF can describe any context-free grammar. Fortunately, computer languages are mostly context-free. Computer languages remove non-context-free meaning by either (a) defining more grammar rules or (b) pushing the problem off to the semantic analysis phase. 4 Scanning and Parsing source file sum = x1 + x2; input stream sum = x1Regular expressions define tokens Scanner tokens BNF rules define grammar elements Parser sum = + x1 x2 parse tree 5 A Context-Free GrammarA grammar is context-free if all the syntax rules apply regardless of the symbols before or after (the context). Example: (1) sentence => noun-phrase verb-phrase . (2) noun-phrase => article noun (3) article => a | the (4) noun => boy | girl | cat | dog (5) verb-phrase => verb noun-phrase (6) verb => sees | pets | bites Terminal symbols: 'a' 'the' 'boy' 'girl' 'sees' 'pets' 'bites' 6 A Context-Free GrammarA sentence that matches the productions (1) - (6) is valid. a girl sees a boy a girl sees a girl a girl sees the dog the dog pets the girl a boy bites the dog a dog pets the boy ... To eliminate unwanted sentences without imposing context sensitive grammar, specify semantic rules: "a boy may not bite a dog" 7 Backus Naur Form Grammar Rules or Productions: define symbols.assignment_stmt ::= id = expression ; The nonterminal symbol being defined. The definition (production) Nonterminal Symbols: anything that is defined on the left-side of some production. Terminal Symbols: things that are not defined by productions. They can be literals, symbols, and other lexemes of the language defined by lexical rules. Identifiers: id ::= [A-Za-z_]\w* Delimiters: ; Operators: = * / % 8 Backus Naur Form (2) Different notations (same meaning):assignment_stmt ::= id = expression + term => = + AssignmentStmt id = expression + term ::=, =>, mean "consists of" or "defined as" Alternatives (" | "): Concatenation: expression => expression + term | expression - term | term number => DIGIT number | DIGIT 9 Backus Naur Form (2) Another way to write alternatives:Null symbol,: e used to allow a production to match nothing. Example: a variable is an identifier followed by an optional subscript expression => expression + term => expression - term => term variable => identifier subscript subscript => [expression] | e 10 Example: arithmetic grammarHere is a grammar for assignment with arithmetic operations, e.g. y = (2*x + 5)*x - 7; assignment => ID = expression ; expression => expression + term | expression - term | term term => term * factor | term / factor | factor factor => (expression) | ID | NUMBER Q: What are the non-terminal symbols? Q: What are the terminal symbols? 11 What Do You Want To Produce???The parser must be told what is a valid input. This is done be specifying one top level production, called the start symbol. Usually the start symbol is the first production. The parser will try to "reduce" the input to the start symbol. Q: What is the start symbol in the previous example? 12 Applying the Grammar Rules (1)Apply the rules to the input. z = (2*x + 5)*y - 7; z = (2*x + 5)*y - 7; Source: Scanner tokens: ID ASSIGNOP GROUP NUMBER OP ID OP NUMBER GROUP OP ID OP NUMBER DELIM values: z = (* x) * y ; Parser 13 Applying the Grammar Rules (2)tokens: ID = (NUMBER * ID + NUMBER) * ID - NUMBER ; parser: ID read (shift) first token factor reduce factor = shift FAIL: Can't match any rules (reduce) Backtrack and try again ID = (NUMBER ... shift ID = (factor ... reduce ID = (term * ... sh/reduce ID = (term * ID ... shift ID = (term * factor ... reduce ID = (term ... reduce ID = (term shift ID = (expression + NUMBER ... reduce/sh ID = (expression + factor ... reduce ID = (expression + term ... reduce Action 14 Applying the Grammar Rules (3)tokens: ID = (NUMBER * ID + NUMBER) * ID -NUMBER; input: ID = (expression reduce ID = (expression) ... shift ID = factor reduce ID = factor * shift ID = term * ID reduce/sh ID = term * factor ... reduce ID = term reduce ID = term shift ID = expression reduce ID = expression - NUMBER ... shift ID = expression - factor ... reduce ID = expression - term reduce ID = expression ; shift assignment reduce SUCCESS!! Start Symbol 15 Applying the Grammar Rules (4)The parser creates a parse tree from the input: assignment ID = expression z term expression - factor term Some productions are omitted to reduce space factor NUMBER factor * 7 ID expression) (y + term factor NUMBER * ID NUMBER 2 x 5 16 Terminology (review) Grammar rules are called productions ... since they "produce" the language. Left-hand sides of productions are non-terminal symbols (nonterminals) or structure names. Tokens (which are not defined by syntax rules) are terminal symbols. Metasymbols of BNF are ::= (or => or), One nonterminal is designated as the start symbol. Usually the rule for producing the start symbol is written first. 17 BNF rules can be recursiveexpr => expr + term | expr - term | term term => term * factor | term / factor | factor factor => (expr) | ID | NUMBER where the tokens are: NUMBER := [0-9]+ ID := [A-Za-z_][A-Za-z_0-9]* 18 Uses of Recursion repetition expr => expr + term=> expr + term + term => expr + term + term + term => term term + term complicated expressions expr => term => term * factor => factor * factor => (expr) * factor => (expr + term) * factor => ... 19 Parse Trees The parser creates a data structure representing how the input is matched to grammar rules. Usually as a tree. Example: x = y*12 - z assignment expr ID = x - term expr term factor * ID term factor z factor NUMBER 12 ID y 20 Parse Tree Structure The start symbol is the root node of the tree.This represents the entire input being parsed. Each replacement in a derivation (parse) using a grammar rule corresponds to a node and its children. Example: term term + factor term factor term + 21 Example: Parse Tree for (2+3)*4expr expr term term * factor ( ) factor expr + number expr term term 4 factor factor number number 3 2 22 Abstract Syntax Trees expr expr term factor ID: x ID: xParse trees are very detailed: every step in a derivation is a node. After the parsing phase is done, the details of derivation are not needed for later phases. Semantic Analyzer removes intermediate productions to create an (abstract) syntax tree. expr term expr term factor Parse Tree: Abstract Syntax Tree: factor ID: x ID: x 23 Example: Abstract Syntax Tree for (2+3)*4 24 Syntax-directed semanticsThe parse tree or abstract syntax tree structure corresponds to the computation to be performed. To perform the computation, traverse the tree in order. Q: what does "in order traversal" of a tree mean? + - * 5 3 4 2 25 BNF and Operator PrecedenceThe order of productions affects the order of computations. Consider this BNF for arithmetic: assignment => id = expression ; expression => id + expression | id - expression | id * expression | id / expression | id | number Does this describe standard arithmetic? 26 Lets check the order of operationsExample: sum = x + y + z; Rule Matching Process: assignment id = expression id = id + expression id = id + id + expression id = id + id + id sum = x + y + z sum = expression id + expression x id expression + Result: sum = x + (y + z) id y Not quite correct: this is right associative z 27 Lets check the order of operationsExample: sum = x - y - z; Rule Matching Process: assignment id = expression id = id - expression id = id - id - expression id = id - id - id sum = x - y - z sum = expression id - expression x id expression - Result: sum = x - (y - z) id y Wrong! Subtraction is not right associative 28 The right-associative problemProblem is that previous rule was right recursive. This produces a parse tree that is right associative. expression => id + expression | id - expression | id * expression Solution is to define the rule to be left recursive. This produces a parse tree that is left associative. expression => expression + id | expression - id | ... 29 Revised Grammar (1) Grammar rule should use left recursion to get left association of the operators assignment => id = expression ; expression => expression + id | expression - id | expression * id | expression / id | id | number Does this work? 30 Check the order of operationsExample: sum = x - y - z; Rule Matching Process: sum = expression sum = expression - id sum = expression - z sum = expression - id - z sum = expression - y - z sum = id - y z sum = x - y z sum = expression - expression id id - id z Result: sum = (x - y) - z x y 31 Check the precedence of operationsExample: sum = x + y * z; Rule Matching Process: sum = expression sum = expression * id sum = expression * z sum = expression + id * z sum = expression + y * z sum = id + y * z sum = x - y z sum = expression * expression + id id id z Result: sum = (x + y) * z X x y 32 Revised Grammar (2) To achieve precedence of operators, we need to define more rules (just like in math)... assignment => id = expression ; expression => expression + term | expression - term | term term => term * factor | term / factor | factor factor => (expression) | id | number 33 Check the precedence of operationsExample: sum = x + y * z; Rule Matching Process: sum = expression sum = expression + term sum = term + term sum = factor + term sum = id + term sum = x + term sum = x + term * factor ... sum = expression + expression term term * factor term factor Result: sum = x + (y * z) . id id y z x 34 Check another case Example: sum = x / y - z; - term id Result: factorsum = expression - expression term term factor / id Result: sum = (x/y) - z factor term . . z x y 35 Precedence: lower is higherRules that are lower in the "cascade" of productions are matched closer to the terminal symbols. Therefore, they are matched earlier. Rule: rules lower in the cascade have higher precedence. expression => expression + term | expression - term | term term => term * factor | term / factor | factor factor => (expression) | id | number Rules lower in cascade are closer to the terminal symbols, so they have higher precedence. 36 Exercise 1: Show the parse tree for y = 2 * (a + b);assignment => id = expression ; expression => expression + term | expression - term | term term => term * factor | term / factor | factor factor => (expression) | id | number 37 Exercise 1: Show the parse tree for y = 2 * (a + b); * factor numberid = expression term * factor factor (expression) number expression + term 2 term factor factor id id b a 38 Exercise 2: Show the parse trees for r1 = x + b / 2 * a; r2 = x + b /(2 * a); assignment => id = expression ; expression => expression + term | expression - term | term term => term * factor | term / factor | factor factor => (expression) | id | number 39 Ambiguity A grammar is ambiguous if there is more than one parse tree for a valid sentence. Example: expr => expr + expr | expr * expr | id | number How would you parse x + y * z using this rule? 40 Example of Ambiguity Grammar Rules:expr => expr + expr | expr expr | (expr) | NUMBER Expression: * 4 Two possible parse trees: 41 Another Example of AmbiguityGrammar rules: expr => expr + expr | expr - expr | (expr) | NUMBER Expression: Parse trees: 42 Ambiguity Ambiguity can lead to inconsistent implementations of a language. Ambiguity can cause infinite loops in some parsers. In yacc and bison the message: 5 shift/reduce conflicts (can be any number) can indicate ambiguity in the grammar rules Specification of a grammar should be unambiguous! 43 Ambiguity (2) How to resolve ambiguity:rewrite grammar rules to remove ambiguity add some additional requirement for parser, such as "always use the left-most match first" EBNF (later) helps remove ambiguity 44 Resolving ambiguitiesReplace multiple occurrences of the same nonterminal with a different nonterminal. Choose replacement that gives correct associativity: expr => expr + expr expr => expr + term Add new rules in order to achieve correct precedence: expr => expr + term | term term => term * factor | factor factor => (expr) | ID | NUMBER In yacc/bison you can specify associativity: %left '+' Rules lower in cascade are closer to the terminal symbols. 45 Problems with BNF NotationBNF notation is too long. Must use recursion to specify repeated occurrences Must use separate an alternative for every option 46 Extended BNF Notation EBND adds notation for repetition and optional elements. {…} means the contents can occur 0 or more times: expr => expr + term | term becomes expr => term { + term } […] encloses an optional part: if-stmt => if (expr) stmt | if (expr) stmt else stmt becomes if-stmt => if (expr) stmt [else stmt] 47 Extended BNF Notation, continued(a | b | ...) is a list of choices. Choose exactly one. expr => expr + term | expr - term | term becomes expr => term { (+|-) term } Another example: term => factor { (*|/|%) factor } 48 EBNF compared to BNF BNF: expression expression + term EBNF:term term * factor | term / factor | factor factor (expression) | id | number EBNF: expression term { (+|-) term } term factor { (*|/) factor } factor '(' expression ')' | id | number 49 EBNF summary EBNF replaces recursion with repetition using {...}.(a|b|c) for choices [opt] for optional elements. In EBNF we need to quote (and) literals as '(' ... ')' expression term { (+|-) term } term factor { (*|/) factor } factor '(' expression ')' | id | number 50 EBNF: variations in notation (1)"opt" subscript instead of [ ]: function ::= type identifier(parameter_listopt) 51 EBNF: variations in notation (2)symbol ":" and one production per line (no "|") factor : (expression) number identifier 52 EBNF: variations in notation (3)"one of" for simple alternatives visibility : one of public private protected 53 EBNF: variations in notation (4)one or more: { }+ statementblock begin { statement ; }+ end expression term { addop term }+ addop one of + - 54 EBNF: class declarationHow would you declare a Java class using standard EBNF variation: "opt", "one of", { }+ 55 Notes on use of EBNF Do not start a rule with {…}:Right: expr => term { + term } Wrong: expr => { term + } term exception: left { x } is OK for simple token expr => { - } term ... For right recursive rules use [...] instead: expr => term + expr | term EBNF: expr => term [+ expr] Square brackets can be used anywhere: expr => expr + term | term | - term EBNF: expr => [-] term { + term } 56 Exercise 3: Rewrite this grammar using Extended BNF.sentence => noun-phrase verb-phrase . noun-phrase => article noun | noun article => a | the noun => boy | girl | cat | dog verb-phrase => verb noun-phrase | verb verb => sees | pets | bites 57 Exercise 4: Extend the grammar shown below to includeexponentiation: x^n Note that exponentiation is usually right associative and has higher precedence than * and / optional unary minus sign, e.g. -x, -4, -(a+b) expression => term { (+|-) term } term => factor { (*|/) factor } factor => '(' expression ')' | id | number 58 Syntax Diagrams An alternative to EBNF. Introduced for Pascal.Rarely used now: EBNF is much more compact. Example: if-statement