Grammar and Parse Trees

(Syntax)

(note to instructor, have a video to load under 1st CFG Grammar Example)

What makes a good programming Language?

* What do you think?
* BTW, don’t take any of these for granted!!

Creation Order of language X

* In a Systems Analysis and Design class you cover the ways applications are developed
  + the process for developing a language is much like a waterfall

|  |
| --- |
| **Creation Order of Language X** |
| Implementers shape how the code is formed, but are handcuffed by the Language |

Syntax vs. semantics

* 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)
* Both are closely related
* A well designed language you should be able to read the statement parentheses syntax and get what it is they will do (semantics)

|  |  |  |
| --- | --- | --- |
| **Syntax vs. Semantics** | | |
|  | Syntax | Semantics |
| Java | int x  = 12; | Set x to the value 12 |
| Python | x = 12 | Set x to the value 12 |

|  |
| --- |
| **Syntax vs. semantics example** |
| While (Boolean\_expression) statement   ] – physical makeup (syntax)      Meaning of the statement (semantics) |

Terms in syntax

* Language
  + set of sentences, combination of keywords
* Sentence
  + a string of characters over some alphabet
  + a line of syntax
* Lexeme
  + lowest level syntactic unit of a language (e.g., \*, sum, begin)
  + Numerical limits, operators, special words, etc…
  + a program is a study of lexemes
* Tokens
  + is a category of lexemes (identifiers)
  + words in a syntax

Lexemes and Tokens Revisited

* “LECK-seem”
* examples (each separated by a space)
  + const {  } cout << 23.23 ++ ; “Lupoli”

|  |
| --- |
| **Lexemes vs. Tokens** |
|  |

* Lexemes
  + are read in and recognized by a Scanner
    - **Scanner described below (images)**
  + that Scanner then places that lexeme into a Token category
* Tokens
  + lexemes broken down into the categories
    - reserved or keywords words
      * an identifier cannot be in the same sentence as a reserved word
      * int else;
    - identifiers
      * names of variables, methods, classes, etc…
    - Operators and special symbols
      * +, -, /, etc…
    - Literals or constants
      * Values placed in equations or hard coded digits

|  |
| --- |
| **Where the scanner is in the entire process** |
|  |

Unary Operators

* operators that act upon a single operand, set around a value
  + prefix or postfix around the value
  + 3 – (-2), x++

|  |  |
| --- | --- |
| **C family of Unary Operators** | |
| Increment: ++x, x++  Decrement: −−x, x−−  Address: &x  Indirection: \*x | Positive: +x  Negative: −x  One's complement: ~x  Logical negation: !x |

Grammars

* type of language generator, meant to describe the syntax of natural languages
* the grammar can have nested, recursive, self-similar branches in their syntax trees
  + so they can handle nested structures well.
* They can be implemented as state automaton with stack
  + This stack is used to represent the nesting level of the syntax
    - one portion may have to wait until another portion is solved
* Two grammar classes
  + Context-free (CF)
  + Regular
* Both used to describe the syntax of programming languages

|  |
| --- |
| **1st CFG Grammar Example** |
| EXP -> NUM  |  ( EXP OP EXP )  OP ->  +  |  - | \* | /  NUM ->  DIGIT  |  DIGIT NUM  DIGIT ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | -1 | -2 | …  Test if      ( 4 – (3 + 2) )        fits the grammar  [How did I get this answer?](https://www.youtube.com/watch?v=aF-Ml_YWF_U)  ***THINK OF THIS AS PATTERN MATCHING*** |

|  |
| --- |
| **4 digit value be covered by this grammar?** |
| // remember, <NUM> ***would not start*** the parse tree |

Do these fit the grammar?? And why, why not??

Hint: STOP as soon as you see something is off!! (The parser/compiler will!!)

Suggestion: double circle your terminal value

EXP -> NUM  |  ( EXP OP EXP )

OP ->  +  |  - | \* | /

NUM ->  DIGIT  |  DIGIT NUM

DIGIT ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | -1 | -2 | …

DRAW THE PARSE TREE!!!!

1. ( -1 \* ( 3 + **43**) )  Answer:
2. 9 ( 4 + 8 ) Answerb:
3. ( 8 + ( 6 \* (5 \* 2) ) ) Answerb:

Context-free language and grammar (CFGs)

|  |  |
| --- | --- |
| **Context Free Grammar (CFG) Example** | |
| a CFG for the language of all palindromes using letters a and b | S → P  P → ε    // think of epsilon as a “null”  P → a  P → b  P → aPa  P → bPb |

What are the differences between uppercase and lowercase letters?

* consists of a series of grammar rules
  + rules consist of a left hand side (LHS) that is a ***single*** phrased structure name
  + then “metasymbol” 🡪
  + followed by a right-hand side (RHS) consisting of a sequence of items that can be symbols or other phrase names.
* also called productions
* productions can be in several forms
  + Context Free Grammar
  + Backus – Naur form (BNF) which is next
* Why “Context Free”??
  + nonterminals appear ***singly*** on the LHS of the productions
  + means that each non-terminal can be replaced by any RHS choice, no matter where the non-terminal may appear

What other sentences would match the grammar above/below? (Create 3 more using the example below, with 7 or more characters)

|  |  |
| --- | --- |
| Sentence | parse tree proving it fits |
| aabaa |  |

S → P

P → ε    // think of epsilon as a “null”

P → a

P → b

P → aPa

P → bPb

Parse Trees

* graphical representation showing the hierarchical syntax structure of the sentence of the language they define
* or (I like better) a hierarchical representation of a derivation
* must know
  + the grammar
    - or may need many parse trees to solve what the grammar is
  + the sentence you are striving for

|  |  |
| --- | --- |
| **Parse Tree Example** | |
| Grammar | Parse Tree for A = B \* (A + C) |
| <assign> 🡪<id> = <expr>  <id> 🡪 A | B | C  <expr> 🡪 <id> + <expr>  | <id> \* <expr>  | **(** <expr> **)**  | <id> |  |

* each **internal** node is a non-terminal symbol (< ? >)
* every leaf is a terminal, ***operators included!!***
* every sub-tree describes one instance of an abstraction in a sentence

Terms in Grammar

* Grammar
  + a finite non-empty ***set of rules***
* abstractions
  + also called non-terminals
  + can have 2 or more distinct definitions/representations …
    - uses the | to separate sentences

|  |
| --- |
| **Defining separate definitions for the same rule** |
| <if\_stmt> 🡪 **if** ( <logic\_expr> ) <stmt>  <if\_stmt> 🡪 **if** ( <logic\_expr> ) <stmt> **else** <stmt>  using “|” will be  <if\_stmt> 🡪 **if** ( <logic\_expr> ) <stmt>  | **if** ( <logic\_expr> ) <stmt> **else** <stmt>  In these rules, <stmt> ….. (finish)  Think of “|” as OR!!! |

* lexemes and tokens ***in a grammar***
  + terminal and nonterminal symbols
  + more explained below

|  |
| --- |
| **Terms in a Grammar** |
|  |

* sentence
  + entire line in a rule
  + and final solution (line of syntax)
    - usually from a derivation (later)
  + ***Sebesta book uses both!!!***
* BNF description/grammar
  + generation device for defining languages
  + collection of rules.
* non-terminals
  + part of a sentence that can be further broken down
  + in normal form (non-BNF), UPPER CASE
* terminal
  + part of a sentence that cannot be broken down any further
  + in normal form (non-BNF), lower case
* start symbol
  + special non-terminal symbol, the very highest, non-reduced symbol in the grammar

Backus – Naur form (BNF)

* a form of CFG
* invented by John Backus for Algol 58
* widely used notation
* Used “abstraction” for general description notation for syntax structures

|  |
| --- |
| **Backus – Naur form (BNF)** |
|  |

* <assign> (or anything in < >) is defined as an instance of an abstraction
* RHS is made up of Tokens and Lexemes (like “,”)
* LHS is non-terminals
* Each line is called rule/production

The meaning of “list” in Grammar

* **not** the array or list data structure!!
* items of the same data type

|  |
| --- |
| **A “list” of items** |
| int a, b, c, d, e, f; |

* uses recursion in the Grammar in order to replicate the same “code” over and over
  + like in recursion, the rule must have a “base case” in order to stop
    - does not call itself (rule) again
  + please notice a “,” is used to separate the items of the list

|  |
| --- |
| **Example rule dealing with lists** |
| <ident\_list> 🡪 identifier ;                              // this line would be the base case  | identifier , <ident\_list> |

General Grammar setup

* some obvious (hopefully) things
  + <program> is the starting point of a syntax
    - or some rule that has an obvious starting name
  + special words or lexemes are in bold and mixed throughout the sentence

|  |
| --- |
| **First Example of a Grammar (again)** |
| <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt>  | <stmt> ; <stmt\_list>  <stmt> 🡪 <var> = <expression>  <expression> 🡪 <var> + <var>  | <var> - <var>  | <var>  <var> 🡪 **A** | **B** | **C** | **D**  Always start with upper left most non-terminal |

1. Which syntax below would NOT work with the Grammar above?
2. begin A = B + C end
3. begin A = A + B + C end
4. begin A = B end
5. Why would the following syntax ***not*** be correct with the grammar given?

begin A = C – B;

B = D + A; end

Hint: No, it’s not because it’s on two lines, and don’t assume anything!!

Answer:

1. Was there recursion in the grammar given?
2. What other operators are not supported in this Grammar? Why?

Answerb:

Derivations in General

* are what you JUST used to solve which syntax did NOT fit
  + in a more formulated, step by step way
* solution set from the Grammar given using its rules and ending with a sentence
* to solve
  + given ***TARGET SYNTAX, GRAMMAR and Derivation Order***
    - order covered in a minute
  + always start with the start symbol in the grammar and its rule
  + then use the other rules to get to the target syntax
* the results
  + the symbol “=>” means derives
  + notice in each line of the derivation, only one abstraction/substitution is derived
  + each line is derived from the line before (above)
  + each line is called sentential form

|  |  |
| --- | --- |
| **Example Grammar/Derivation Setup** | |
| <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt>  | <stmt> ; <stmt\_list>  <stmt> 🡪 <var> = <expression>  <var> 🡪 A | B | C  <expression> 🡪 <var> + <var>  | <var> - <var>  | <var> | target syntax:  **(begin A = B + C; B = C end)**  <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**  …  \*\* this uses a Left most derivation (explained below) |

Derivation Order

* order of derivation **replacement** of a sentential form
* Leftmost
  + in each line of the derivation, the **leftmost** non-terminal is replaced using a rule in the Grammar
* Rightmost (another order option)
  + rightmost non-terminal is solved first
  + creates different sentences (maybe)
* In order (another order option)

|  |
| --- |
| **Leftmost examples from Previous Derivation** |
| => **begin** <stmt> ; <stmt\_list> **end**  => **begin** <var> = <expression> **;** <stmt\_list> **end**  \*\* notice in line 2, you have two options, but only the left was replaced  => **begin** A = B + C **;** <var> = <expression> **end**  => **begin** A = B + C **;** B = <expression> **end** |

* Derivation order (should not at least) has no effect on the language generated by the grammar
* by using ALL of the different order combinations do you get your entire language sentences
  + if we did not have a target syntax
  + which in reality, would be impossible (super huge!) to get ALL combinations, but at least a good feel for it

Solving Derivations and First Example

* to solve
* given ***TARGET SYNTAX, GRAMMAR and Derivation Order***
* always start with the start symbol in the grammar and its rule
* then use the other rules to get to the target syntax
* but you CANNOT change derivation orders (left, right, etc…) within the same derivation
  + either the whole derivation is left, right, or inorder

|  |  |
| --- | --- |
| **1st Leftmost Derivation Example** | |
| **Given Grammar** | **Derivation with Target Syntax**  **(begin A = B + C; B = C end)** |
| <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt>  | <stmt> ; <stmt\_list>  <stmt> 🡪 <var> = <expression>  <var> 🡪 A | B | C  <expression> 🡪 <var> + <var>  | <var> - <var>  | <var> | <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** |

|  |  |
| --- | --- |
| **Exercise #1** | |
| **Given Grammar** | **Leftmost Derivation w/ Target Syntax** |
| **<sentence> -> <subject> <predicate>**  **<subject> -> <article> <noun>**  **<predicate> -> <verb> <direct-object>**  **<direct-object> -> <article> <noun>**  **<article> -> THE | A**  **<noun> -> MAN | DOG**  **<verb> -> BITES | PETS** | **A DOG PETS A DOG**  <sentence> ->  Answer: |
| **Exercise #2** | |
| **Given Grammar** | *Rightmost* **Derivation w/ Target Syntax** |
| **<sentence> -> <subject> <predicate>**  **<subject> -> <article> <noun>**  **<predicate> -> <verb> <direct-object>**  **<direct-object> -> <article> <noun>**  **<article> -> THE | A**  **<noun> -> MAN | DOG**  **<verb> -> BITES | PETS** | **THE MAN BITES A DOG**  **(rightmost!!)**  Answer: |

Using the JFlap tool

* [www.jflap.org](http://www.jflap.org)
  + free to download
  + used to draw a parse tree
  + used to test a grammar in many ways
* installation
  + fill out form
  + find “JFLAP\_Thin.jar” and download to desktop. No installation needed.
* [how to create parse trees in JFlap](https://www.youtube.com/watch?v=PAynFSleNmU)

Ambiguity

* where a sentence can be represented by more than one parse tree
  + NOT DERIVATION!!!
  + this is bad!!!
  + ***could be that***
    - ***left/right most derivations do not match!!***
    - ***several lefts don’t match!!***
* why does this matter??
  + mathematically seems the same
  + just try programming this!!
* In ***proving ambiguity*** you are TRYING TO MISMATCH with the SAME TARGET Syntax
  + just make sure your syntax is correct first!!

|  |  |
| --- | --- |
| **1st Ambiguous Grammar Example** | |
| Grammar | Solution(s) for: A = B + C \* A |
| <assign> 🡪<id> = <expr>  <id> 🡪 A | B | C  <expr> 🡪 <expr> + <expr>  | <expr> \* <expr>  | **(** <expr> **)**  | <id> |  |

|  |  |  |
| --- | --- | --- |
| **2nd Ambiguous Grammar Example** | | |
| Grammar | Solution(s) for: 3 + 4 \* 5 | |
| *E* → E + E *E* → E \* E *E* → **i** | (left most                           -                 right  most) | |
|  | Derviation | |
|  | => E=> E *+* E=> *3* ***+***E=> *3* ***+***E *\** E=> *3* ***+*** *4 \** E=> *3* ***+*** *4 \* 5* | => E=> E *\** E=> E***+***E *\** E=> *3* ***+***E *\** E=> *3* ***+*** *4 \** E=> *3* ***+*** *4 \* 5* |

|  |  |
| --- | --- |
| **3rd Ambiguous Grammar Example** | |
| Grammar | Solution(s) for: const – const / const |
| <expr> 🡪 <expr> <op> <expr>   |  **const**  <op> 🡪 **/**  |  **-** | (left most                   -         right most) |

* if the Grammar is ambiguous there are some notable things to avoid
  + not defining operator precedence
    - further explained below, but you already know this
  + have the SAME abstraction in a rule more than one
* ***Anything ambiguous can be re-written***

Fixing Ambiguity – well kinda

* sadly, there is no “procedure” to fix
* cannot be done automatically
* more of a trial and error
* but ***USUALLY*** it includes
  + more rules
  + better precedence order (see below)
  + not as many “|” in a single sentence

Proving Ambiguity within a Grammar

* start with a Grammar and a legit sentence
  + ***USE ALL RULES WITHIN THE GIVEN GRAMMAR!!!***
* try both a left and right derivation
  + if the trees EXACTLY match, grammar ***could*** be good

|  |  |
| --- | --- |
| **Proving Ambiguity with input string:  x + y + z** | |
| Left (parse tree) | Right (parse tree) |
| Grammar to determine if ambiguous  <S> 🡪 <A>  <A> 🡪 <A> + <A> | <id>  <id> 🡪 a | b | c | |
| Leftmost (GREEDY) Parse Tree | Rightmost (GREEDY) Parse Tree |
|  |  |

What does greedy mean?

* watch how to prove it using parse trees
  + <https://www.youtube.com/watch?v=3F8kBL07TEc>

Greedy means the Grammar is crap

* Well, really means crap or educational example
* The better the grammar the less you worry about greedy

|  |  |
| --- | --- |
| **Greedy versus Spare responses** | |
| Grammar to determine if ambiguous  <S> 🡪 <A>  <A> 🡪 <A> + <A> | <id>  <id> 🡪 a | b | c | |
| Greedy Leftmost | Spare Leftmost |
|  |  |

<S> 🡪 <A>

<A> 🡪 <A> + <A> | <id>

<id> 🡪 a | b | c

1. Create ***another*** target syntax string that works with the given grammar above
   1. Instead of x + y + z, try something new with a,b,c respectfully
2. Try left most and right most ***greedy*** ***parse tree*** to prove ambiguity on that new target string
3. Do the same thing (greedy) with the grammar below: **Answerb:**

**<binary-string> -> 0**

**| 1**

**| <binary-string> <binary-string>**

|  |  |
| --- | --- |
| Left (***parse-tree***) | Right (***parse-tree***) |
|  |  |

<S> 🡪 <A> a <B> b

<A> 🡪 <A> b | b

<B> 🡪 a <B> | a   **Answerb:**

1. Which of the following sentences are in the languages generated above?
2. baab
3. bbbab
4. bbaaaaa Answer:
5. bbaab
6. Describe in English, the language defined by the following grammar:

<S> 🡪 <A> <B> <C>

<A> 🡪 a <A> | a

<B> 🡪 b <B> | b

<C> 🡪 c <C> | c Answer:

Remember how to solve trees?

* remember it’s bottom up
* leaves-ish are solved first
* tree = recursive, remember the last valid recursive call REALLY gets solved first
* we use this idea for precedence below

|  |  |
| --- | --- |
| **Inorder Expression Trees (Solving Trees)** | |
| (((5 + 2) \* 5) + 3) |  |

1. In the left tree, what portion of the equation is completed first? Why?
2. What would the answer be for either of these?
3. What would the equations below look like as a tree??? Use our normal understanding of (PEMDAS) to create the tree.
   1. (((3 + 4) \* 5) \*6)
   2. 6 + 128 \* 34
   3. 2 ^ 6 \* 12 + 5  **Answersb:**

Setting up Operator Precedence in a Grammar

* setting the order of operations in a grammar
* can assign different levels or precedence for operators in the grammar design
* operators **lower**(est) on the parse tree must be completed/solved first
  + left parse tree from example below is what we want

|  |
| --- |
| **Solving by precedence** |
| How would we solve 3 + (5 \* 9 + 2) using a tree?  <assign> 🡪<id> **=** <expr>  <expr> 🡪 <expr> **+** <term>  | <term> // but if not +, add layer  <term> 🡪 <term> **\*** <factor>  | <factor> // but if not \*, add another layer  <factor> 🡪 **(** <expr> **)**  | <id>  <id> 🡪 A | B | C |

* the correct setup requires
  + separate non-terminal symbols to represent each operator
  + the rule for \* and / must be more derivations from the + and – operators
    - this will push them further DOWN the parse tree making them FIRST to be completed

<expr> 🡪 <expr> + <term>

|  term

|  |
| --- |
| **Unambiguous grammar for Operator (+ and \*) Precedence** |
| <assign> 🡪<id> **=** <expr>  <expr> 🡪 <expr> **+** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <factor>  <factor> 🡪 **(** <expr> **)**  | <id>  <id> 🡪 A | B | C  What other operators should be deeper or even with <factor>?  \*\*\* notice to JUST get to factor (which has the highest precedence) it takes many rules to get to!! <expr> 🡪 <term> 🡪 <factor> |

1. Draw the simple parse trees for: (ignore left or rightmost for now)
   1. A = C + B which level (starting at 0) does the + reside?
   2. A = C \* B which level does the \* reside?

answerb:

1. Using the grammar above, which ALWAYS will come first, (top down) <expr> or <term>?

(We’ll do a more complex one next!)

|  |  |
| --- | --- |
| **Derivation for A = B + C \* A** | |
| <assign> 🡪<id> **=** <expr>  <expr> 🡪 <expr> **+** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <factor>  <factor> 🡪 **(** <expr> **)**  | <id>  <id> 🡪 A | B | C |  |
| (leftmost)    <assign> => <id> = <expr>  => A = <expr>  => A = <expr> + <term>  => A = <term> + <term>  => A = <factor> + <term>  => A = <id> + <term>  => A = B + <term>  => A = B + <term> \* <factor>  => A = B + <factor> \* <factor>  => A = B + <id> \* <factor>  => A = B + C \* <factor>  => A = B + C \* <id>  => A = B + C \* A | (rightmost)  <assign> => <id> = <expr>  => <id> = <expr> + <term>  => <id> = <expr> + <term> \* <factor>  => <id> = <expr> + <term> \* <id>  => <id> = <expr> + <term> \* A  => <id> = <expr> + <factor> \* A  => <id> = <expr> + <id> \* A  => <id> = <expr> + C \* A  => <id> = <term> + C \* A  => <id> = <factor> + C \* A  => <id> = <id> + C \* A  => <id> = B + C \* A  => A = B + C \* A |
| **Parse Tree for A = B + C \* A** | |
| Draw left parse tree here!! (**answernp**) | Draw right parse tree here!! (**answernp**) |

|  |
| --- |
| **Parse Tree for A = B + C \* A** |
| **(left and right!!)**    Remember, the as long as the tree is the same (left or right) the grammar is unambiguous |

1. Try creating parse trees for these:

|  |  |
| --- | --- |
| <assign> 🡪<id> **=** <expr>   // **this grammar is unambiguous**  <id> 🡪 A | B | C  <expr> 🡪 <expr> **+** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <factor>  <factor> 🡪 **(** <expr> **)**  | <id> | |
| A = (B + C) \* A  **Answerb:** | A = (B \* C) \* A  **Answerb:** |

1. When you drew the tree, did:
   1. the tree solve the actual problem from bottom up correctly?
   2. the grammar support the precedence correctly?

Associativity in General

* not only if you have to deal with \* and /
* but operators of the same precedence
* parse trees (leftmost and rightmost) SHOULD look exactly the same, BUT solve to the same equation because of **associativity**
* **BUT THE DERIVATIONS (leftmost and rightmost) will NOT LOOK the same!!**
* **remember, usually LEFT to RIGHT when using operators of the same precedence.**
  + +, -, \*, /, % are all evaluated ***left to right***
  + B + C + A
    - is really (B + C) + A

|  |
| --- |
| **Associativity Grammar for Example** |
| <assign> 🡪<id> **=** <expr>  <id> 🡪 A | B | C  <expr> 🡪 <expr> **+** <term>     // left associativity  | <term>  <term> 🡪 <term> **\*** <factor>     // left associativity  | <factor>  <factor> 🡪 **(** <exp> **)**  | <id> |

Associative Rule Breakers!!  Intro. to Recursion

* ***Right to Left*** rule breakers!!!
  + unary
  + power (7 \*\* 8)  // ( \*\* or ^ depending on the language)
  + !
  + ~ (bitwise compliment)

|  |
| --- |
| **Why Power breaks the rules** |
| 1.0000001 \* 107 = (Correctly) 10000001  (1.0000001 \* 10)7 = (LEFTSIDE done first) = 101.0000101 1.0000001 \* (107) = (RIGHTSIDE done first) = 10000001 |

* Recursion
  + rule calls itself
* Left Recursion, call (symbol) ***to itself*** is physically LEFT of the operator
* Right Recursion, call (symbol) ***to itself*** is physically RIGHT of the operator
* ***Remember, the part of the tree that “dangles” the lowest is completed first!!***

|  |  |  |  |
| --- | --- | --- | --- |
| **Example of Left and Right Recursion Parse Trees** | | | |
| Left Recursive | | Right Recursive | |
| Solves left hand side first!! | | Solves right hand side first!! | |
| **Recursion Example** | | |
| Left Recursion | Right Recursion | |
| (from previous examples)  <expr> 🡪 <expr> **+** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <factor> | (specifically for power)  <factor> 🡪 <exp> \*\* <factor>  | <exp>  <exp> 🡪 ( <expr> )  | <id> | |

|  |  |  |
| --- | --- | --- |
| **Showing the Proper Parse Tree for \*\*** | | |
| Grammar | A + B \* C | A \*\* B + C |
| <expr> 🡪 <expr> **+** <term>  | <expr> **-** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <term> **/** <factor>  | <factor>  <factor> 🡪 <exp> \*\* <factor>  | <exp>  <exp> 🡪 ( <expr> )                      | <id>  <id> 🡪 A | B | C | … |  | |

1. Try A + (B \* C) / D just to get used to this new grammar. Did it come out right?
2. Then try A \*\* B \*\* C. Which portion was the lowest on the tree?

Answersb:

FYI Section

Who needs to know and understand your language?

* there are many people needed in the development of a new programming language, but who are they??

Answers:

Language recognizers and Generators

* Language Recognizers (also called Finite State Machines!!)
  + accept a Language like a Machine.
    - The Machines take a string as input.
    - The Machines will accept the input if when run,
    - the Machine stops at an accept state. Otherwise the input is rejected.
    - If a Machine M recognizes all strings in Language L, and accepts input provided by a given string S, M is said to accept S.
      * Otherwise M is said to reject S. S is in L if and only if M accepts S.
    - JFLAP is a FSM application!!
  + Context Free Grammar (CFGs) are a well-known type of language generator!!
* Language Generators
  + create the strings (sentences) of a Language.
  + A generator provides a construction description.
  + If a generator is able to construct all stings in a Language L, and every string S that can be constructed by that generator is in L, we can say that the generator is a generator for the language L.
  + If there is no way to construct a string S from the generator, S is not in L.
  + [Push Down Automata](http://everything2.com/title/Push+Down+Automata)s (PDAs) are a well known form of language recognizers.
    - not really covered here

Extended (Updated) BNF (EBNF)

* like anything else, some updates were made for convenience
* only increased the readability/writability (for us!!)
* there are other versions of the updates
* 3 most common updates
  + optional part in RHS
    - Optional parts are placed in brackets [ ]
    - ***almost*** anything with |’s in reality is now replaced
      * if symbol is unique to any of the rules, it may need to stay
    - less rules, or as many lines to write

|  |
| --- |
| **EBNF “Optional” Update (Extension)** |
| in C++:  <if\_stmt> 🡪 **if**( <expression> ) <statements> [**else** <statements> ]    <if\_stmt> 🡪 **if**( <expression> ) <statements>  | **if**( <expression> ) <statements> **else** <statements> |

* repeating
  + 0 or more!!
  + use of braces in an RHS to indicate that the enclosed part can be repeated indefinitely ***OR*** left out altogether
  + works great for lists!!
  + look for any recursion in the BNF form to be replaced

|  |
| --- |
| **EBNF “Repeating” Update** |
| <ident\_list> 🡪 <identifiers> { , <identifiers> }    <ident\_list> 🡪 identifier  | identifier , <ident\_list> |

* Multiple choice!!
  + choose a single element from a group
  + options are placed in ( )s and separated by |
  + notice the | count is the same, just now in one line

|  |
| --- |
| **EBNF “Multiple Choice” Update** |
| <term> 🡪 <term> ( \* | / | % ) <factor>    <term> 🡪 <term> \* <factor>  | <term> / <factor>  | <term> % <factor> |

This new fangled EBNF thingy

* the brackets, braces, and parentheses in the EBNF for are called “metasymbols”
  + notational tools
  + not terminal symbols
* issues
  + in case the metasymbols are also terminal symbols in the language
    - the instance that are terminals are underlined or quoted
  + loss of associativity
    - using the EBNF for of + above
    - no longer does it imply direction of associativity
      * this is fixed by using a EBNF syntax analyzer discussed later

|  |  |
| --- | --- |
| **Comparison #1 between BNF and EBNF** | |
| BNF | EBNF |
| <expr> 🡪 <expr> **+** <term>  | <expr> **-** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <term> **/** <factor>  | <factor>  <factor> 🡪 <exp> \*\* <factor>  | <exp>  <exp> 🡪 ( <expr> )  | <id> | <expr> 🡪 <term> { (**+ | -)**  <term> }  <term> 🡪 <factor> { (**\* | /)**  <factor> }  <factor> 🡪 <exp> { \*\* <factor> }  <exp> 🡪 ( <expr> )  | <id> |

|  |  |
| --- | --- |
| **Comparison #2 between BNF and EBNF** | |
| BNF | EBNF |
| <expr> → <expr> + <term>  | <expr> - <term>  | <term>  <term> → <term> \* <factor>  | <term> / <factor>  | <factor> | <expr> → <term> {(+ | -) <term>}  <term> → <factor> {(\* | /) <factor>} |

Converting BNF to EBNF Hints

* Look for recursion in grammar:

A ::= a A | B

⇒ A ::= a { a } B

* Look for common string that can be factored out with grouping and options.

A ::= a B | a

⇒ A := a [B]

|  |  |
| --- | --- |
| **BNF to EBNF Conversion Help** | |
| (BNF)  <expr> ::= <digits>  <digits> ::= <digit>  | <digit> <digits> | (EBNF)  <expr> ::= <digit>+ |
| <expr> ::= <digits> | empty | <expr> ::= <digit>\* |
| <id> ::= <letter>  | <id><letter>  | <id><digit> | <id> ::= <letter> (<letter> | <digit>)\* |

|  |
| --- |
| **Lupoli’s Strategy on BNFs to EBNF** |
| <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt>  | <stmt> ; <stmt\_list>  <stmt> 🡪 <var> = <expression>  <var> 🡪 A | B | C  <expression> 🡪 <var> + <var>  | <var> - <var>  | <var>     1. Find all Multiple choice, notice no ALL | are “multiple choice”    1. those that have a < ? > **option** < ? > are prime canditates    2. anything with multiple terminal “identifiers” are fair game    3. ( )s   <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt>  | <stmt> ; <stmt\_list>  <stmt> 🡪 <var> = <expression>  <var> 🡪 (A | B | C)  <expression> 🡪 <var> (+ | -) <var>  | <var>     1. Find “repeating” sequences (anything that can have multiple statements)    1. repeating can be 0!! (or more)    2. { }s   <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt> {; <stmt\_list> }  <stmt> 🡪 <var> = <expression>  <var> 🡪 (A | B | C)  <expression> 🡪 <var> { (+ | -) <var> }     1. Optionals, rules that are very similar, one extends from another    1. none in this case   <program> 🡪 **begin** <stmt\_list> **end**  <stmt\_list> 🡪 <stmt> {; <stmt\_list> }  <stmt> 🡪 <var> = <expression>  <var> 🡪 (A | B | C)  <expression> 🡪 <var> { (+ | -) <var> } |

1. Convert the BNF grammar to EBNF

|  |  |
| --- | --- |
| BNF | EBNF |
| <assign> 🡪<id> = <expr>  <id> 🡪 A | B | C  <expr> 🡪 <id> + <expr>  | <id> \* <expr>  | **(** <expr> **)**  | <id> |  |

1. Convert the following from EBNF to BNF

S 🡪 A {bA}

A 🡪 a [b] A

Answers

|  |
| --- |
| **Simple Grammar Exercises** |
| #2 |
| #3 |

|  |  |
| --- | --- |
| **Binary String Grammar Example** | |
| **<binary-string> -> 0**  **| 1**  **| <binary-string> <binary-string>**  **1010111** | |
| Leftmost Greedy | Rightmost Greedy |
|  |  |

|  |
| --- |
| **Turning Equations into Trees** |
|  |
|  |

|  |
| --- |
| **Why certain operations are not covered in grammar** |
| Where order does not matter (covered) |
| 3 + 4 =  4 + 3 =  3 \* 4 =  4 \* 3 = |
| Where order does matter (would need to be added) |
| 4 / 2 =  2 / 4 = |

|  |  |
| --- | --- |
| **Proving Order precedence works** | |
| A = C + B | A = C \* B |
|  | |

|  |  |
| --- | --- |
| A = (B + C) \* A | A = (B \* C) \* A |
|  |  |

|  |  |
| --- | --- |
| A + ( B \* C ) / D | A \*\* B \*\* C |
|  |  |

Resources:

<http://stackoverflow.com/questions/2842809/lexers-vs-parsers>

<http://teaching.idallen.com/cst8152/97w/slides/sld021.htm>

<http://www.slideshare.net/dasprid/about-tokens-and-lexemes>

<http://everything2.com/title/Language+Generators+vs.+Language+Recognizers>

<http://www.antlr.org/wiki/display/CS652/Grammars>

<http://goose.ycp.edu/~dhovemey/fall2009/cs340/lecture/lecture2.html>

<http://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf>

<http://en.unitedstatesof.net/2008/09/11/2-dlr-scanner/>

<http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples.html>

<http://www.box.com/shared/e31pciv7b9>

<http://www.codeproject.com/KB/cs/intro_functional_csharp2/figure3.png>