COSC252: Programming Languages:

Language Basics and Overview

Jeremy Bolton, PhD
Asst Teaching Professor

GEORGETOWN UNIVERSITY

Notes

Notes: Read ALSU 1 – 2

Topics covered in CH1 will be revisited in more detail throughout

Sign up for Gradiance and begin Assignment #2

Outline

I. Language Basics
   I. Definitions
II. Grammars and Syntactic Analysis
   I. BNF
   II. Derivations
   III. Parse trees
III. Tokens and Lexical Analysis

The Basics of Programming Languages

- Before we can learn about each of these steps, we will formalize the concepts and nomenclature

Basic Elements of a Language

- Language: A set of sentences.
  - Exhaustive list (roster notation) is not practical. Usually a syntax is used.
- Sentence: a string of lexemes in a language (as specified by the syntax of the language)
- Syntax: A specification that can be used to determine whether a sentence is in a set (or not).
- Lexeme: Lowest level, integral, syntactic unit that has meaning
  - e.g. word, number, operator, …
- Token: A lexeme group or lexeme category

The Basics of Programming Languages

- Lexical Analysis: identifying and categorizing lexemes
- Syntactic Analysis: determining if sentence (string of lexemes) is in language.

Example: Identify the lexemes and tokens

- Example: English Language
- Sample Sentence:
  - John has a cat, and Mary has a dog.

Lexeme	Token (lexeme category)
John	noun
has	Verb
a	article
cat	noun
,	comma
and	conjunction
…	

Example: Identify the lexemes and tokens

- Example: Programming Language
- Sample Sentence:
  - sum = sum + 1.0;

Lexeme	Token (lexeme category)
sum	variable
=	assignment operator
sum	variable
+	plus operator
1.0	double
;	semicolon

Discussion: Syntactic Analysis

- Identifying and categorizing lexemes is often an easy task, but how do we determine whether a sentence is in a language?

- Example: Language L is the set of all English sentences.
  - L = {s | s is an English sentences}
  - When given a sentence s, how doe we determine whether it is in the English Language?
    - Lets say s = “The cat has a hat.”
    - Option: Compare s to all sentences in L
      - Brute Force, Exhaustive search
      - No – not practical

Rules for a Language

- Syntax Rules are used to characterize a language
- Lets try to develop some simple rules for English Sentences:

Syntax

- How can we use syntax to define or characterize a language?
  - There is no “perfect” solution or recipe

- There are 2 categories for syntax implementation and therefore 2 categories for language characterization.
  - Recognizers

  - Generators

Generators vs Recognizers

- Both generators and recognizers are useful in different applications, but each is limited
- Recognizers
  - EG compiler, Clearly useful
  - Compilers are generally very complex: Not good for *communicating* a language or describing a language
- Generators
  - Useful for learning or describing a language
  - (Examples upcoming)

Recognizers

- Recognizers take as input a sentence and determine whether the sentence is in the language.
  - Example: compiler
  - A compiler is an algorithmic representation of syntax
  - A compiler is a syntactic analyzer
    - The parsing portion of the compiler

Using Generators to Characterize a Language

- Noam Chomsky
  - MIT Linguist
  - Published work on categories of Languages / Grammars, 1950s
    - Two categories are commonly used in Programming Languages
      - Context-Free Languages
        - » Often used to characterize programming language sentence structure
      - Regular Languages
        - » Often used to characterize the structure of lexemes / tokens
- John Backus and Peter Naur developed a formal notation for generating a Context Free Language, (similar to the notation used by Chomsky)
  - BNF (Bachus Naur Form)

Grammars

- Generators are often implemented as a set of rules, called grammars.

- BNF grammars are grammars consisting of
  - terminals: lexemes (integral syntactic units)
  - non-terminals: abstract compositions of terminals
    - Have at least two possible forms
  - productions: composition rules for non-terminals.

BNF Example

- BNF production for English grammar example

$$< sentence > \rightarrow < noun > < verb > < adjective > < period >$$

Each line in BNF is a rule or *production*. Each production consists of a left hand side (LHS), followed by an arrow ("is produced by"), followed by a right hand side (RHS).

- The production defines the LHS abstraction, or *non-terminal*
  - Here sentence is defined to be produced by a noun followed by a verb followed by an adjective followed by a period.
- Note this production requires that we define noun, verb, … as well.

BNF Example

- BNF snippet for English grammar example

$$< sentence > \rightarrow < noun > < verb > < adjective > < period >$$
$$< noun > \rightarrow Bob$$
$$< verb > \rightarrow is$$
$$< adjective > \rightarrow tall$$
$$< period > \rightarrow \ .$$

Each line in BNF is a rule or *production*. Each production consists of a left hand side (LHS), followed by an arrow (“derives”), followed by a right hand side (RHS).

Each non-terminal is enclosed by a “< >”. Each non-terminal should have a defining production.

All other character strings, *lexemes*, are called *terminals* in this context.

BNF Example

- BNF for English grammar example

$< sentence > \rightarrow < noun > < verb > < adjective > < period >$

$< noun > \rightarrow Bob$

$< verb > \rightarrow is$

$< adjective > \rightarrow tall$

$< period > \rightarrow$.

What sentence(s) can be generated using this BNF grammar?
What sentence(s) are in this Language?
How many sentence(s) are in this Language?

BNF Example

- BNF snippet for Programming grammar example

$$< assignment > \rightarrow < variable > = < expression >$$
$$< expression > \rightarrow < variable > \ + < variable >$$
$$< expression > \rightarrow < variable > \ - < variable >$$

Use the "|" symbol to represent the logical OR. Multiple definitions of each non-terminal can be written in the same production.

$$< expression > \rightarrow < variable > \ + < variable > \ | \ < variable > - < variable >$$

GEORGETOWN UNIVERSITY

Listing out non-terminals

- Non-terminals can generally take on many forms
- How can we practically list out all possible forms of a non-terminal?
- Example <expr>

$$< expr > \rightarrow < var > \ + < var > \ |$$
$$< var > + < var > + < var >$$
$$< var > + < var > + < var > + < var >$$
$$\ldots$$

Listing Out Non-Terminals

$$< expr > \rightarrow < var > \ + < var > \ |$$
$$< var > + < var > + < var >$$
$$< var > + < var > + < var > + < var >$$
$$\ldots$$

The above is tedious and impractical. We can avoid such definitions by recursively defining non-terminals.

$$< expr > \rightarrow < expr > \ + < var > \ | < var >$$

Generating Sentences using BNF

- Start Symbol: One unique abstraction used to start a derivation
- Derivation: a repeated application of BNF rules

A simple but complete grammar

$$< program > \rightarrow begin < stmt > end$$
$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < var > + < var > \mid < var > - < var >$$
$$< var > \rightarrow x \mid y \mid z$$

Generation / Derivation Example

- Derivation: a repeated application of BNF rules
  - Starting with start symbol, repeatedly replace a non-terminal using on of the production rules until no non-terminals remain
  - Only reduce one non-terminal per line of derivation – it is best not to perform multiple steps per line

- Try to generate: begin x = y + z end

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < var > + < var > \mid < var > - < var >$

$< var > \rightarrow x \mid y \mid z$

- $< program >$

GEORGETOWN UNIVERSITY

Derivation Practice

- Try to derive sentence: begin y = x – y end

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < var > + < var > \mid < var > - < var >$

$< var > \rightarrow x \mid y \mid z$

- $< program >$
- $begin < stmt > end$
- $begin < var > = < expr > end$
- $begin \quad y = < expr > end$
- $begin \quad y = < var > - < var > end$
- $begin \quad y = x \quad - < var > end$
- $begin \quad y = x \quad - \quad y \quad end$

GEORGETOWN UNIVERSITY

Derivation Practice

- Try to derive sentence: begin y = x – y + z end

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < var > + < var > \mid < var > - < var >$

$< var > \rightarrow x \mid y \mid z$

- $< program >$
- $begin < stmt > end$
- $begin < var > = < expr > end$
- $begin \quad y = < expr > end$
- $begin \quad y = < var > - < var > end$
- ...
- Cannot! Need a more general production rule
- What rule can we add?

Derivation example with recursive production

- Try to derive sentence: begin y = x – y + z end

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > - < var >$

$< var > \rightarrow x \mid y \mid z$

- $< program >$
- $begin < stmt > end$
- $begin < var > = < expr > end$
- $begin \quad y = < expr > end$
- $begin \quad y = < expr > + < var > end$
- $begin \quad y = < expr > - < var > + < var > end$
- $begin \quad y = < var > - < var > + < var > end$
- $begin \quad y = x - < var > + < var > end$
- $begin \quad y = x \ - y + < var > end$
- $begin \quad y = x \ - y + \ z \ end$

GEORGETOWN UNIVERSITY

LMD vs RMD

- Note: Multiple derivations *may* exist per sentence.
  - Many sentences have multiple derivations
- Two common derivation strategies
  - Left Most Derivation: During derivation, always reduce left-most non-terminal at each step.
  - Right Most Derivation: During derivation, always reduce right-most non-terminal at each step.
- Note: Derivation order has no effect on the language generated

LMD vs RMD: $begin\ y = x - y\ end$

LMD

- $< program >$
- $begin < stmt > end$
- $begin < var > = < expr > end$
- $begin \quad y = < expr > end$
- $begin \quad y = < var > - < var > end$
- $begin \quad y = x \ - < var > end$
- $begin \quad y = x \ - \ y \ end$

RMD

- $< program >$
- $begin < stmt > end$
- $begin < var > = < expr > end$
- $begin < var > = < var > - < var > end$
- $begin < var > = < var > - y \ end$
- $begin < var > = \ x \ - \ y \ end$
- $begin \quad y = x \ - \ y \ end$

Note: Any intermediate representation derived from the start symbol called a *sentinel form. If derived via a {LMD, RMD} it is a {left, right} sentinel form.*

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < var > + < var > \mid < var > - < var >$

$< var > \rightarrow x \mid y \mid z$

GEORGETOWN UNIVERSITY

In-class Exercise

$< program > \rightarrow begin < stmt > end$

$< stmt > \rightarrow < var > = < expr >$

$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > - < var >$

$< var > \rightarrow x \mid y \mid z$

- Derive (LMD or RMD): begin x = y + z – x end

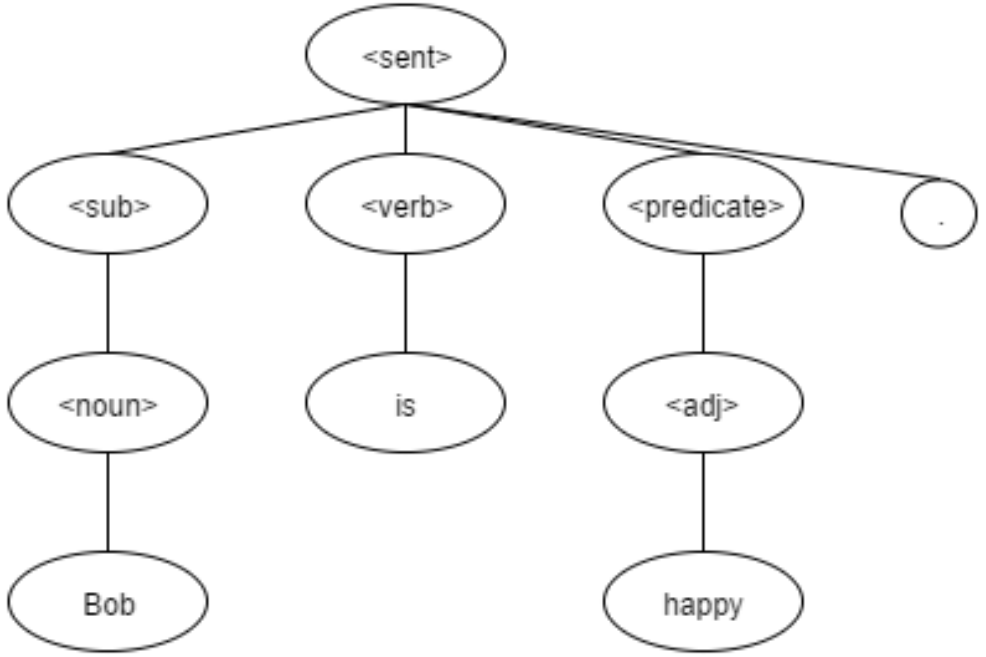
In Class Exercise

- Construct a grammar that allows programs with operators +, -, /, and *.

Parse Trees

- Hierarchical representation of the syntactic structure of a sentence
  - Based on syntax rules / derivation
  - *LHS of production is parent node and RHS are children nodes*
  - Example: Bob is happy.

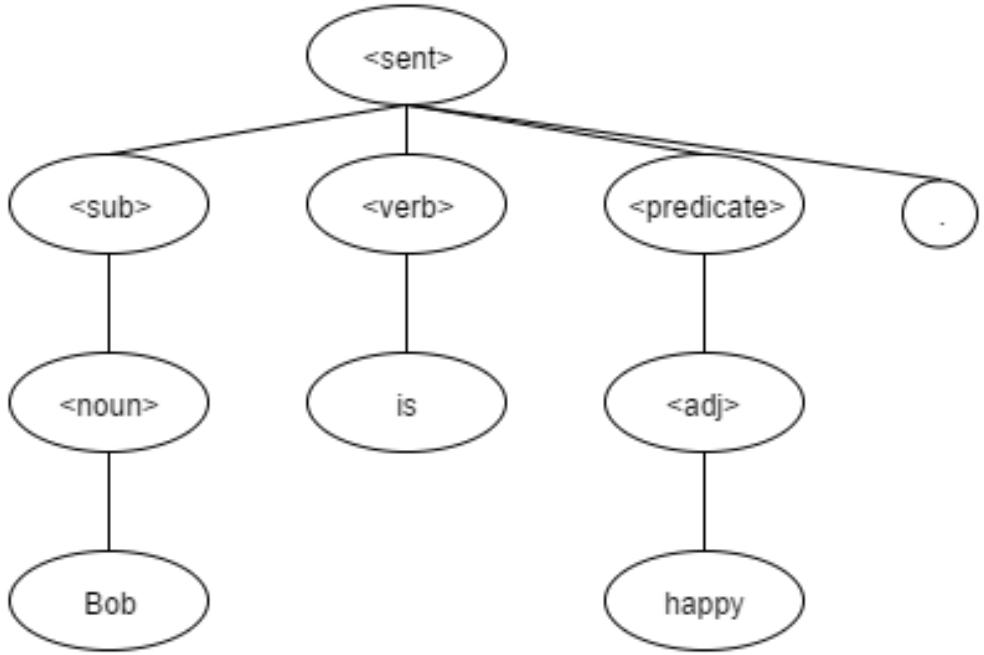
$< sent > \rightarrow < sub > < verb > < predicate > .$
$< sub > \rightarrow < noun >$
$< predicate > \rightarrow < art > < noun > | < adj >$
$< noun > \rightarrow Bob$ | Mary
$< art > \rightarrow a$ | the
$< verb > \rightarrow is$
$< adj > \rightarrow tall$ | happy | sad



GEORGETOWN UNIVERSITY

Parse Trees

- Observations
  - Root: start symbol
  - Internal nodes are non-terminals
  - Leaf nodes are terminals



$< sent > \rightarrow < sub > < verb > < predicate > .$
$< sub > \rightarrow < noun >$
$< predicate > \rightarrow < art > < noun > \mid < adj >$
$< noun > \rightarrow Bob$ | Mary
$< art > \rightarrow a$ | the
$< verb > \rightarrow is$
$< adj > \rightarrow tall$ | happy | sad

GEORGETOWN UNIVERSITY

Parse Tree (programming) Example

- Example
  - Based on syntax rules / derivation
  - Example: begin x = x + y end

$$< program > \rightarrow begin < stmt > end$$

$$< stmt > \rightarrow < var > = < expr >$$

$$< expr > \rightarrow < var > \mid < expr > + < var >$$

$$\mid < expr > - < var >$$

$$< var > \rightarrow x \mid y \mid z$$

In Class -- Parse Tree (programming) Example

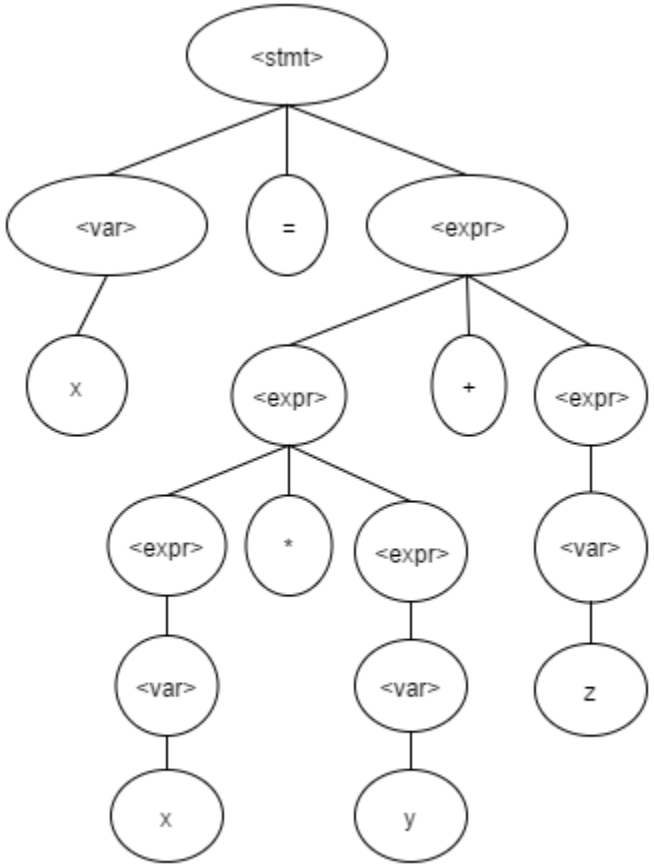
- Example: create parse tree

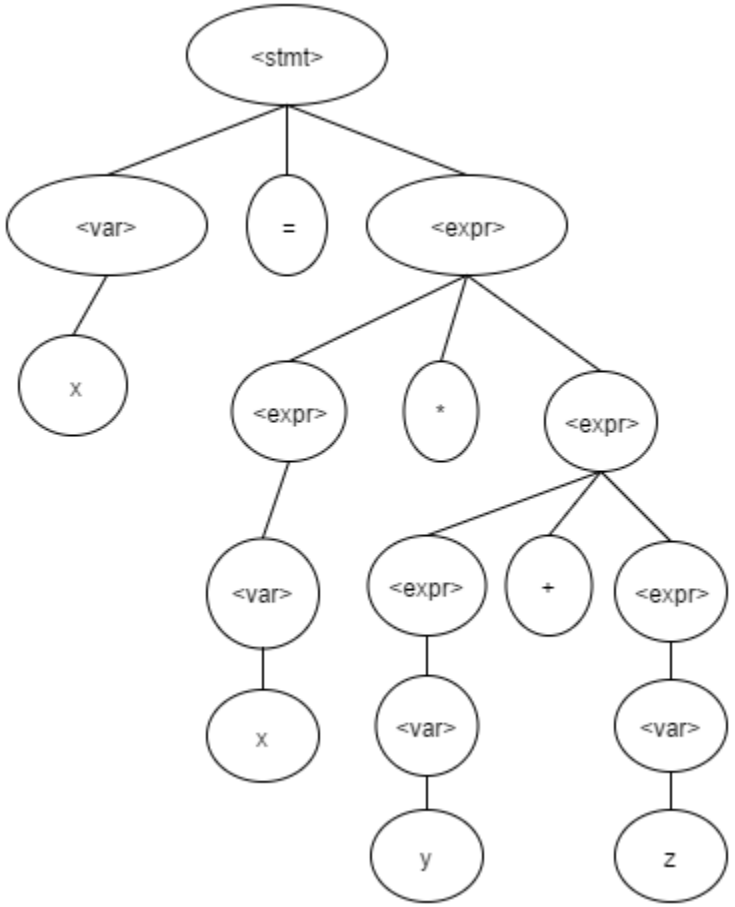
  x = x * y + z

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < expr > + < expr > \mid < expr > * < expr >$$
$$< var > \rightarrow x \mid y \mid z$$

Ambiguous Grammars

- A grammar that generates a sentence with two or more possible parse trees is called an ***ambiguous grammar***
- A grammar is ambiguous if it produces more than one left most derivation or more than one right most derivation
- Example:
  x = x * y + z





$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < var > \mid < expr > + < expr > \mid < expr > * < expr >$
$< var > \rightarrow x \mid y \mid z$

GEORGETOWN UNIVERSITY

Ambiguous Grammars

- Inspection: Why is this grammar ambiguous …

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid \boxed{< expr >} + \boxed{< expr >} \mid \boxed{< expr >} * \boxed{< expr >}$$
$$< var > \rightarrow x \mid y \mid z$$

- The <expr> non terminal can be expanded on either side of the + or * operator. We can derive the either side first possibly giving rise to two distinct parse trees

- Note: Syntactically this may be OK, but in practice, this may cause semantic issues

*Ambiguous Grammar : English**

- I see Bob on the roof with binoculars.

Disambiguating Grammars

- This grammar is ambiguous as there is the same non-terminal on either side of the '+' and '*' operators, which may permit multiple parse trees

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < expr > \ + < expr > \mid < expr > * < expr >$$
$$< var > \rightarrow x \mid y \mid z$$

- Fix:
  - Change the ***associativity*** of the operator within the productions.
  - Make each operator either *left associative* or *right associative*, but not both.
  - Associativity can dictate derivation and disambiguate a grammar.

Operator Associativity

- A production is left recursive if its LHS appears at the beginning of the RHS. This implies left associativity.
  - The following grammar is left associative and unambiguous.

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > * < var >$$
$$< var > \rightarrow x \mid y \mid z$$

- A production is right recursive if its LHS appears at the beginning of the RHS. This implies right associativity.
  - The following grammar is right associative and unambiguous.

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < var > + < expr > \mid < var > * < expr >$$
$$< var > \rightarrow x \mid y \mid z$$

GEORGETOWN UNIVERSITY

Parse Tree Example

- Lets attempt our previous example using a non-ambiguous grammar.
  - Example: x = x * y + z

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > * < var >$$
$$< var > \rightarrow x \mid y \mid z$$

Does associativity solve all language issues?

- NO – Associativity may help with ambiguity issues, but *semantic issues* may still remain.
  - Example 1: operator precedence.
  - Example 2: the dangling “else” problem.

- Operator Precedence Concern: apply semantics to “ x = x + y * z ”
  - Semantic Goal: evaluate the multiplication first!
  - Note: Semantics is applied to the parsed structure – parse tree.
    - The lower in the parse tree, the higher the precedence. That is, items lower on the parse tree (along the same branch) are evaluated first

Example: Operator Precedence

- Parse " x = x + y * z " , using the following grammar

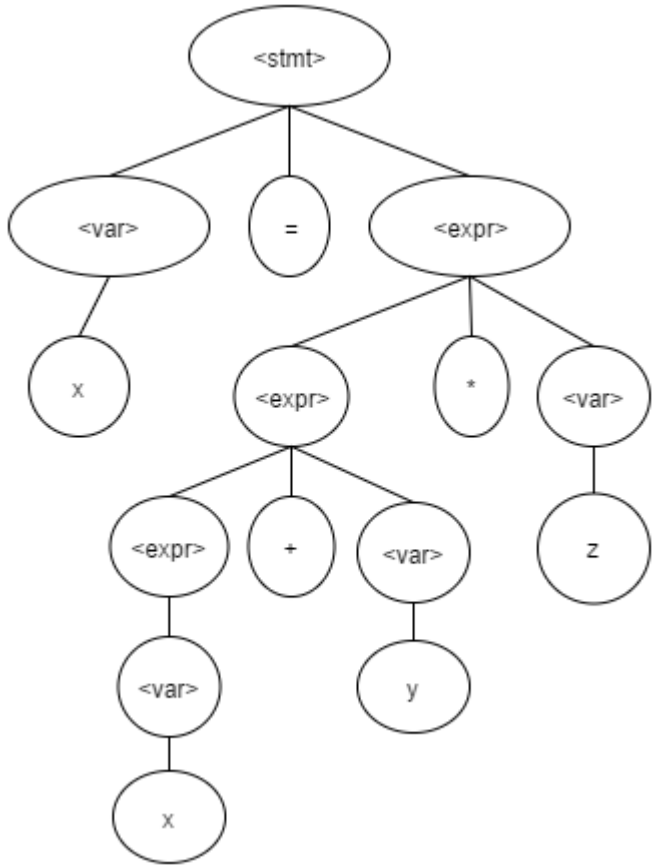
$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > * < var >$$
$$< var > \rightarrow x \mid y \mid z$$

- Using Left Associative Productions leads to improper parse tree
  - *Addition is lower in the parse tree and is evaluated first*

- Question
  - Does using a Right Associative production fix the issue?
  - For this one example, yes, but for the general case **NO!**
  - Associativity will change the resulting parse tree based on the order each of the operators or symbols, but cannot be used alone to enforce operator precedence.

- Perspective: Parsing and evaluation as a depth first traversal
  - Post Order!
  - Parsing can be viewed as a downward operation – traversing the tree downward. Applying semantics can be viewed as an upward operation, performed when traversing back up the tree.

- So … How can we enforce operator precedence?



GEORGETOWN UNIVERSITY

Operator Precedence

- How can we enforce operator precedence?
  - With the addition of non-terminals and rules

- Example: Require operands of the '+' operator to consist of variables or products of variables (but not the reverse – do not allow operands of products to consist of additions of variables). This will ensure that multiplication is lower on the parse tree.

No enforced precedence

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < var > \mid < expr > + < var > \mid < expr > * < var >$
$< var > \rightarrow x \mid y \mid z$

'*' has precedence over '+'

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term >$
$< term > \rightarrow < var > \mid < term > * < var >$
$< var > \rightarrow x \mid y \mid z$

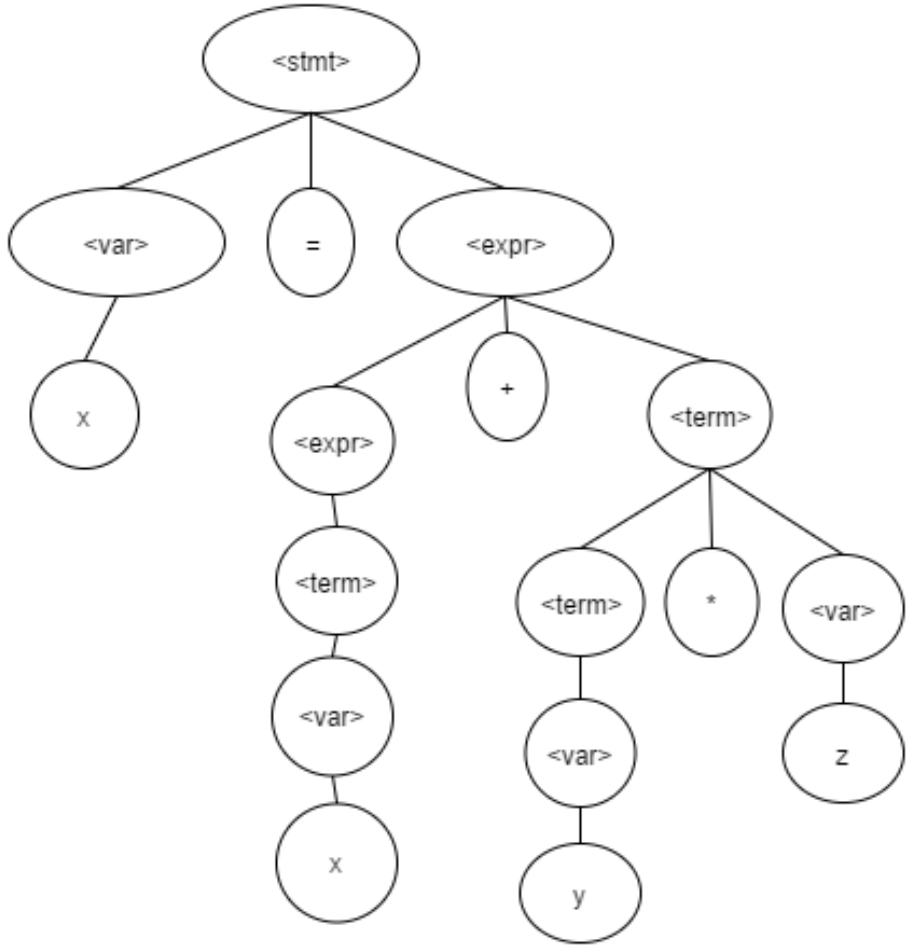
GEORGETOWN UNIVERSITY

Example: Operator Precedence

- Parse “ x = x + y * z ” , using the following grammar

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term >$$
$$< term > \rightarrow < var > \mid < term > * < var >$$
$$< var > \rightarrow x \mid y \mid z$$

- Since the operands of expressions are composed of terms, (and terms are not composed of expressions), terms will appear lower in the parse tree



In Class Example

- Given the following grammar, parse the following sentences (if possible).

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < var > \mid < expr > - < var >$$
$$< term > \rightarrow < var > \mid < term > * < var > \mid < term > / < var >$$
$$< var > \rightarrow x \mid y \mid z$$

- Sentences:

  x = x * y / z

  y = z + x / y

  z = -x / y + z

Example: non-ideal grammar

- Using the following grammar, try to parse sentences

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < term > + < var > \mid < expr > - < var >$$
$$< term > \rightarrow < var > \mid < var > * < var > \mid < term > / < var >$$
$$< var > \rightarrow x \mid y \mid z$$

- Which of these sentences are in the language? (ie which can we parse)

  x = x * y / z

  y = z + x / y

  z = - x / y + z

- What is “wrong” with this grammar? How can we “fix” it?

Designing a Grammar

- Each operator with different precedence should have its own non-terminal with production(s)
- The productions should be designed to enforce the correct precedence
- Example: Add the unary negation operator to the existing grammar

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < var > \mid < term > * < var > \mid < term > / < var >$$
$$< var > \rightarrow x \mid y \mid z$$

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow x \mid y \mid z$$

GEORGETOWN UNIVERSITY

In-Class Example

- Everyone try: Update the following grammar to allow for parenthesis within a mathematical expression
  - Questions to ask yourself
    - What precedence level is a parenthetical expression?
    - Given that, where should this new rule go within the BNF?
      - Should we add a new non-terminal and production, or a new production for an existing non-terminal?
      - What is the form of all possible parenthetical statements, ie what is the form of the production?

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow x \mid y \mid z$$

GEORGETOWN UNIVERSITY

Example (cont)

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid -< var >$
$< var > \rightarrow x \mid y \mid z$

- Placing the parenthetical near the top of the BNF hierarchy is intuitive in many ways; HOWEVER, given its general use and *semantics*, this construct should be lower in the hierarchy.
- Why? BNF is a hierarchical construct, building a abstractions out of lower level abstractions. The lower the abstraction the more atomic (indivisible) … in the context of operators: the higher the precedence.

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
| (<expr>)
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid -< var >$
$< var > \rightarrow x \mid y \mid z$

- Will this generate?:

  x = x + (y + z)

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
| (<expr>) | <expr> + (<expr>) | (<expr>) + <expr> | …
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid -< var >$
$< var > \rightarrow x \mid y \mid z$

- Will this generate?:

  x = x + (y + z) yes

  x = x * (x + y) no

GEORGETOWN UNIVERSITY

Example (cont)

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid - < neg >$
$< var > \rightarrow x \mid y \mid z$

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < p > \mid - < neg >$
$< p > \rightarrow (< expr >) \mid < var >$
$< var > \rightarrow x \mid y \mid z$

- When adding a construct to a grammar (with its own level of precedence … indivisibility), add a new production at the appropriate level of the hierarchy.

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid - < neg >$
$< var > \rightarrow x \mid y \mid z \mid$ (<expr>)

Can we generate?: x = -(x + y)
YES

$< stmt > \rightarrow < var > = < expr >$
$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$
$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$
$< neg > \rightarrow < var > \mid - < neg > \mid$ (<expr>)
$< var > \rightarrow x \mid y \mid z$

Can we generate?: x = -(x + y)
YES

GEORGETOWN UNIVERSITY

Else-Matching: the dangling else problem

- Consider the following BNF for <if stmts>

$$< if\_stmt > \rightarrow if(< expr >) < stmt > \mid$$
$$if(< expr >) < stmt > else < stmt >$$

$$< stmt > \rightarrow < if\_stmt > \mid < assign >$$

…

- Consider the following code snippet and parse using grammar above:

```
if( x )
if( y )
        x = x + y
else
        y = x + y
```

Else-Matching: the dangling else problem

- Consider the following BNF for <if stmts>

$$< if\_stmt > \rightarrow if(< expr >) < stmt > \mid$$
$$if(< expr >) < stmt >$$
$$else < stmt >$$

$$< stmt > \rightarrow < if\_stmt > \mid < assign >$$

…

- Consider the following code snippet and parse using grammar above:

```
if( x )
if( y )
        x = x + y
else
        y = x + y
```

GEORGETOWN UNIVERSITY

Else-Matching: the dangling else problem (solution match else with nearest if)

$$< if\_stmt \quad > \rightarrow \quad < matched > \mid < unmatched >$$

$$< matched > \ if(< expr >) < matched > else < matched > \ \mid$$

$$< assign >$$

$$< unmatched > \rightarrow if(< expr >) < if\_stmt > \ \mid$$

$$if(< expr >) < matched > else < unmatched >$$

- If time, try to parse the following

```
if( x )
if( y )
    x = x + y
else
    y = x + y
```

GEORGETOWN UNIVERSITY

If-else discussion

- Observation: the dangling else problem arises from the combination of a matched and unmatched if_stmt . Allowing “flexible” matchings would produce an ambiguous grammar

- One can add a disambiguating rule to a grammar to enforce a matching scheme / rule. Note: This is not the only solution.

- What else might one do to disambiguate such a grammar?

Example: A more complex grammar with PL like syntax

- Note operator precedence and composition rules of the productions.
- Exercise:
  - How can we permit sequences of statements?
  - How can we permit while loops?
  - How can we permit Boolean expressions?

$$< program > \rightarrow <stmt>$$
$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow x \mid y \mid z$$

$$< program > \rightarrow ?$$
$$< while > \rightarrow ?$$
$$< stmtList > \rightarrow ?$$
$$< stmt > \rightarrow < var > = < expr >$$
$$< bool > \rightarrow ?$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow x \mid y \mid z$$

GEORGETOWN UNIVERSITY

Parsing

- Goals
  - Find syntax errors
  - Produce or trace parse tree

- Types of parsers
  - Top-down: build tree from top down
  - Bottom-up: build tree from bottom up
    - Later!

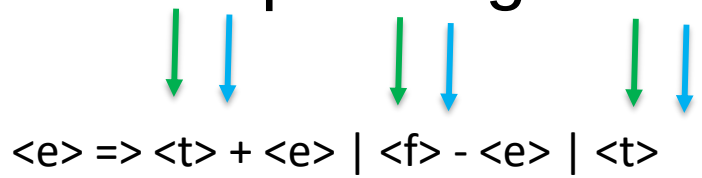
Top-Down Parsers

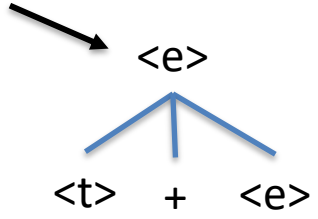
- Produces a hierarchical representation of a left-most derivation
  - Traces or builds tree in pre-order
- Recursive Descent Parsing
  - A Top – Down Parser implemented such that a set of recursive methods is used to parse the input.
- Predictive Parser – A recursive descent parser, that may require a lookahead symbol(s) to correctly predict a production at various stages of a derivation.

Predictive Parsers

- Crux: Must choose / ***predict*** correct RHS when expanding a node in the parse tree.
  - Eg: 3+4

  - For efficient design best to use next token or lookahead only one token





Top Down Parsing

- Formally, a top down parser must do the following:
  - Given a sentence in the form xAα , the parser must choose the correct A=> rule .
    - Notation
      - Lowercase: terminal sequence
      - Uppercase: non-terminal
      - Greek: sequence of terminals and non-terminals
  - Its best if this decision can be made based on the first tokens of A's RHSs.
    - This is true if all first terminals of A productions are different.
- Implementation of Top Down Parser
  - Recursive decent
  - Table driven
    - Later!

Left Recursion and Top Down Parsing: (also - Using BNF as a design template)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <term> * <factor> | …
*/
void term() {

/* Parse the first factor */
 term();

/* As long as the next token is * or /,
   next token and parse the next factor */
  if(nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
} /* End of function term */
```

Recursive Decent Parsing

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {

/* Parse the first factor */
  factor();

/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
} /* End of function term */
```

Limitations of Recursive Descent Parsers

- Left Recursion
  - Modify Grammar rules to remove direct left recursion (or similarly use appropriate EBNF)

    For each nonterminal, A,

    1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ where none of the β's begins with A
    2. Replace the original A-rules with

       $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

       $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

    NOTE: operator association not explicitly specified in this format (can still be left associative). We will see during implementation

- Prediction
  - Pairwise disjointedness: assure first terminals are different for each A Production
    - The inability to determine the correct RHS on the basis of one token of lookahead
    - FIRST()
  - Left factoring

Eliminate Left Recursion Example

E => E + T | E – T | T

T => 0 | 1 | 3 | … | 9

E => TE’

E’ => +TE’ | -TE’ | ε

T => 0 | 1 | 3 | … | 9

For each nonterminal, A,

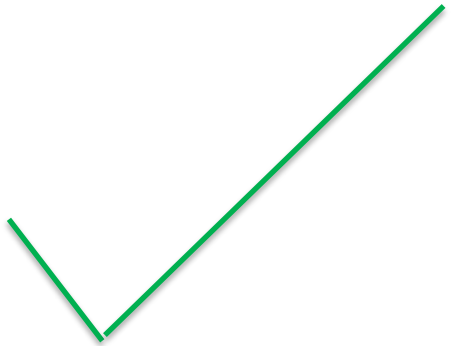
1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
   where none of the β‘s begins with A
2. Replace the original A-rules with
   $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
   $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Prediction: FIRST to determine pairwise disjointedness

- Def: FIRST(α) = {a | α =>* aβ }
  (If α =>* ε, ε is in FIRST(α))
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that
    $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$

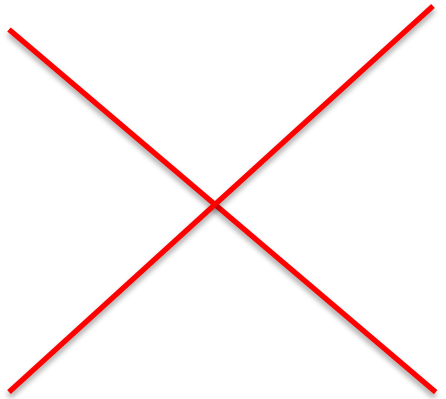
- EG
  A => aB | bB | BA
  B => c | d

  FIRST (aB) = {a}
  FIRST (bB) = {b}
  FIRST (B) = {c,d}



A => aB | bB | BA
B => c | a

FIRST (aB) = {a}
FIRST (bB) = {b}
FIRST (B) = {c,a}



Left Factoring for Prediction

- Sometimes we can resolve a nondisjointed FIRST through left factoring by factoring out similar FIRSTs

<variable> → identifier | identifier [<expression>]

<variable> → identifier <new>

<new> → ε | [<expression>]

EBNF notation

- Extended BNF
  - Uses curly braces to indicate repetition (in place of explicit recursion)
  - Uses (|) to indicate logical OR applied to individual symbols rather than entire RHSs

- Example:

  BNF

  $$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$

  EBNF

  $$< expr > \rightarrow < term > \{ \ (+ \ \mid \ - \ ) \quad < term > \ \}$$

Regular Languages

- Both context free languages and regular languages are useful in programming languages (Chomsky)

- We have seen that BNF is a great way to formalize context free grammars (to define a context free language)

$$
\begin{aligned}
&< stmt > \rightarrow < var > = < expr > \\
&< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term > \\
&< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg > \\
&< neg > \rightarrow < var > \mid -< var > \\
&< var > \rightarrow x \mid y \mid z
\end{aligned}
$$

- BNF defines sequences of tokens (sentences) for a languages.

- **<u>Regular languages are often used to model the tokens of a language</u>**

Use of Language Models for PLs

- In general: RLs are used to model the tokens
  - Each token class is a regular language. *The input alphabet is the set of all characters.*
  - The job of the tokenizer is to “recognize” each token class
    - In a left to right scan of the input, the tokenizer can recognize the beginning and end of each token (given the rules: fsm of each token class). Thus creating a token list.
- In general: CFLs are used to model the programming language (int terms of the tokens / lexemes)
  - Each programming language is a CFL. *The input alphabet is the set of all tokens.*
  - The job of the parser is to “recognize” and produce a parse tree.
    - In a left to right scan of the input (token sequence), the parser will determine if the token sequence conforms with the rules of the grammar, thus creating a parse tree.

Usage of RLs and CFLs in PLs (revisited)
Use RLs to define rules for tokens. Use CFL for PL rules.

- Could you use BNF to define the set of all possible variables or ints? YES. But instead we will use regular languages and regular expressions. (More on this later)

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow see\ regular\ expression\ for\ variables.$$
$$< int > \rightarrow see\ regular\ expression\ for\ integers.$$

Groups of abstractions: compositions of terminals and nonterminals. We use CFGs to model these constructs.

It is intuitive to define our language in terms of its smallest syntactic unit (the most integral units that have meaning). However – these "lexemes" consist of parts (chars) themselves. Solution: Use 1 model for the (smallest syntactic units) lexemes and 1 model for the language.

Groups of tokens (sets of terminals)

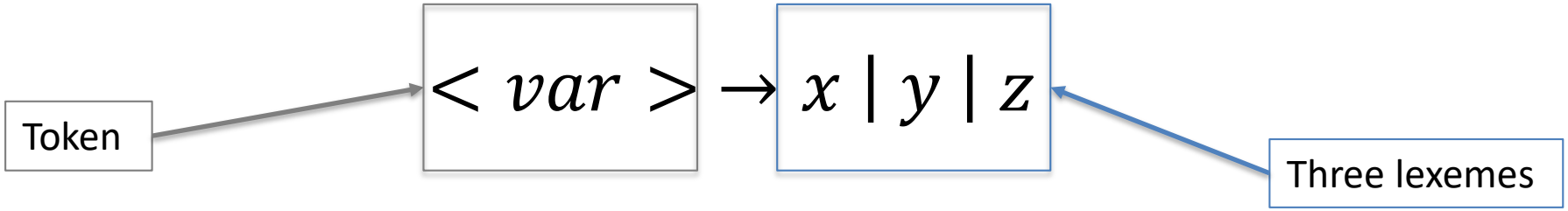
We will use regular grammars to model these constructs

GEORGETOWN UNIVERSITY

Regular Languages

- Remember: Tokens are categories of lexemes and lexemes are terminals (the smallest syntactic unit)
  - If the number of lexemes per token category are small, then we need only list them out. That is, there is no need to use a Regular Language to model all of the lexemes in the Token Class (Regular Language)

- In our previous example, there were only 3 possible variable tokens.



- If a token class has many possible lexemes, then a formal rule system can be used to help define all possible lexemes (rather than list them all out!). *For example, try listing out all possible variable names in C++.*

Use Regular Languages to define rules for tokens.

- Could you use BNF to define the set of all possible variables or ints? YES. But instead we will use regular languages and regular expressions. (More on this later)

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow see\ regular\ expression\ for\ variables.$$
$$< int > \rightarrow see\ regular\ expression\ for\ integers.$$

Groups of abstractions: compositions of terminals and nonterminals. We use CFGs to model these constructs.

Groups of tokens (sets of terminals)

We will use regular grammars to model these constructs

GEORGETOWN UNIVERSITY

Regular Languages

- Regular Languages used to model Tokens

- Regular Expression is a generator for a regular language (it defines a regular language)
  - Consists of characters and regular operations
  - Just as BNF is a generator and define a context free language.

- Regular Operations:
  - Concatenation
  - Repetition
    Repeat 0 or more times: *
    Repeat 1 or more times: +
  - Selection: |

  - Other common symbols used
    Optional: ?
    Any character: .
    Short hand for 1 element in a set: [ ]

Appendix

GEORGETOWN UNIVERSITY