# COMP 520 - Compilers

## Lecture 2 (Jan 13, 2022)

## *Specification of Programming Languages*

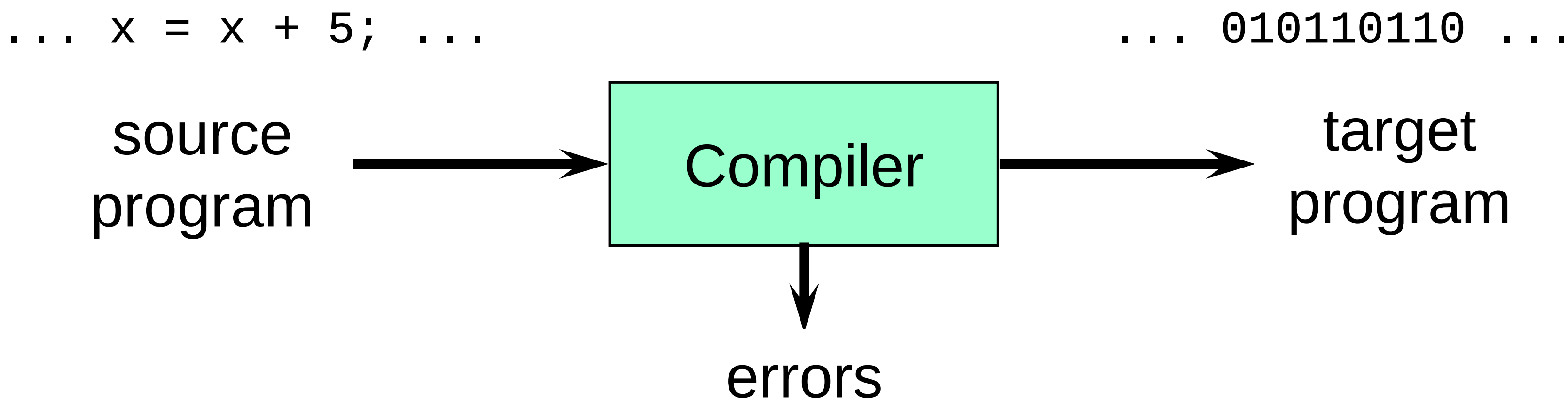
- Please turn in at the front of the room
  - Written assignment 1 (red folder)

- For Tue 1/18
  - Skim PLPJ Chapter 3 (pp 55 – 70)
  - Study PLPJ Chapter 4 Secns 4.1, 4.2 (pp 73 – 83)
  - … then look at PA1 again

# Today's Topics

- Formal description of programming languages
  - syntactic description:  context free grammars
  - concrete and abstract syntax
  - contextual constraints
  - semantics
- Phases of compilation
  - Compiler project timeline
- Tools and machines needed in this class
- Review of WA1
- Quick look at PA1

# Functional description of a compiler




- source and target programs
  - are expressed as a sequence of characters (source) or instruction codes (target)
- translation of a programming language
  - split into two parts
    - syntax – the structure of “sentences” in the language
    - semantics – the meaning of “sentences” in the language
- if we can precisely describe the source and target languages …
  - the compiler is a meaning-preserving translation from sentences in the source language to sentences in the target language

# Syntactic description of a language

- A simple *context-free grammar* (CFG) describes a few English sentences

| | | |
|---|---|---|
| *Sentence* | ::= | Subject Predicate Object |
| Subject | ::= | Article Noun |
| Predicate | ::= | Verb |
| Object | ::= | Article Noun |
| Article | ::= | `a` \| `the` |
| Noun | ::= | `dog` \| `cat` \| `mice` |
| Verb | ::= | `chase` \| `chases` |

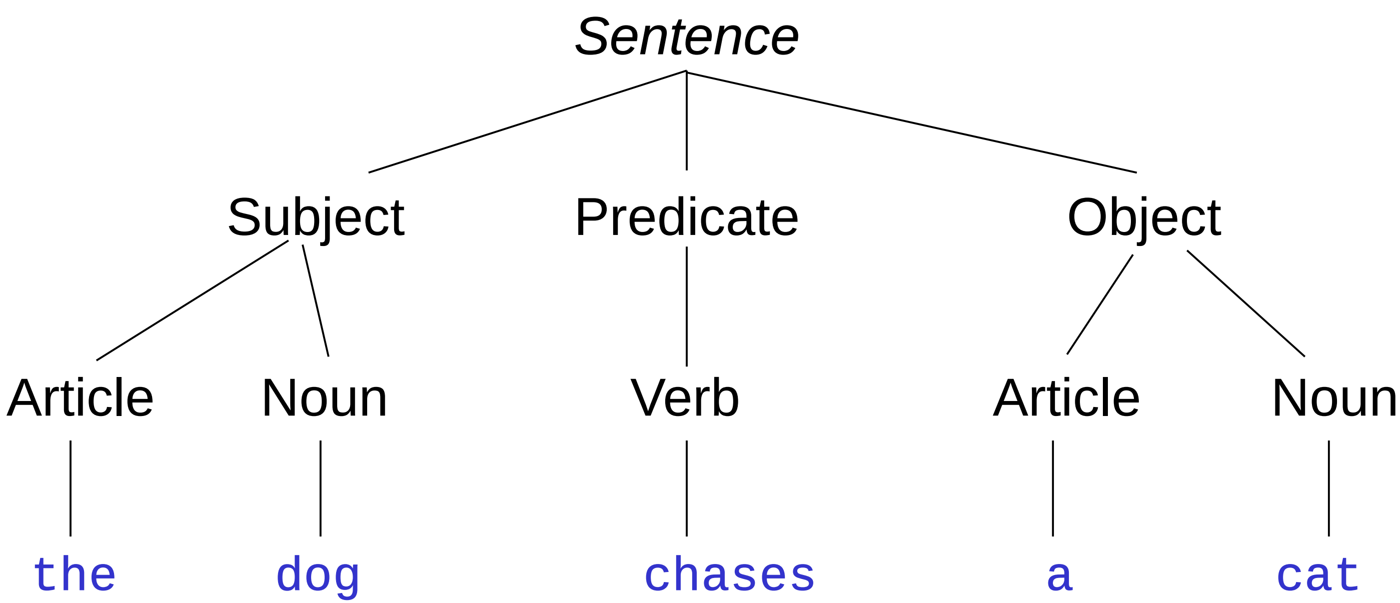
- Components of a CFG
  - Terminals {`a`, `the`, `dog`, `cat`, `mice`, `chase`, `chases`}
  - Nonterminals {*Sentence*, Subject, Predicate, Article, Noun, Verb}
  - Start nonterminal *Sentence*
  - Rules (shown above)

- *Language* generated by a context free grammar (CFG)
  - is a set of sentences
  - each sentence
    - composed entirely of terminals
    - can be generated by repeated application of the rules commencing from the start nonterminal

# *Derivation* of a sentence using a CFG

*Sentence* ::= Subject Predicate Object
Subject ::= Article Noun
Predicate ::= Verb
Object ::= Article Noun

Article ::= `a` | `the`
Noun ::= `dog` | `cat` | `mice`
Verb ::= `chase` | `chases`



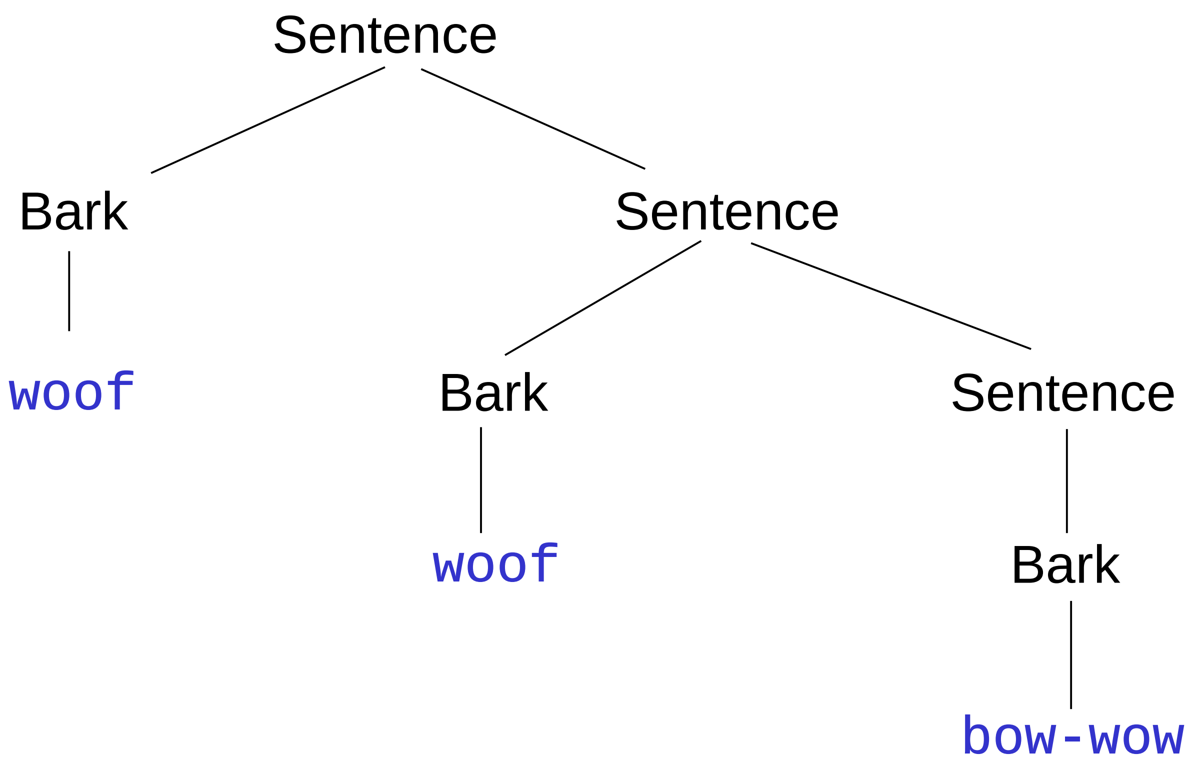

A *syntax tree* records the derivation of a sentence

# Arbitrarily long sentences: CFG for dog language

- Recursion in the rules

  Sentence ::= Bark | Bark Sentence

  Bark ::= `woof` | `bow–wow`

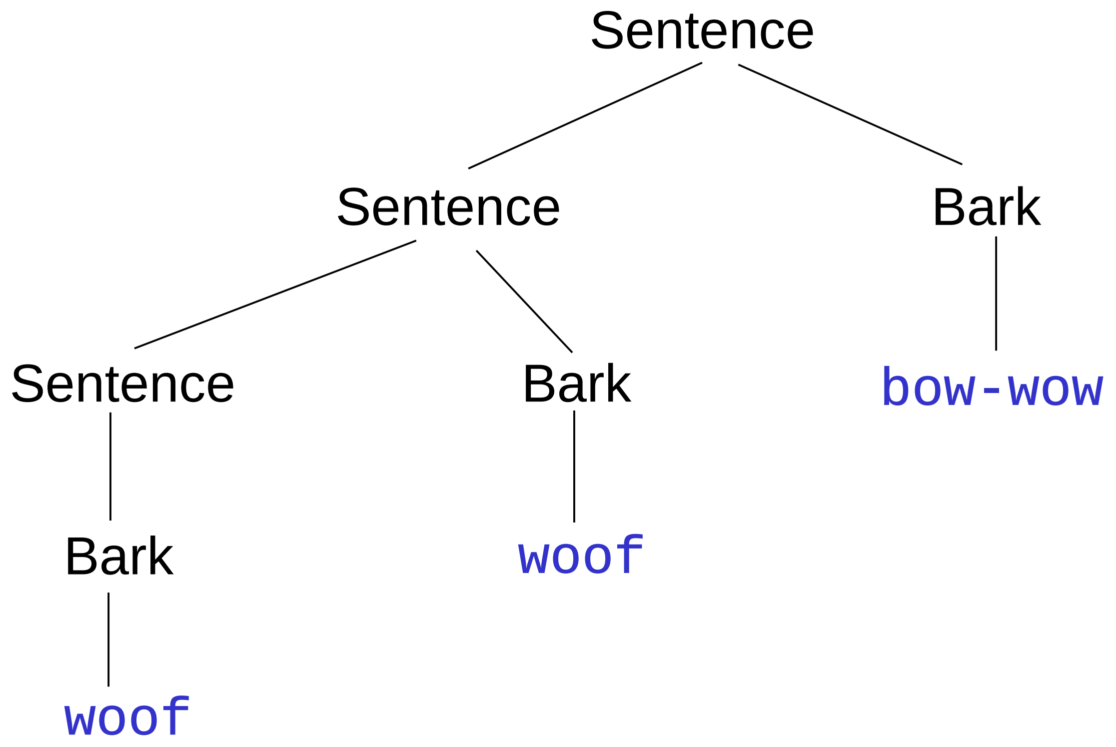


# Another CFG for dog language

- Same language, different grammar

  Sentence ::= Bark | Sentence Bark

  Bark ::= `woof` | `bow–wow`

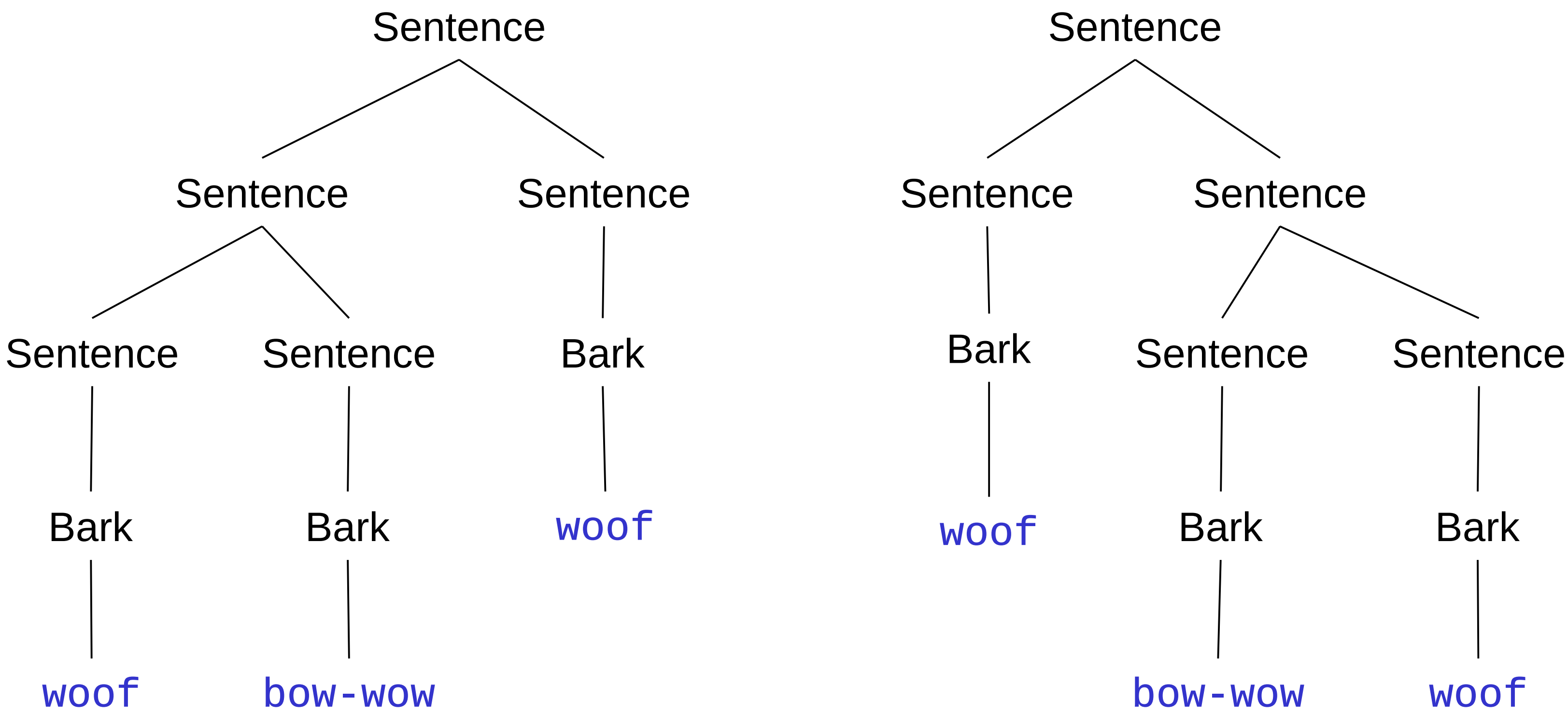


# Yet another CFG for dog language

- A grammar with multiple syntax trees for the same sentence!

  Sentence ::= Bark | Sentence Sentence

  Bark ::= `woof` | `bow–wow`




This grammar is *ambiguous*.  Ambiguous grammars are problematic for language specifications

# Uses and limitations of CF Grammars

- CF grammars describe a superset of meaningful sentences
  - examples of incorrect sentences
    - The mice chases a dog
    - The dog chases a mice
  - we need additional constraints to determine validity
    - these are outside of the CFG framework

- CF grammars can be used to find the structure of a sentence
  - A *parser* is used to find the syntax tree for a sentence
  - The syntax tree describes the sentence structure

- CF grammars for programming languages
  - ensure a unique syntax tree for each sentence
    - no *ambiguous* grammars
  - can be efficiently parsed
    - using parsers with time complexity linear in sentence length
    - Not all CFGs can be efficiently parsed

# A CFG for a programming language

- Expressions in Mini-Triangle

  *Exp* ::= PrimExp | Exp Oper PrimExp

  PrimExp ::= *`intlit`* | *`id`* | Oper PrimExp | `(` Exp `)`

  Oper ::= `+` | `-` | `*` | `/` | `<` | `>` | `=`

- Special interpretation of terminals
  - *`intlit`* stands for any integer
  - *`id`* stands for any identifier
  - blanks are ignored

- Construct syntax trees for

```
-20
x + y
x – y > 0
0 < x – y
0 < (x – y)
m >= n
```

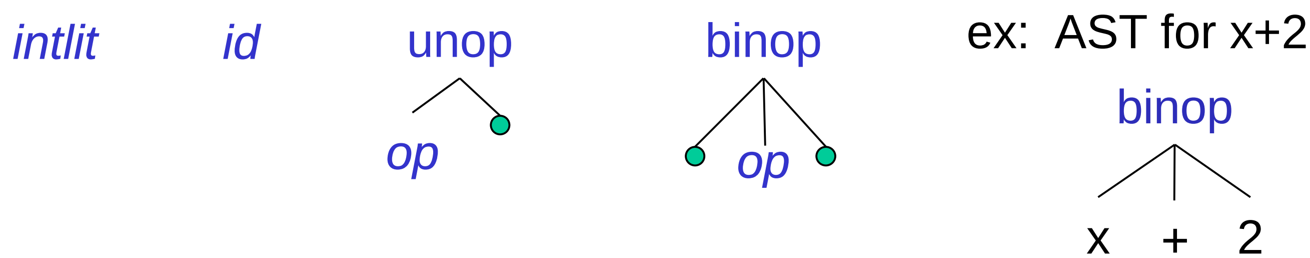
# Abstract Syntax Trees (ASTs)

- The problem with syntax trees
  - unnecessary detail to control derivation interferes with utility

- *Abstract syntax trees* for mini-Triangle expressions
  - *abstract* structure of expressions

    Exp ::= *`intlit`* | *`id`* | *`op`* Exp | Exp *`op`* Exp

  - let ● represent an Exp. Substitution choices for the four rules above are:




- ASTs are a better representation of the "meaning" of a program
  - so why not parse using AST grammar?
  - construct the abstract syntax tree for `x – y = 0`

# AST for commands

- A piece of the mini-Triangle grammar for commands

  Program ::= Cmd

  Cmd ::= *`id`* `:=` Exp | `let` Decl `in` Cmd

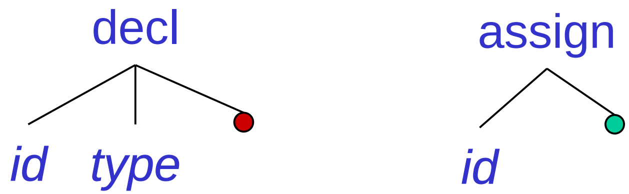
  Decl ::= `var` *`id`* `:` *type*

- Abstract syntax trees for mini-Triangle commands
  - *abstract* structure of commands

  Cmd ::= *id* *type* Cmd      declaration (decl)

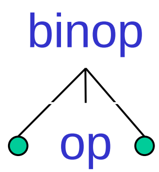
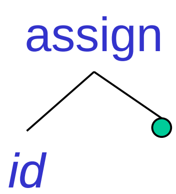
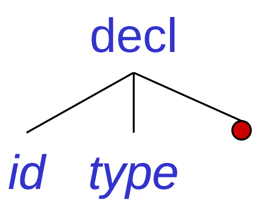
        | *id* Exp      assignment (assign)

  - let represent a Cmd, and represent an Exp, possible Cmd ASTs:



# Semantics

- Typically an evaluation rule for each kind of node in an AST

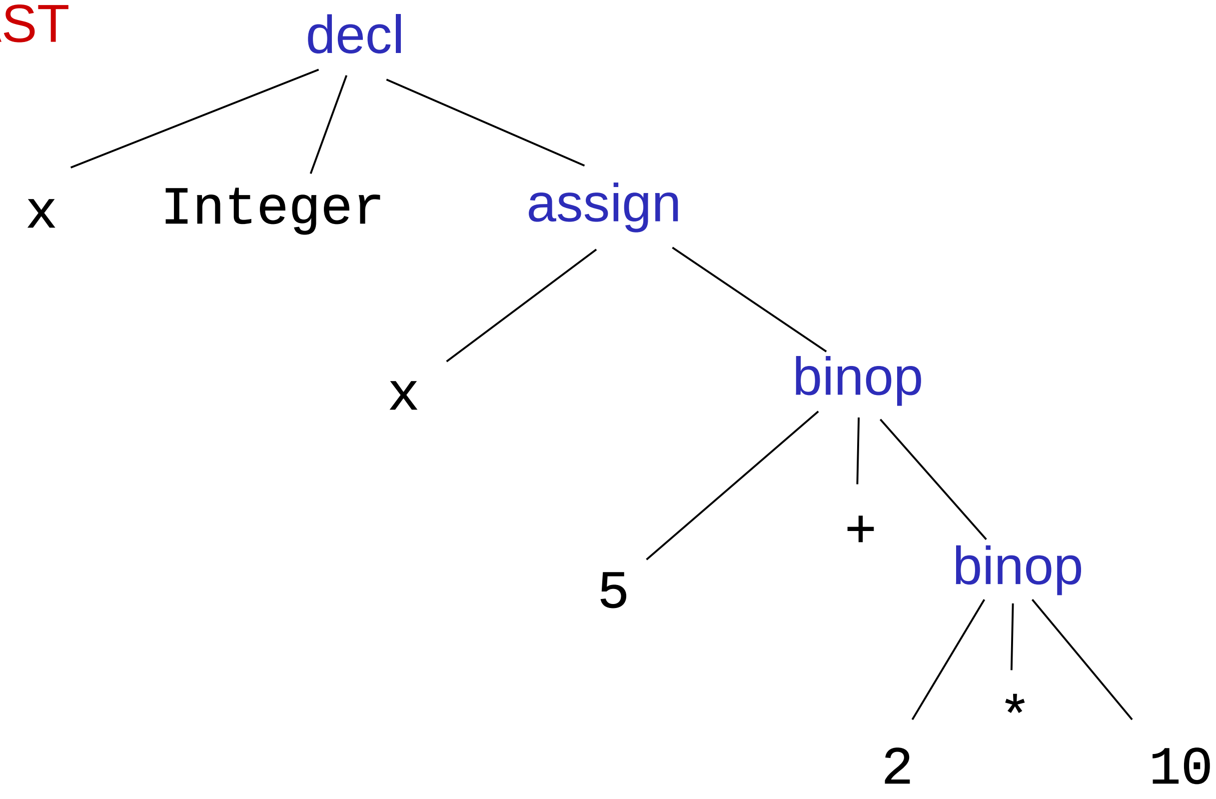
| AST node | Evaluation rule |
|---|---|
|   | evaluate left Exp and then evaluate right Exp and then combine the results using *op* (+, -, etc.) |
|   | evaluate Exp and store result into variable id |
|   | create space for variable id and then evaluate Cmd |

- Example
  - `let var x: Integer in x := 5 + (2 * 10)`

# Semantics: AST + evaluation rules ⇒ meaning

- Mini-triangle program
  - `let var x: Integer in x := 5 + (2 * 10)`

- Corresponding AST



# Contextual constraints

- The mini-Triangle grammar for commands

  Program ::= Cmd

  Cmd ::= *id* `:=` Exp | `let` Decl `in` Cmd

  Decl ::= `var` *id* `:` *type*

- The following can be derived using this grammar

```
let  var x: Integer  in   x := 5 + 5
let  var x: Integer  in   y := 5 + 5
let  var x: Integer  in   x := 5 > 3
```

- contextual constraints must be added to ensure
  - declaration before use of variables
  - type of variable appropriate for operation
  - type of value appropriate for assignment

# Summary: specification of a programming language

- Syntax (formal)
  - context free grammar
  - additional lexical rules (comments, whitespace in text. etc.)

- Additional contextual constraints (can be made formal)
  - Identifier declaration and reference rules
  - Type rules

- Semantics
  - Operational definition of evaluation

# Phases of compilation

Front end: lexical analysis, syntactic analysis, contextual analysis

Back end: optimization, code generation

Unified in PLPJ text: lexical analysis, syntactic analysis

| Phase | Input | Output |
|---|---|---|
| lexical analysis (scanner) | character stream | tokens |
| syntactic analysis (parser) | tokens | abstract syntax tree (AST) |
| Contextual analysis (type checker / identifier resolution) | AST | decorated AST |
| (*optional*) Optimization | decorated AST | decorated AST |
| Code generation | decorated AST | machine instructions |

# Compiler Project timeline

- Project timeline (may be subject to change)

| project phase | assigned | due | time |
|---|---|---|---|
| syntactic analysis | Thu Jan 13 | Mon Jan 31 | (18 days) |
| AST construction | Tue Feb 1 | Mon Feb 21 | (20 days) |
| contextual analysis | Tue Feb 22 | Mon Mar 21 | (22 days + brk) |
| code gen / execution | Tue Mar 22 | Mon Apr 11 | (21 days) |
| complete project | Tue Apr 12 | Thu Apr 28 | (16 days) |

*specification may be available before the preceding due date

- Team commitments
  - send to me by email, by due date of first phase (Mon Jan 31)
  - they are binding for remainder of project
  - team project earns credit at 80% rate

# CS machine access

- Development machines
  - We are using Java, so lots of choice
    - Windows, Mac OS, or Linux - all will run Java and Eclipse
    - most folks prefer to develop on their own machine
  - COMP 520 server (details to follow)
    - comp520-1sp22.cs.unc.edu  (accessible from UNC network)
      - login with your onyen
    - you will submit your project checkpoints and receive scores on this machine
- Tools
  - Windows
    - secureCRT provides terminal windows for logins across network
      - can drag and drop files between your machine and cs machines using built-in zmodem protocol
      - Download from http://software.unc.edu/
    - alternatively use sftp
  - Mac, Linux
    - Use unix tools:
      - terminal/ssh  (for login)
      - scp (to move files or hierarchies)

# Development environment

- If you are already set up to run Java with Eclipse
  - generally this will be sufficient for our project

- Java SE development kit (JDK)
  - Use Java 8

- Eclipse for Java Developers
  - version 4.5 (Mars) or later for Java Developers (latest is fine)
  - http://www.eclipse.org

# Assignments

- Compiler Project PA1
  - Review handout online

# Triangle Examples (1)

- Triangle commands
  - Conditional command
  - Scope command

```
if x > y then
   let const xcopy ~ x
   in
      begin
         x := y;
         y := xcopy
      end
else
```

# Triangle Examples (2)

- Triangle expressions
  - Scoped expression
  - Conditional expression

```
let
   const taxable ~ if income > allowance
                   then income – allowance
                   else 0
in
   taxable / 4
```

# Triangle Examples (3)

- Triangle types, procedures, and operators
  - Named type
  - Function declaration
  - Operator declaration

```
type Point ~ record
                x: Integer, y: Integer
             end;

func projection (pt: Point) : Point ~
   {x ~ pt.x, y ~ 0 – pt.y};

func /\ (b1: Boolean, b2 : Boolean) : Boolean ~
   if b1 then b2 else false
```