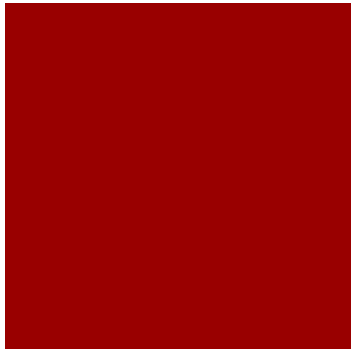
# COM S 342

Recitation 09/30/2019 – 10/2/2019



# Topic

- Antlr grammar syntax

- Q&A

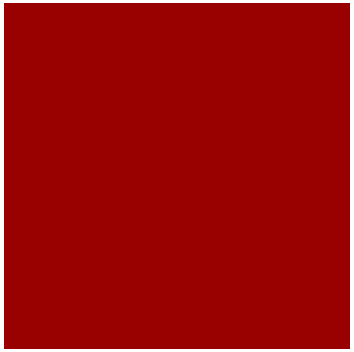
# Big Picture

- To implement a language, we have to build an application that reads sentences and reacts appropraitely to the phrases and input symbols it discovers.
- To react appropriately, the interpreter or translator has to recognize all of the valid sentences, phrases, and subphrases of a particular language.
- Programs that recognize languages are called parsers or syntax analyzers.
- We build ANTLR grammars to specify language syntax.

# Big Picture

- A grammar is a set of rules, each one expressing the structure of a phrase.
- The process of grouping characters into words or symbols(tokens) is called lexical analysis or simply tokenizing
- We call a program that tokenizes the input a lexer
- Tokens consist of at least two pieces of information: the token type and the text matched for that token by the lexer.

# Big Picture



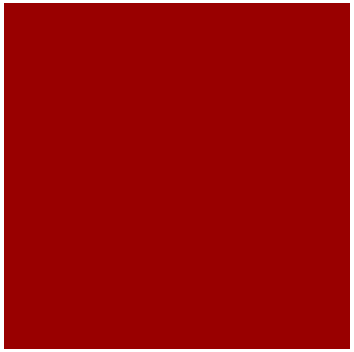
- The second stage is the actual parser and feeds off of these tokens to recognize the sentence structure.
- ANTLR-generated parsers build a data structure called a parse tree or syntax tree that records how the parser recognized the structure of the input sentence.

# Big Picture

- The interior nodes of the parse tree are phrase names that group and identify their children.
- The root node is the most abstract phrase name.
- The leaves of a parse tree are always the input tokens.

- We will talk about the part that you will need to do your homework here.

# Implementing



- The ANTLR tool generates recursive-descent parsers from grammar rules.
- Recursive-descent parsers are one kind of top-down parser implemntation.
- The rule we invoke first, the start symbol, becomes the root of the parse tree.

# Core Grammar Notation

| Syntax | Description |
| --- | --- |
| x | Match token, rule reference, or subrule x |
| x y … z | Match a sequence of rule elements |
| (…\|…\|…) | Subrule with multiple alternatives |
| x? | Match x or skip it |
| x* | Match x zero or moremes |
| x+ | Match x one or more times |
| r: … ; | Define rule r |
| r: … \|… \|… ; | Define rule r with multiple alternatives |

# Examples

- FOR : 'for';
- ID : [a-zA-Z]+;  //does NOT match 'for'

//ANTLR puts the implicityly generated lexical rules for literals before explicit lexer rules.

- DIGIT : [0-9];
- expr: ID '[' expr ']'
  | '(' expr ')'
  | INT
  ;

# Attributes and Actions

- Some language applications require executing application-specific code while parsing.
- To do that, we need the ability to inject code snippets, called actions.
- Actions are arbitrary chunks of code written in the target language enclosed in {...}.
- Typically, actions operate on the attributes of tokens and rule references.
- For example, we can ask for the text of a token or the text matched by an entire rule invocation

# Examples

```
stat:   e NEWLINE          {System.out.println($e.v);}
      | ID '=' e NEWLINE   {memory.put($ID.text, $e.v);}
      | NEWLINE
      ;

e returns [int v]      :
        a=e op=('*'|'/') b=e    {$v = eval($a.v, $op.type, $b.v);}
      | a=e op=('+'|'-') b=e    {$v = eval($a.v, $op.type, $b.v);}
      | '(' e ')'               {$v = $e.v;}
      | INT                     {$v = $INT.int;}
      ;
```

# Example

```
addexp returns [AddExp ast]
        locals [ArrayList<Exp> list]
        @init { $list = new ArrayList<Exp>(); } :
        '(' '+'
                e=exp { $list.add($e.ast); }
                ( e=exp { $list.add($e.ast); } )+
        ')' {$ast = new AddExp($list); }
        ;
```

```
public static class AddExp extends CompoundArithExp {
        public AddExp(List<Exp> args) {super(args); }
        public Object accept(Visitor visitor, Env env)
                        { return vsitor.visit(this, env); }
```

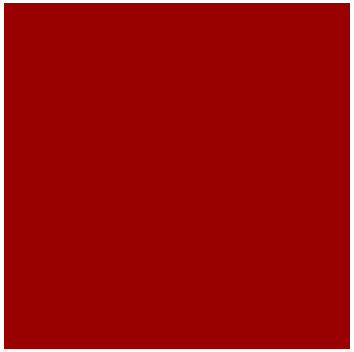
# Example

```
definedecl returns [DefineDecl ast] :
            '('   Define
                  id=Identifier
                  e=exp
            ')' { $ast = new DefineDecl($id.text, $e.ast); }
   ;

public static class DefineDecl extends Exp {
            String _name;
            Exp _value_exp;

            public DefineDecl(String name, Exp value_exp) {
                        _name = name;
                        _value_exp = value_exp;       }

            public Object accept(Visitor visitor, Env env) {  return visitor.visit(this, env);   }
            public String name() { return _name; }
            public Exp value_exp() { return _value_exp; }
}
```

# Accessing Token and Rule Attributes

- As with parameters and return values, the declarations in a locals section become fields in the rule context object.
- The init action happens before anything is matched in the rule, fegardless of how many alternatives there are.

# Example



- exp1 locals [int i]

  @init {i = 0} :

  ...

  ;

or

- exp2 locals[int i = 0} :

  ...

  ;

# Q&A