CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

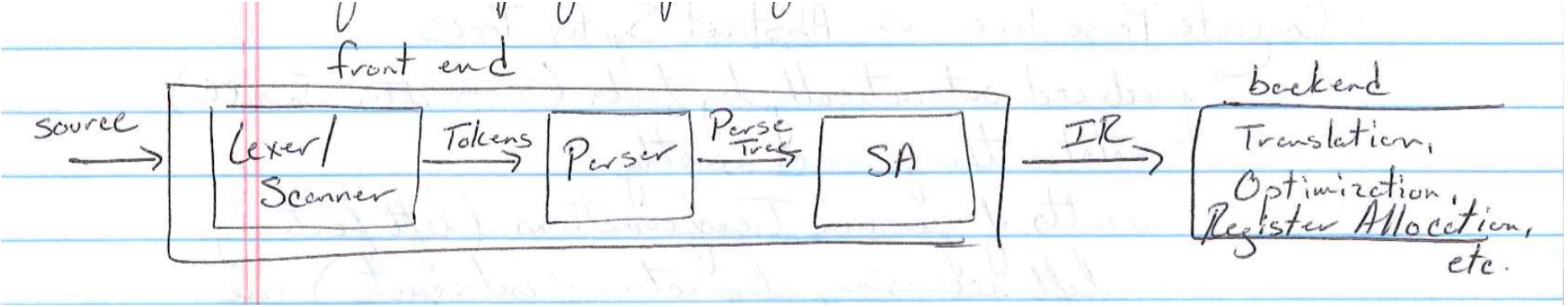
West Chester University
Dr. Richard Burns
Fall 2023

WHERE ARE WE?

- Scanning
- Parsing

NOW

- Final stages of compiler's front-end



Semantic Analyzer

- <u>Type Checking:</u> asks questions such as:
  - In a stm or exp, do operands have proper types given the operator?
    - `+` should have `int` operands, not `bool` operands
- <u>Symbol Tables:</u> asks questions such as:
  - Are identifiers declared?

Today’s Roadmap

1. Parse trees
   - (concrete) syntax trees vs. abstract syntax trees
   - AST: central data structure for all post-parsing activities
   - will be used in SA, symbol table construction, optimization, CG (Code Generation), etc.
2. Generating parse trees
   - JavaCC doesn't do this automatically. Two solutions:
   1. JTB (Java Tree Builder) tool for an automatic solution.
   2. Hand-write **semantic actions** into `.jj` file.
3. Motivate the "Visitor" SE design pattern.
   - To traverse the parse trees.
   - Each new functionality will be employed using *Visitor* pattern.

Syntax Trees

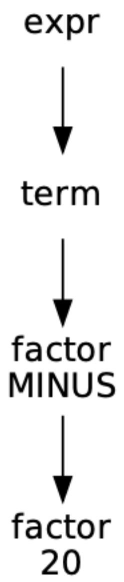
CONCRETE PARSE TREES

- Shows every detail of the parse and matches the grammar *exactly*.
- Results of grammar transformations (left factoring, left recursion, elimination of ambiguity) are carried through.

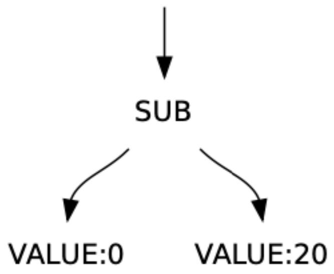
ABSTRACT SYNTAX TREES

- Clean interface between parser and later phases of compiler.
- Conveys phrase structure of source program with parsing issues resolved.

Parse Tree for `-20`



AST for `-20`



Concrete Syntax Trees

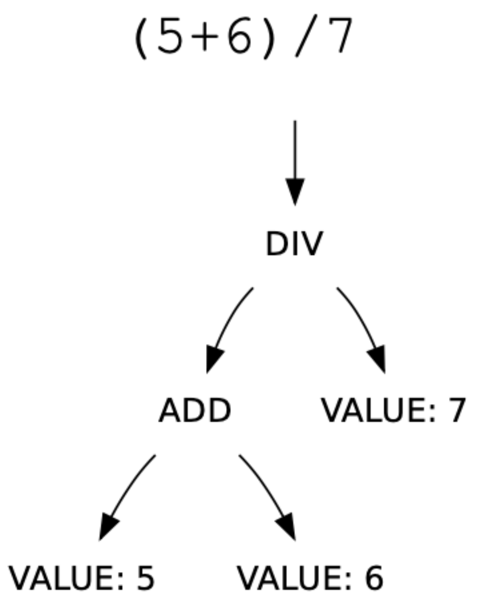
Grammar (unambiguous, left recursion removed, left factored) used in parsing "became complicated":

```
E   -> TE'        T   -> FT'       F -> num
E'  -> +TE'       T'  -> *FT'
E'  -> -TE'       T'  ->
E'  ->
```

- (Concrete) Parse Tree for `5 * 3 + 2`
- Do we really need the T' and E' productions in post-parsing compiler steps?

Abstract Syntax Trees

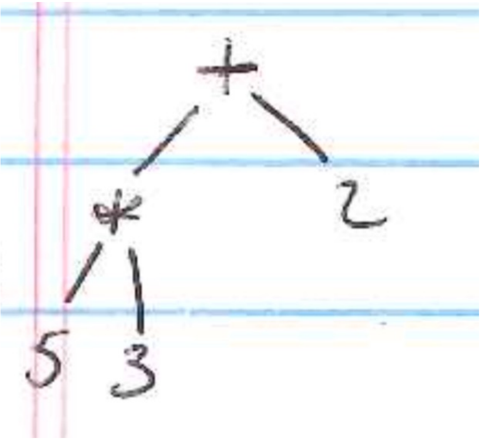
- ***Q: what is the abstract syntax of an expression?***
- Ambiguous for a grammar, but fine for abstract syntax!
- The parser produces the parse tree (using more a complicated, but unambiguous grammar)
- Parse tree disambiguates everything!
- AST is the starting point for the semantic analysis of a program



SLPInterpreter (Prog Assn #1)

This is actually a parse tree, constructed using an `OpExp` object.

```
Stm s = new OpExp(new OpExp(5, "*", 3), "+", 2)
```



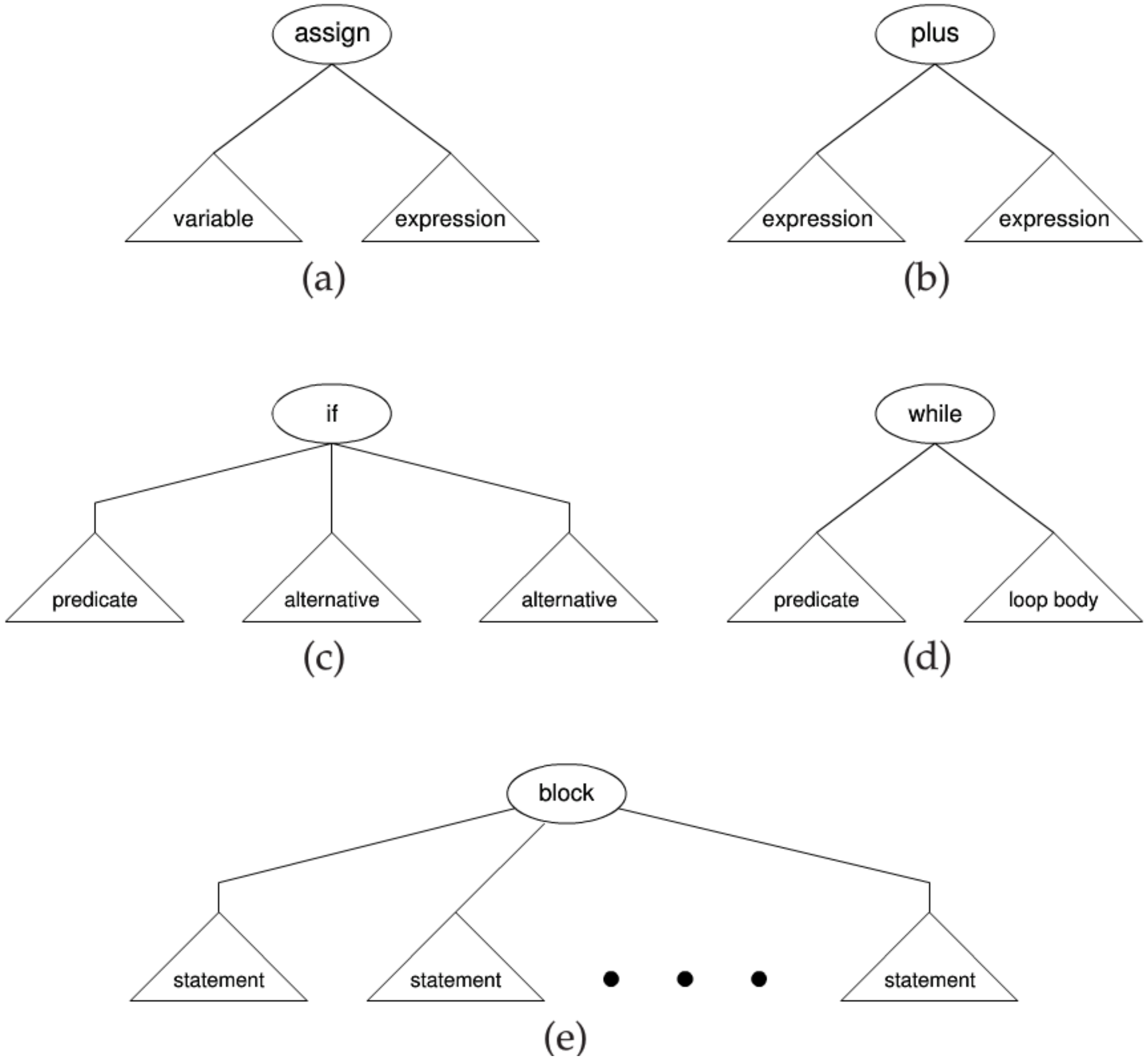
- No T' and E' were needed. We knew the * happens first in this program because it was deeper in the parse tree.
- How can we generate a parse tree like this one? -- which we will call an abstract parse tree.
- **Big idea:** an abstract parse tree disambiguates an *ambiguous grammar*.

Efficient AST Data Structures

- Some AST nodes have a *fixed* number of children
  - *Example:* binary addition and multiplication have two children
- Some P/L constructs may require an arbitrary large number of children
  - *Example:* compound statements (zero or more statements)
  - method parameter and argument lists

Good AST Design

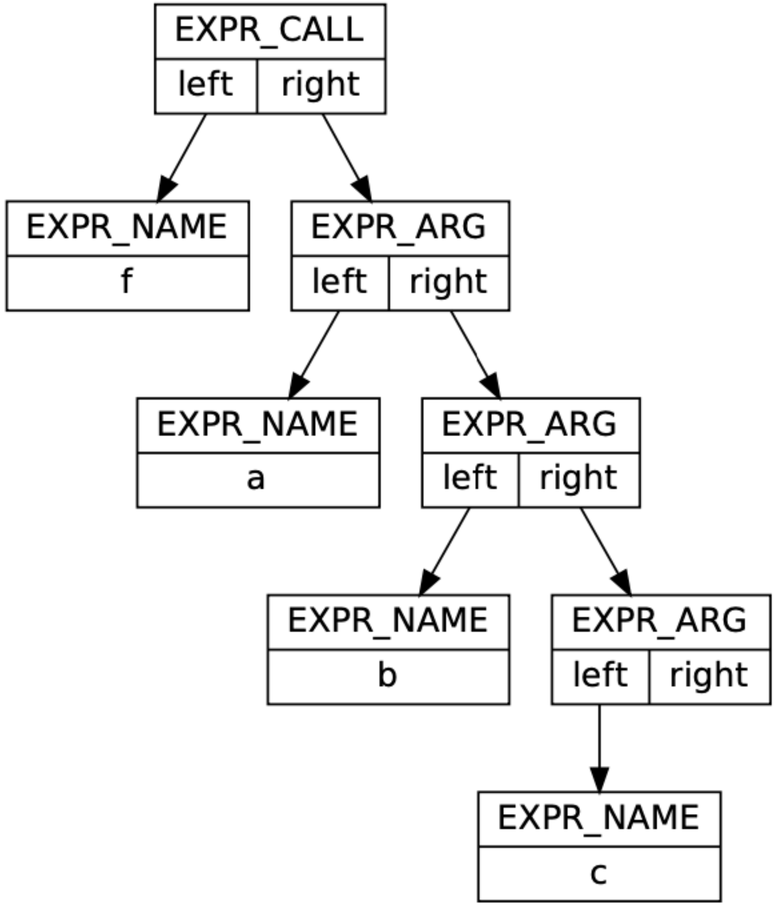
- The AST nodes must hold sufficient information to recall the essential elements of the program fragment they represent.
- It should be possible to un-parse an AST into a form that is sufficiently similar to the program represented by the AST.



AST Design

- Eventually will need to think about the software engineering and design of the AST nodes

```
f(a,b,c)
```

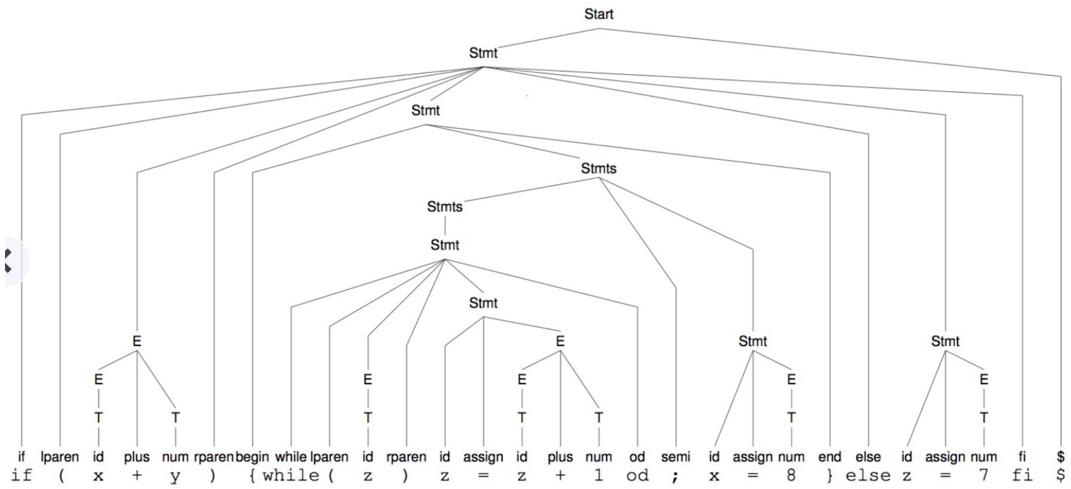


Given a source programming lang L, the development of a grammar and the design of an appropriate AST structure typically proceeds as follows:

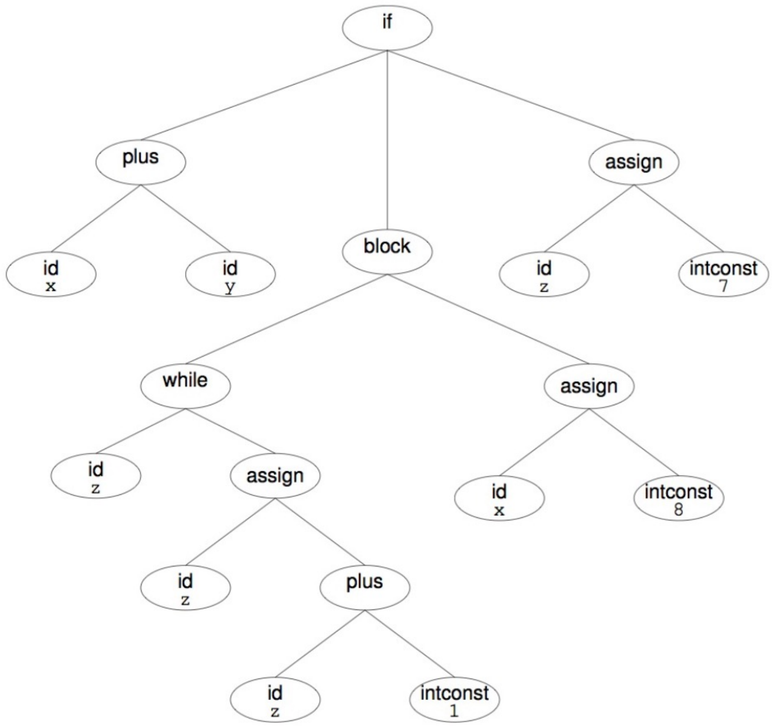
1. An *unambiguous* grammar for L is devised.
   - left recursion removed, left factored
2. An AST of L is devised.
   - omit useless symbols such as `,` and `;`
3. Semantic actions are placed in the grammar to construct the AST.
   - (more on this later)
4. Phases of the compiler are designed.
   - Each phase may place new requirements on the AST in terms of its structure and contents.

Another Example

CONCRETE PARSE TREE



ABSTRACT SYNTAX TREE



Position Information

A compiler that uses AST data structures need *not* do all the parsing and semantic analysis in *one pass*.

- Lexer reaches `<EOF>` before semantic analysis even begins, so if semantic error is detected during traversal of parse tree, the current position of the lexer will not be useful.
- AST data structures must be sprinkled with `pos` fields.
- Lexer passes source file positions of beginning and end of each token to the parser.
- Augment constructor calls to take a `pos` argument.

```
public class AssignStm extends Stm {
    String id;
    Exp exp;
    AssignStm(String i, Exp e) { id = i; exp = e; }
    int line, col;
    AssignStm(String i, Exp e, int l, int c) {
        id = i; exp = e; line = l; col = c;
    }
}
```

Semantic Actions

Semantic Actions

Possible to include semantic actions in most grammars (such as JavaCC)

Why is this useful?

- JavaCC uses recursive descent to (hopefully) make a successful derivation of some input token stream
- It *does not* automatically produce any of the desired parse trees

(Some parsing tables can automatically produce concrete syntax trees.)

- (e.g. SableCC, JJTree extension to JavaCC, JTB extension to JavaCC, …)

MiniJava Abstract Syntax

How to build syntax tree for
`x = y.m(1, 4+5);`

Program (`MainClass m, ClassDeclList cl`)
MainClass (`Identifier i1, Identifier i2, Statement s`)

`abstract class ClassDecl`
ClassDeclSimple (`Identifier i, VarDeclList vl, MethodDeclList ml`)
ClassDeclExtends (`Identifier i, Identifier j,`
`VarDeclList vl, MethodDeclList ml`) *see Ch.14*

VarDecl (`Type t, Identifier i`)
MethodDecl (`Type t, Identifier i, FormalList fl, VarDeclList vl,`
`StatementList sl, Exp e`)
Formal (`Type t, Identifier i`)

`abstract class Type`
IntArrayType () **BooleanType** () **IntegerType** () **IdentifierType** (`String s`)

`abstract class Statement`
Block (`StatementList sl`)
If (`Exp e, Statement s1, Statement s2`)
While (`Exp e, Statement s`)
Print (`Exp e`)
Assign (`Identifier i, Exp e`)
ArrayAssign (`Identifier i, Exp e1, Exp e2`)

`abstract class Exp`
And (`Exp e1, Exp e2`)
LessThan (`Exp e1, Exp e2`)
Plus (`Exp e1, Exp e2`) **Minus** (`Exp e1, Exp e2`) **Times** (`Exp e1, Exp e2`)
ArrayLookup (`Exp e1, Exp e2`)
ArrayLength (`Exp e`)
Call (`Exp e, Identifier i, ExpList el`)
IntegerLiteral (`int i`)
True ()
False ()
IdentifierExp (`String s`)
This ()
NewArray (`Exp e`)
NewObject (`Identifier i`)
Not (`Exp e`)

Identifier (`String s`)

list classes
ClassDeclList () **ExpList** () **FormalList** () **MethodDeclList** () **StatementList** () **VarDeclList** ()

MiniJava Grammar

Program	→	*MainClass ClassDecl**
MainClass	→	**class** *id* { **public static void main** (**String** [] *id*)
		{ *Statement* } }
ClassDecl	→	**class** *id* { *VarDecl** *MethodDecl** }
	→	**class** *id* **extends** *id* { *VarDecl** *MethodDecl** }
VarDecl	→	*Type id* ;
MethodDecl	→	**public** *Type id* (*FormalList*)
		{ *VarDecl** *Statement** **return** *Exp* ; }
FormalList	→	*Type id FormalRest**
	→	
FormalRest	→	, *Type id*
Type	→	**int** []
	→	**boolean**
	→	**int**
	→	*id*
Statement	→	{ *Statement** }
	→	**if** (*Exp*) *Statement* **else** *Statement*
	→	**while** (*Exp*) *Statement*
	→	**System.out.println** (*Exp*) ;
	→	*id* = *Exp* ;
	→	*id* [*Exp*] = *Exp* ;
Exp	→	*Exp op Exp*
	→	*Exp* [*Exp*]
	→	*Exp* . **length**
	→	*Exp* . *id* (*ExpList*)
	→	*INTEGER_LITERAL*
	→	**true**
	→	**false**
	→	*id*
	→	**this**
	→	**new int** [*Exp*]
	→	**new** *id* ( )
	→	! *Exp*
	→	(*Exp*)
ExpList	→	*Exp ExpRest**
	→	
ExpRest	→	, *Exp*

- MiniJava grammar *does not* specify the precedence of operators!
- Need to make sure that multiplication occurs deeper in a parse tree than addition, for an expression like 3 + 2 * 5

Solutions:

1. Rewrite grammar to capture Java precedence rules
2. Capture precedence in <u>semantic actions</u>

#1 Disambiguating Grammars for Operator Precedence

- The removal of left recursion gets tricky (and ugly!)
  - … as you may have noticed in Prog Assignment #3

```
Exp -> Exp op Exp
```

Original Grammar:

```
E -> E + E | E * E | num | id | (E)
```

Rewritten for proper order precedence:

```
E -> E + T | T
T -> T * F | F
F -> num | id | (E)
```

Rewritten to eliminate left recursion:

```
E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> num | id | (E)
```

Before: "goal" was only to determine if valid parse is possible

Now: also interested in <u>semantics</u>

- What is the result (or interpretation) of some expression?
- Want multiplication deeper in the parse tree than addition! (as long as there are no parens)

A hand-crafted program (for determining successful parse) using recursive descent, in JavaCC-like syntax:

```
E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> num | id | (E)
```

Prompt: How can this code be modified with semantic actions for interpreting?

```
void T() :
{}
{
    F() TPrime()
}

void TPrime() :
{}
{
    [ <TIMES> F TPrime() ]
}

void F():
{}
{
    <INT>
    |
    <IDENTIFIER>
    |
    <LPAREN> E() <RPAREN>
}
```

An Approach:
Have Methods Return Values

- Seems straight forward:
  - E -> E+E (E returns value of E+E)
- ... but this was rewritten to ...
  - E -> E+T (E returns value of E, added to what T returns)
- ... but this was rewritten to ...
  - E -> TE'
  - E' -> +TE' | ε
    - E and E' should return ints...
    - ...but the calculation of them doesn't seem as trivial anymore.

#2 Semantic Actions Can Be Specified in JavaCC

- … for building a simple mathematical interpreter, such as what's presented in the JavaCC Lookahead tutorial
- … or for building an abstract syntax tree
- When an expression is parsed, the result of the expression is also calculated along the way!

General syntax of JavaCC productions:

```
returntype ProductionName() :
{
    // local variables that can be declared
}
{
    // RHS of production, augmented with semantic actions {    }
}
```

- Local variables can be primitives or objects.

JavaCC Version

```
void Start() :
{ int i; }
{
    i=Exp() <EOF> { System.out.println(i); }
}
int Exp() :
{ int a,i; }
{
    a=Term()
    ( "+" i=Term() { a=a+i; } | "-" i=Term() { a=a-i; } )*
    { return a; }
}
int Term() :
{ int a,i; }
{
    a=Factor()
    ( "*" i=Factor() { a=a*i; } | "/" i=Factor() { a=a/i; } )*
    { return a; }
}
int Factor() :
{ Token t; int i; }
{
    t=<IDENTIFIER>          { return lookup(t.image); }
    | t=<INTEGER_LITERAL> { return Integer.parseInt(t.image); }
    | "(" i=Exp() ")"       { return i; }
}
```

More Precedence of Operators Also Needs to Be Captured

From MiniJava Grammar:

```
Exp -> Exp op Exp | Exp[Exp] | Exp.length |
          Exp.id(ExpList) | INT_LITERAL | true | false |
          id | this | new int [Exp] | new Id() | !Exp |
          (Exp)
```

What is the order of operator precedence in Java?

Rewriting Grammar According to Precedence Hierarchy

```
Exp -> AndExp
AndExp -> EqualityExp (&& EqualityExp)*
EqualityExp -> RelationalExp ((== | !=) RelationalExp)*
RelationalExp -> AdditiveExp ((< | <= | > | >=) AdditiveExp)*
AdditiveExp -> MultiplicativeExp ((+ | -) MultiplicativeExp)*
MultiplicativeExp -> UnaryExp ("*" UnaryExp)*
UnaryExp -> "!"UnaryExp | Primary
Primary -> ( INT_LITERAL | true | false | id | this |
            "("Exp")" | new int "[" Exp "]" |
            new id "("")" )
            ([Exp]|.length|.id(ExpList))*
```

- Kleene closure on last line because of chaining.
- Notice how semantic checks are still postponed.
- Grammar now more complicated (concrete syntax tree would be too).
- Semantic actions that build AST simplify this complexity back to abstract syntax; so, not as bad as it appears.

Visitor Design Pattern

Visitor Design Pattern

Motivation:

- Given an AST, what is best way to traverse it for some task:
  - e.g. interpreting it (Prog #1)
  - semantic analysis (symbol table creation + type checking)
  - code translation
  - optimization
  - etc.
- We will be doing *many* different tasks on the trees...

Next slides: **thought process towards a good SE design**

From MiniJava abstract syntax:

```
Plus(Exp e1, Exp e2)
```

Idea #1: Separate code for tasks

Functional Approach

- *Ideas:*
  - extensive use of `instanceof` operator
  - separate files for each task
    - `Interpreter.java`
    - `Typechecker.java`
    - `Optimizer.java`
    - …

- Every file has `eval` method.
- Implementation is task-specific
- `eval` method in `Interpreter.java`

```
int eval (Exp e) {
    if (e instanceof PlusExp)
        return eval(((PlusExp) e).e1) + eval(((PlusExp)e).e2);
    else if (e instanceof MinusExp)
        ...
}
```

Idea #1:
Separate code for tasks

```
int eval (Exp e) {
    if (e instanceof PlusExp)
        return eval(((PlusExp) e).e1) + eval(((PlusExp)e).e2);
    else if (e instanceof MinusExp)
        ...
}
```

If wanted to write a type checker (we will) … we would create `TypeChecker.java`

ADVANTAGES

- No need to recompile everything for new task.

DISADVANTAGES

- Lots to type casting, not OO!
- If syntax changes (e.g. adding modulus to P/L), need to modify *all* tasks (`interp.java`, `typechecker.java`) and recompile everything

Idea #1:
Separate code for tasks

```
// Interpreter.java
int eval (Exp e) {
    if (e instanceof PlusExp)
        return eval(((PlusExp) e).e1) + eval(((PlusExp)e).e2);
    else if (e instanceof MinusExp)
        ...
}
```

How to invoke in `main`?

```
Exp e = ...;
println(Interpreter.eval(e));    // functional call
```

Alternative Idea #2:
Separate code for Abstract Syntax Classes

<u>OO Approach</u>

- *Ideas:*
  - Each class has a method for each task

- `PlusExp.java`

```
// PlusExp.java
public class PlusExp extends Exp {
    Exp e1, e2;
    PlusExp(Exp a1, Exp a2) { e1 = a1; e2 = a2; }
    int eval() {
        return e1.eval() + e2.eval();
    }
    void typeCheck() { ... }
    String generateCode() { ... }
}
```

Alternative Idea #2:
Separate code for Abstract Syntax Classes

```
public class PlusExp extends Exp {
    Exp e1, e2;
    PlusExp(Exp a1, Exp a2) { e1 = a1; e2 = a2; }
    int eval() {
        return e1.eval() + e2.eval();
    }
    void typeCheck() { ... }
    String generateCode() { ... }
}
```

ADVANTAGES

- "easy" to change syntax (e.g. include modulus): just make a new class and define method for each task

DISADVANTAGES

- For a new *task*, need to modify every class.
- Mixing pretty-printing code w/ flow analysis code, etc.
- Difficult to maintain?

Alternative Idea #2:
Separate code for Abstract Syntax Classes

```
// PlusExp.java
public class PlusExp extends Exp {
    Exp e1, e2;
    PlusExp(Exp a1, Exp a2) { e1 = a1; e2 = a2; }
    int eval() {
        return e1.eval() + e2.eval();
    }
    void typeCheck() { ... }
    String generateCode() { ... }
}
```

How to invoke in `main`?

```
Exp e = ...;
println(e.eval());          // OO call
```

BEST Idea #3:
Visitor Design Pattern

Big NEW IDEA:

- Try to combine the best of both approaches.
- Want to separate the node classes from tasks/phases (interpreter, type checker, optimizer, etc.)
  - Found in OO approach
- Want to keep code for a task together
  - Found in Functional approach
- Want to avoid frequent typecasts or use of `instanceof`
  - Present in Functional approach

What is a Design Pattern?

- Could devote an entire SE course to design patterns
  - there are lots of design patterns
- *Guiding Principle:* patterns are generally meant to save time and effort in developing and maintaining software
  - patterns simplify software construction
  - initially, may seem to be complicating it

Recall OO Approach – Node types *combined* with Tasks – BAD!

```
abstract class Exp {
        public abstract int eval();
}
public class PlusExp extends Exp {
        private Exp e1,e2;
        public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
        public int eval() {
                return e1.eval()+e2.eval();
        }
}
public class MinusExp extends Exp {
        private Exp e1,e2;
        public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
        public int eval() {
                return e1.eval()-e2.eval();
        }
}
...
public class IntegerLiteral extends Exp {
        private String f0;
        public IntegerLiteral(String n0) { f0 = n0; }
        public int eval() {
                return Integer.parseInt(f0);
        }
}
```

Visitor Approach – Node types *separate* from Tasks – GOOD!

From MiniJava abstract syntax:

Plus (Exp e1, Exp e2)

```
// Visitor.java
public interface Visitor {
        public int visit(PlusExp n);
        public int visit(MinusExp n);
        public int visit(TimesExp n);
        public int visit(DivideExp n);
        public int visit(Identifier n);
        public int visit(IntegerLiteral n);
}
```

```
// Interpreter.java
public class Interpreter implements Visitor {
        public int visit(PlusExp n) { return n.e1.accept(this)+n.e2.accept(this); }
        public int visit(MinusExp n) { return n.e1.accept(this)-n.e2.accept(this); }
        public int visit(TimesExp n) { return n.e1.accept(this)*n.e2.accept(this); }
        public int visit(DivideExp n) { return n.e1.accept(this)/n.e2.accept(this); }
        public int visit(Identifier n) { return lookup(n.f0); }
        public int visit(IntegerLiteral n) { return Integer.parseInt(n.f0); }
}
```

Visitor Approach – Node types *separate* from Tasks – GOOD!

Exp.java
PlusExp.java
MinusExp.java
TimesExp.java
DivideExp.java
Identifier.java
IntegerLiteral.java

```
public abstract class Exp {
        public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
        public Exp e1,e2;
        public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
        public int accept(Visitor v) { return v.visit(this); }
}
public class MinusExp extends Exp {
        public Exp e1,e2;
        public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
        public int accept(Visitor v) { return v.visit(this); }
}
public class TimesExp extends Exp { … }
public class DivideExp extends Exp { … }
public class Identifier extends Exp {
        public String f0;
        public Identifier(String n0) { f0 = n0; }
        public int accept(Visitor v) { return v.visit(this); }
}
public class IntegerLiteral extends Exp {
        public String f0;
        public IntegerLiteral(String n0) { f0 = n0; }
        public int accept() { return v.visit(this); }
}
```

BEST Idea #3: Visitor Design Pattern

How to invoke in `main`?

```
Exp e = ...;
e.accept(new Interpreter());
```

BEST Idea #3:
Visitor Design Pattern

Maintain two class hierarchies:

1. "elements" being operated on (the nodes, the AST classes)
2. the "visitors" that define operators on the "elements" (the interpreter, type checker, code generator, etc.)

ELEMENTS/AST CLASSES:

- Each element/node has an `accept()` method that takes a visitor as an argument.
- The method calls a `visit()` method of the visitor, passing itself as an argument.

VISITORS/TASKS:

- visitor interface (Visitor.java)
- list types of "elements"/AST node classes
- ensures we implement task on each one
- class for each task (e.g. Interpreter.java, TypeChecking.java); *implements* the interface
- overloaded `visit()` method for each element
- the `visit()` method performs the interesting parts of the tasks

BEST Idea #3: Visitor Design Pattern

Visitor pattern implements **double dynamic dispatch**:

1. dynamic (polymorphism) type of the node element
2. dynamic type of the visitor (task - e.g. Interpreter)

```
Exp e = ...;
e.accept(new Interpreter());
```