COMP 520 - Compilers

Lecture 8 (Tue Feb 8, 2022)

Abstract Syntax Trees

- Reading for Feb 10
  - Finish Chapter 4
    - section 4.6 – Syntactic Analysis in Triangle (pp 124 – 128)

- Materials on our website
  - simpleAST
    - illustrates AST construction and evaluation for miniArith
  - AbstractSyntaxTrees.zip
    - `miniJava.AbstractSyntaxTrees` package needed to construct AST's for miniJava in PA2

Topics

- From concrete syntax trees to abstract syntax trees
  - AST “grammar”
  - AST representation choices
  - AST construction
- AST traversal
  - generalize using Visitor
- miniJava AST classes (available on web)
  - use these to construct the AST in PA2
  - you will need to provide some implementation of sourcePosition

Concrete syntax tree

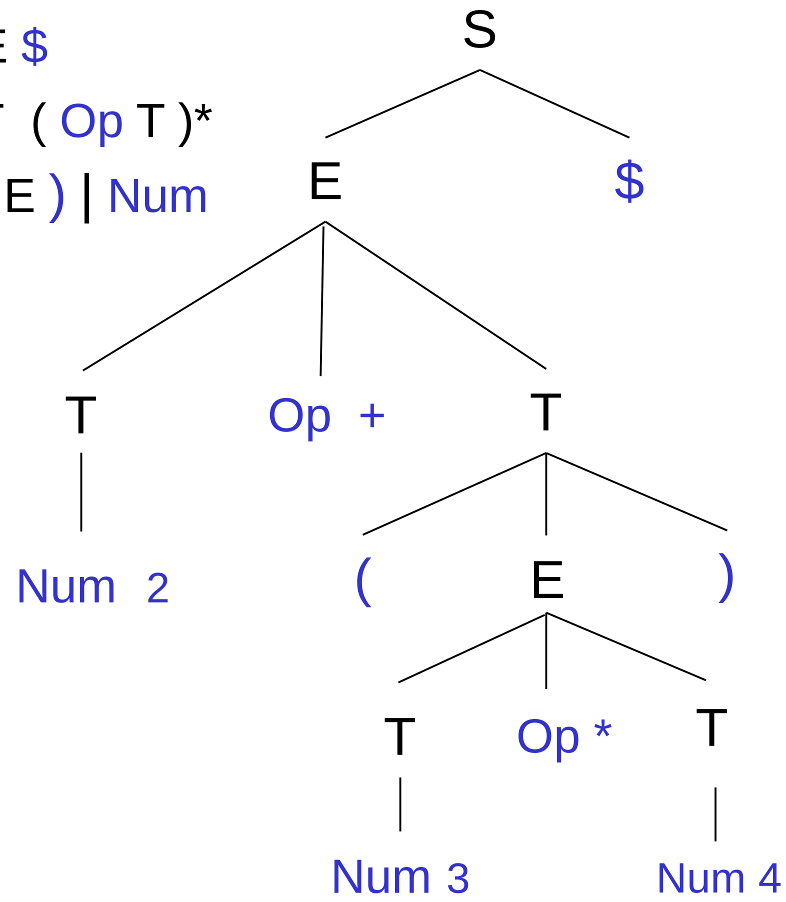
- Concrete syntax is described by an (EBNF) CFG grammar
  - ex: simple arithmetic expressions

S ::= E $
E ::= E Op T | T
T ::= (E) | Num

→

S ::= E $
E ::= T (Op T)*
T ::= (E) | Num

- Concrete syntax tree for
  2 + (3 * 4) $

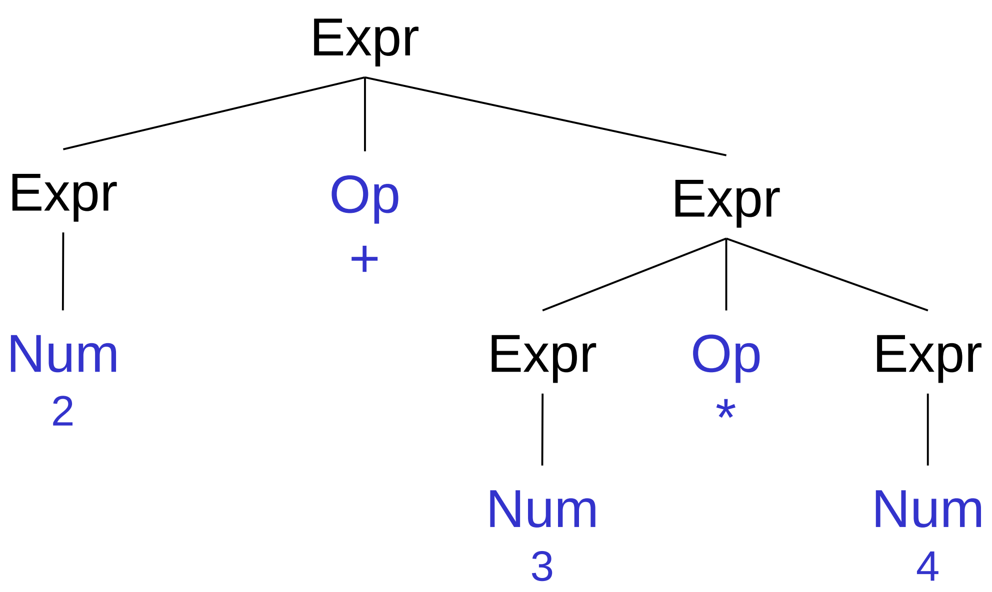


Abstract syntax

- Abstract syntax can be described by a simple CFG grammar
  - ex: simple arithmetic expressions

    Expr ::= Expr Op Expr (BinExpr)
    | Num (NumExpr)

- Abstract syntax tree for 2 + (3 * 4)



AST representation in Java (Strategy 1)

- AST classes define an (n-ary) tree with tagged nodes
  - simple to define, more complex to construct and traverse
  - minimal benefit from Java type checker

```
enum NodeType {UNARYEXPR, BINARYEXPR, NUM}
public abstract class AST {
   public NodeType n;
}

public class Node extends AST {
   public AST[] children;
   public Node(NodeType n, AST[] children) { … }
}

public class Leaf extends AST {
   public String spelling;
   public Leaf(NodeType n, String spelling) { … }
}
```

AST representation in Java (Strategy 2)

- AST classes closely follow structure of AST grammar
    - more complex to define AST classes, but simpler to construct AST
    - we gain considerable benefit from the Java type checker
    - more rigorously supports AST traversal using a Visitor interface

- Rules
    - create abstract class AST
    - for each nonterminal in AST grammar
        - construct an abstract subclass of AST
    - for each choice within a rule
        - construct a concrete subclass of the LHS nonterminal

- <u>We will follow this strategy</u>

AST representation in Java (Strategy 2)

- Example

Expr ::= Expr Oper Expr	(BinExpr)
| Num	(NumExpr)

```
abstract public class AST {}

abstract public class Expr extends AST {}

public class BinExpr extends Expr {
   public Terminal oper;
   public Expr left, right;
   public BinExpr(Expr left, Terminal oper, Expr right) { … }
}

public class NumExpr extends Expr {
   public Terminal num;
   public NumExpr(Terminal num) { … }
}

public class Terminal extends AST {
   public String spelling;
   public Terminal(String spelling) { … };
}
```

Building an AST during a concrete syntax parse

- concrete syntax for arithmetic expression grammar

  E ::= T | E *op* T
  T ::= (E) | *num*

- transformed and augmented

  S ::= E $
  E ::= T (*op* T)*
  T ::= (E) | *num*

- abstract syntax

  Expr ::= Expr Op Expr (BinExpr)
  | Num (NumExpr)

- how to build AST?
  - modify parse procedures to return pieces of AST

```
Expr parseS() {
   Expr e = parseE();
   accept(Token.eot);
   return e
}

Expr parseE() {
   Expr e1 = parseT();
   while (curToken.kind == Token.op) {
       Terminal op = new Terminal(curToken);
       acceptIt();
       Expr e2 = parseT();
       e1 = new BinExpr(e1,op,e2);
   }
   return e1;
}

Expr parseT() {
   case  (curToken.kind) {
     Token.LPAREN:
        acceptIt();
        Expr e1 = parseE();
        accept(Token.RPAREN);
        return e1;

     Token.num:
        NumExpr e2 = new NumExpr(curToken);
        acceptIt();
        return e2;
   }
}
```

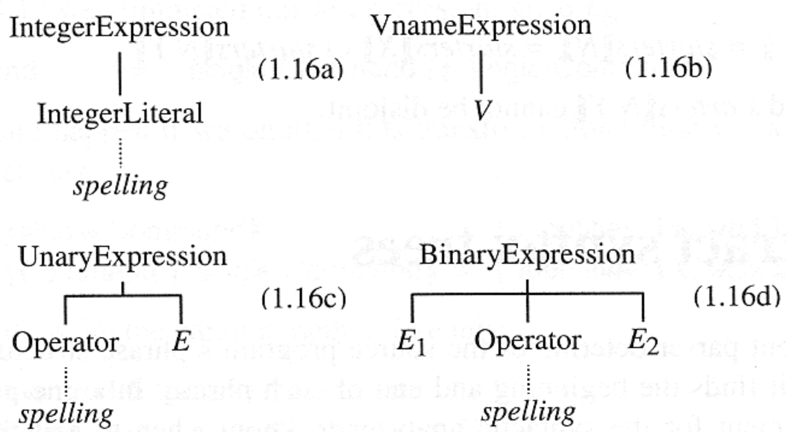
mini-Triangle – Expression ASTs

Concrete Syntax (EBNF)

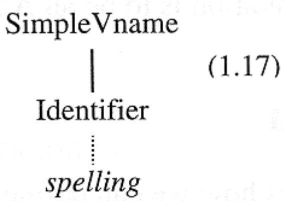
Expression	::=	primary-Expression	(1.4a)
	|	Expression Operator primary-Expression	(1.4b)
primary-Expression	::=	Integer-Literal	(1.5a)
	|	V-name	(1.5b)
	|	Operator primary-Expression	(1.5c)
	|	(Expression)	(1.5d)
V-name	::=	Identifier	(1.6)
Operator	::=	+ | - | * | / | < | > | = | \	(1.10a–h)
Identifier	::=	Letter | Identifier Letter | Identifier Digit	(1.11a–c)
Integer-Literal	::=	Digit | Integer-Literal Digit	(1.12a–b)

Tokens (Operator, Identifier, Integer-Literal)

- Expression ASTs (*E*):



- V-name ASTs (*V*):



Abstract Syntax (ASTs)

mini-Triangle – Command ASTs

Concrete Syntax

Program	::=	single-Command	(1.1)
Command	::=	single-Command	(1.2a)
	|	Command **;** single-Command	(1.2b)
single-Command	::=	V-name **:=** Expression	(1.3a)
	|	Identifier **(** Expression **)**	(1.3b)
	|	**if** Expression **then** single-Command **else** single-Command	(1.3c)
	|	**while** Expression **do** single-Command	(1.3d)
	|	**let** Declaration **in** single-Command	(1.3e)
	|	**begin** Command **end**	(1.3f)

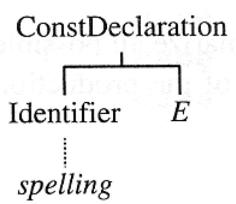
ASTs

mini-Triangle – Declaration ASTs

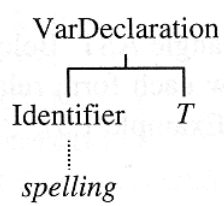
Concrete Syntax

Declaration	::=	single-Declaration	(1.7a)
	|	Declaration **;** single-Declaration	(1.7b)
single-Declaration	::=	**const** Identifier **~** Expression	(1.8a)
	|	**var** Identifier **:** Type-denoter	(1.8b)
Type-denoter	::=	Identifier	(1.9)

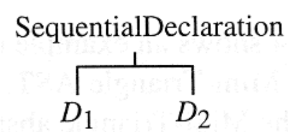
- Declaration ASTs (*D*):



(1.18a)

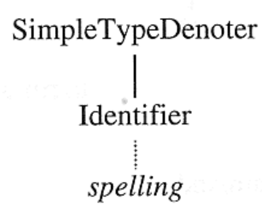


(1.18b)



(1.18c)

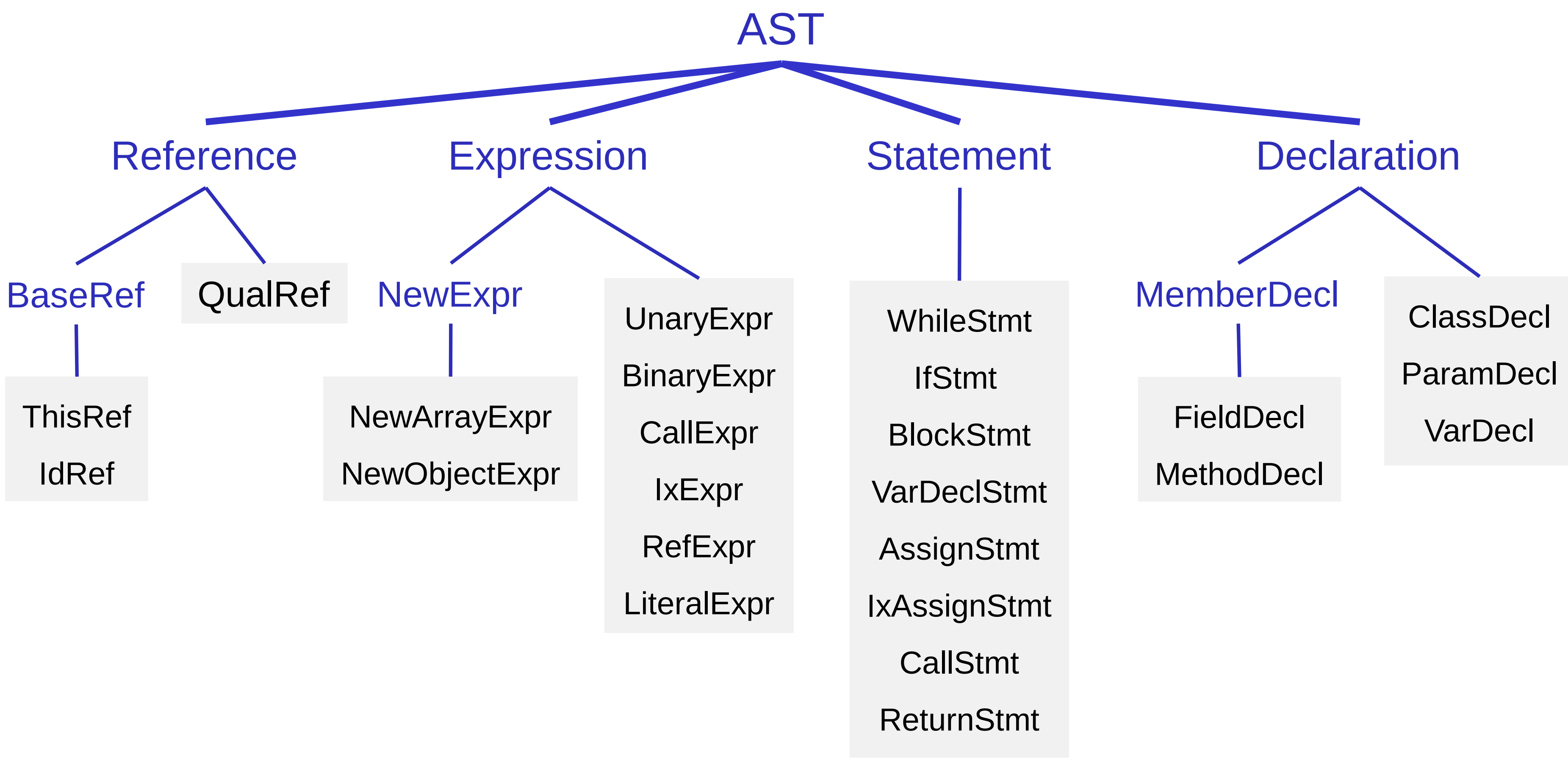
- Type-denoter ASTs (*T*):



(1.19)

ASTs

The miniJava AST classes



Additional constructors for lists of class instances, with an iterator for traversal of the list:
ClassDeclList, FieldDeclList, MethodDeclList, ParamDeclList, StatementList, ExprList

miniJava AST types

- AST
  - Declaration
    - ClassDecl
    - LocalDecl
      - ParameterDecl
      - VarDecl
    - MemberDecl
      - FieldDecl
      - MethodDecl
  - Expression
    - BinaryExpr
    - CallExpr
    - IxExpr
    - LiteralExpr
    - NewExpr
      - NewArrayExpr
      - NewObjectExpr
    - RefExpr
    - UnaryExpr
  - Package
  - Reference
    - BaseRef
      - IdRef
      - ThisRef
    - QualRef
  - Statement
    - AssignStmt
    - BlockStmt
    - CallStmt
    - IfStmt
    - IxAssignStmt
    - ReturnStmt
    - VarDeclStmt
    - WhileStmt
  - Terminal
    - BooleanLiteral
    - Identifier
    - IntLiteral
    - Operator
  - TypeDenoter
    - ArrayType
    - BaseType
    - ClassType
- ASTDisplay

Implementing a general AST traversal

- Strategy 1
  - add methods to each AST class for each kind of traversal
    - Example
      - display methods for AST display
      - eval methods for AST evaluation
  - drawback
    - Classes become very large when traversals get complex
      - Contextual analysis
      - Code generation
    - Code for each kind of traversal is scattered over many classes

Implementing a general AST traversal

- Strategy 2
  - use a *visitor* pattern
    - A visitor Interface
      - specifies a visitX method for each concrete class X in AST
    - a visit method in each AST class
      - public Object visit(Visitor v, Object arg)
  - each separate traversal implements the Visitor interface
    - sample Visitor instances
      - AST display
      - Identification
      - Type checking
      - Code generation
    - Code for each type of visitor is collected in one class
      - but a bit cumbersome to write out

Simple set of AST classes

```
abstract class AST {}

abstract class Expr extends AST {}

class BinExpr extends Expr {
    public Token oper;
    public Expr left;
    public Expr right;

    public BinExpr(Expr left, Token oper, Expr right) { . . . }
}


class NumExpr extends Expr {
    public int val;

    public NumExpr(int val) { . . . }
}
```

The Visitor interface

- Visitor interface requires a visitX method for every (non-abstract) AST class X

```
public interface Visitor {
    visitBinaryExpr(BinaryExpr e);
    visitNumExpr(NumExpr e);
}
```

- Each AST class is augmented with a single visit method

```
class NumExpr extends Expr {
    public int val;
    public NumExpr(int v) { . . . }

    public void visit(Visitor v) { v.visitNumExpr(this); }
}
```

- All AST traversals use the same "visit" method in each node type
  - the visit method "connects" a specific visitor v to **this** specific node

Inorder traversal of the AST

- The InorderWalk AST traversal implements the Visitor interface

```
public class InorderWalk implements Visitor {
    public void visitBinExpr(BinExpr be) {
            be.left.visit(this);
            System.out.println(be.oper.spelling);
            be.right.visit(this);
    }

    public void visitNumExpr(NumExpr ne) {
            System.out.println(ne.val);
    }

    // print nodes of AST inorder
    public void walk(AST a) {
            a.visit(this);
    }
}
```

Adding arguments to the traversal (Book method)

- methods implemented by a Visitor have an Object arg and Object result

```
public interface Visitor {
    Object visitBinExpr(BinExpr e, Object arg);
    Object visitNumExr(NumExpr e, Object arg);
}
```

- AST class visit method has an Object arg and Object result

```
class NumExpr extends Expr {
    public int val;

    public NumExpr(int val) { . . . }

    public Object visit(Visitor v, Object arg) {
        return v.visitNumExpr(this, arg);
    }
}
```

Example use of the Visitor

- Individual traversals implement Visitor with custom logic for each node type

```
class Checker implements Visitor {

   public Object visitAssignStmt(AssignStmt s, Object arg) {
        Type t1 = (Type) s.ref.visit(this, arg);
        Type t2 = (Type) s.exp.visit(this, arg);
        return (t1.equals(t2)? t1 : Type.ERROR)
   }
}
```

- Good solution?
  - (+) appropriately OO
  - (+) compiler insures visitor defined for all node types
  - (+) specific definitions gathered together in a single class
  - (-) Object parameters and results will require a lot of explicit casting

Adding arguments to the traversal (parameterized types)

```
public interface Visitor<ArgType,ResultType> {
   public ResultType visitBinExpr(BinExpr expr, ArgType arg);
   public ResultType visitNumExpr(NumExpr expr, ArgType arg);
}
```

```
class NumExpr extends Expr {
    public int val;
    public NumExpr(int val) { . . . }

    public <ArgType,ResType> ResType
        visit(Visitor<ArgType,ResType> v, ArgType arg) {
            return v.visitNumExpr(this, arg);
    }
}
```

Example use of the Visitor

- Individual traversals implement Visitor with custom logic for each node type

```
class Checker implements Visitor<Type,Type> {

   ...

   public Type visitAssignStmt(AssignStmt s, Type arg) {
        Type t1 = s.ref.visit(this, arg);
        Type t2 = s.exp.visit(this, arg);
         if (! t1.equals(t2))
               typeError("incompatible types in assignment",s);
        return Types.StmtType;
}
```

- Good solution?
  - Improves type checking in the visitors
  - Improves readability

PA2 Submission Details

1. Your compiler project must follow these requirements with respect to the package names and structure

2. Your `Compiler.java` mainclass must have the functionality called out on the following page, but you may vary your implementation.

```java
package miniJava;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

import miniJava.SyntacticAnalyzer.Parser;
import miniJava.SyntacticAnalyzer.Scanner;
import miniJava.AbstractSyntaxTrees.*;

/**
 * Recognize whether input file named in args[0] contains a syntactically valid
 * miniJava program, and, if valid, display corresponding AST.
 *
 */
public class Compiler {

    public static void main(String[] args) {
        InputStream inputStream = null;
        try {
            inputStream = new FileInputStream(args[0]);
            }
        catch (FileNotFoundException e) {
                System.out.println("Input file " + args[0] + " not found");
                System.exit(3);
            }

        ErrorReporter errorReporter = new ErrorReporter();
        Scanner scanner = new Scanner(inputStream, errorReporter);
        Parser parser = new Parser(scanner, errorReporter);

        System.out.println("Syntactic analysis ... ");
        AST ast = parser.parseProgram();
        System.out.print("Syntactic analysis complete:  ");
        if (errorReporter.hasErrors()) {
            System.out.println("Invalid miniJava program");
            System.exit(4);
        }
        else {
            System.out.println("Valid miniJava program:");
            new ASTDisplay().showTree(ast);
            System.exit(0);
        }
    }
}
```

use this termination code if unable to open input file (arrow to `System.exit(3);`)

create AST (arrow to `AST ast = parser.parseProgram();`)

output AST using ASTDisplay(), for accepted miniJava programs (arrow to `new ASTDisplay().showTree(ast);`)

use these termination codes for invalid/valid miniJava programs, respectively (arrows to `System.exit(4);` and `System.exit(0);`)