COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering

The University of New South Wales

Sydney, NSW 2052, Australia

`http://www.cse.unsw.edu.au/~cs3131`

`http://www.cse.unsw.edu.au/~cs9102`

Week (2nd Lecture): Abstract Syntax Trees (ASTs)

1. Assignment 3
2. Why a physical tree?
3. Parse trees v.s. syntax trees
4. Design of AST classes
5. Use of AST classes
6. Attribute grammar
7. Implementation details specific to Assignment 3

Assignment 3

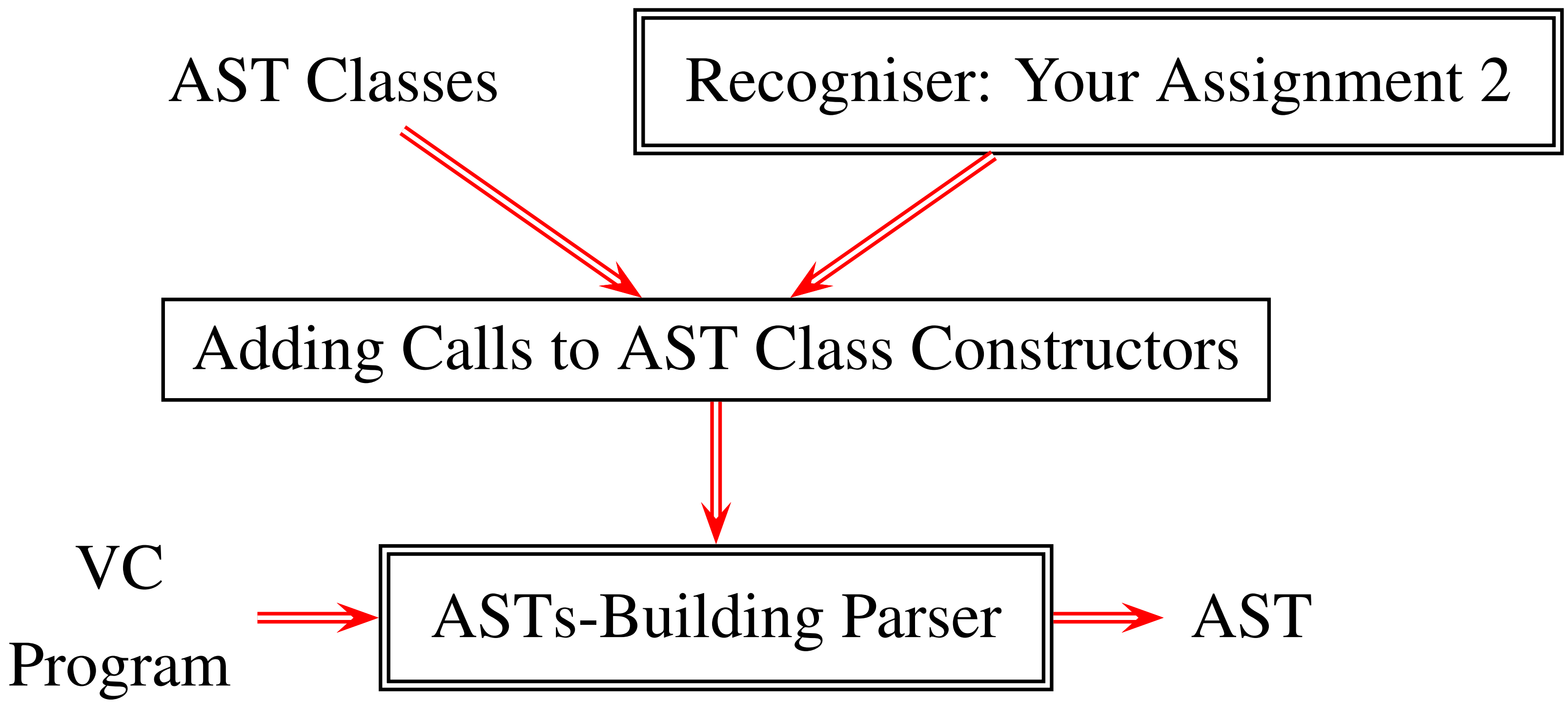
- Packages:

PACKAGE	FUNCTIONALITY
VC.ASTs	AST classes for creating tree nodes
VC.Parser	Parser
VC.TreeDrawer	Draws an AST on the screen
VC.TreePrinter	Print an ASCII AST
VC.UnParser	Traverses an AST to print a VC program

- The VC Compiler options:

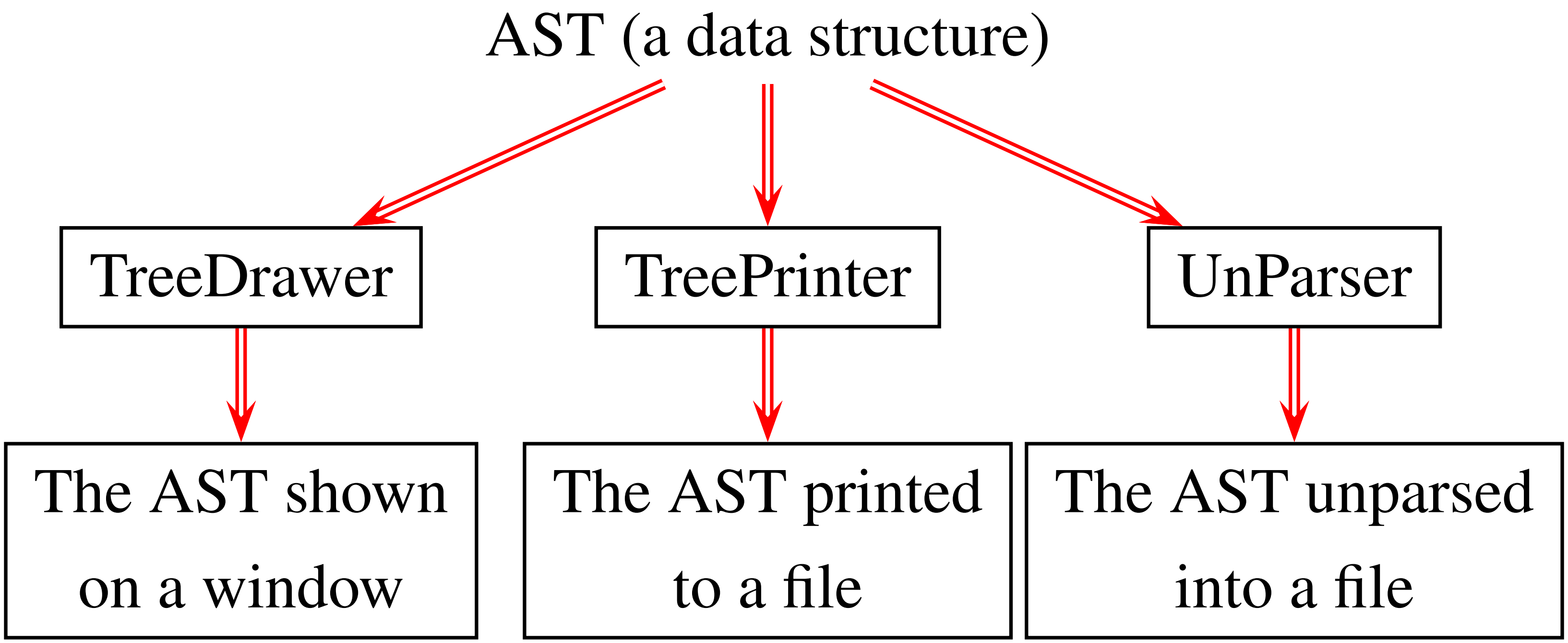
```
[daniel 3:00pm] java VC.vc
Usage: java VC.vc [-options] filename
      -ast              display the AST (without SourcePosition)
      -astp             display the AST (with SourcePosition)
      -t file           print the AST into <file>
      -u file           unparse the AST into <file>
```

Constructing ASTs in Assignment 3



Only constructors in AST classes are used in Assignment 3.

Programming Environment for Assignment 3



- All three packages coded using the Visitor Design Pattern

  http://www.newthinktank.com/2012/11/visitor-design-pattern-tutorial/
  http://www.zzrose.com/tech/pmr_sweDesignPatternVisitor.html

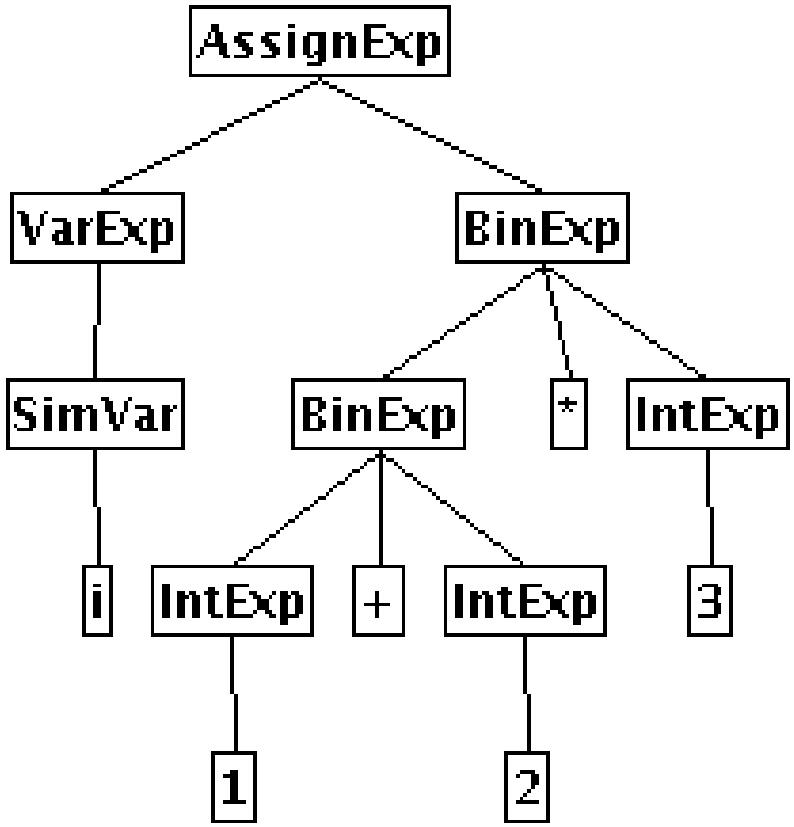
- The pattern to be used in Assignments 4 & 5
- Can be understood by examining the codes
- Tree-walkers like these can be generated automatically using attribute grammars once supporting codes are given

Example

- Program (ex.vc):

```
i = (1+2)*3;
```

- The AST (using option ”-ast”):

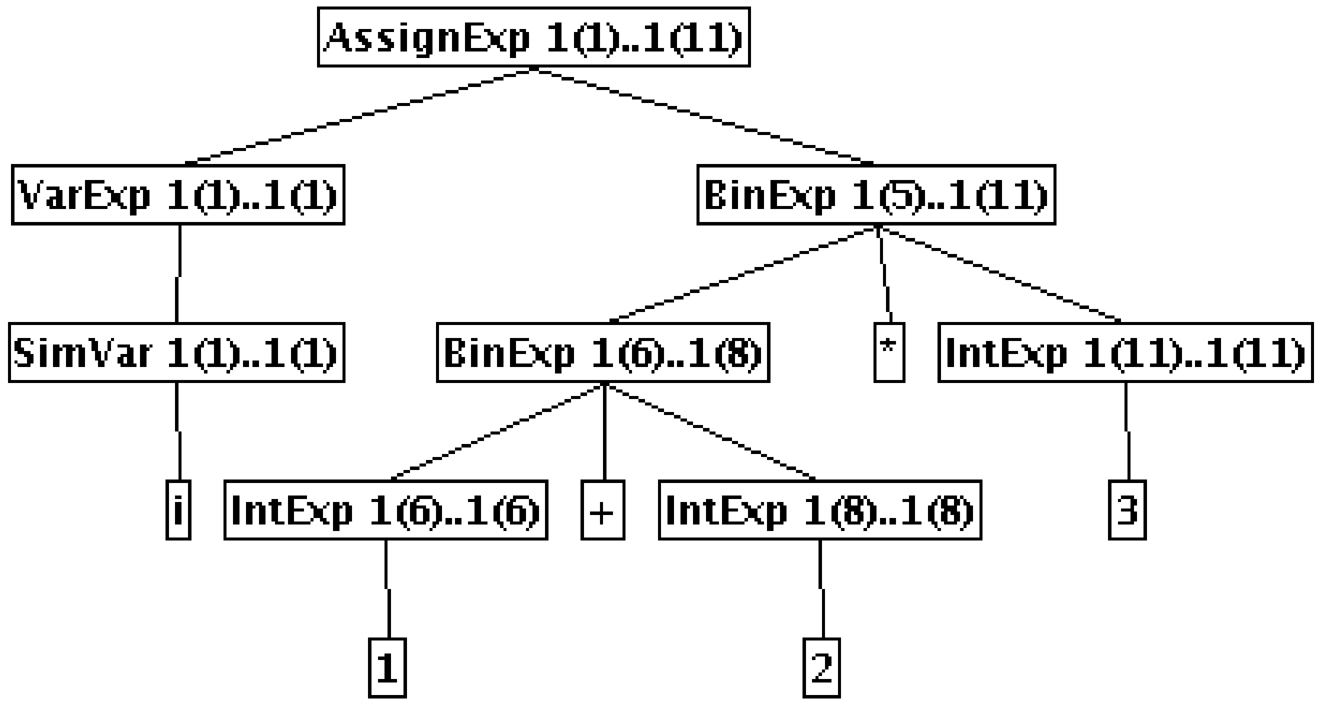


Example (Cont'd)

- Program (ex.vc):

```
i = (1+2)*3;
```

- The AST (using option "-astp"):



Example (Cont'd)

- Program (ex.vc):

```
i = (1+2)*3;
```

- The ASCII AST Using "-ast" (ex.vct):

```
AssignExpr
  VarExpr
    SimpleVar
      i
  BinaryExpr
    BinaryExpr
      IntExpr
        1
      +
      IntExpr
        2
    *
    IntExpr
      3
```

Example (Cont'd)

- The unparsed VC program (ex.vcu):

  ```
  (i=((1+2)*3));
  ```

  - The UnParser.java demonstrates the implementation of a pretty printer or editor
  - Our UnParser is not quite a pretty printer since some information in the original VC is missing in the AST (see Slide 252)
  - UnParser:
    - Will be used for marking Assignment 3
    - Can be used for debugging your solution (see spec)

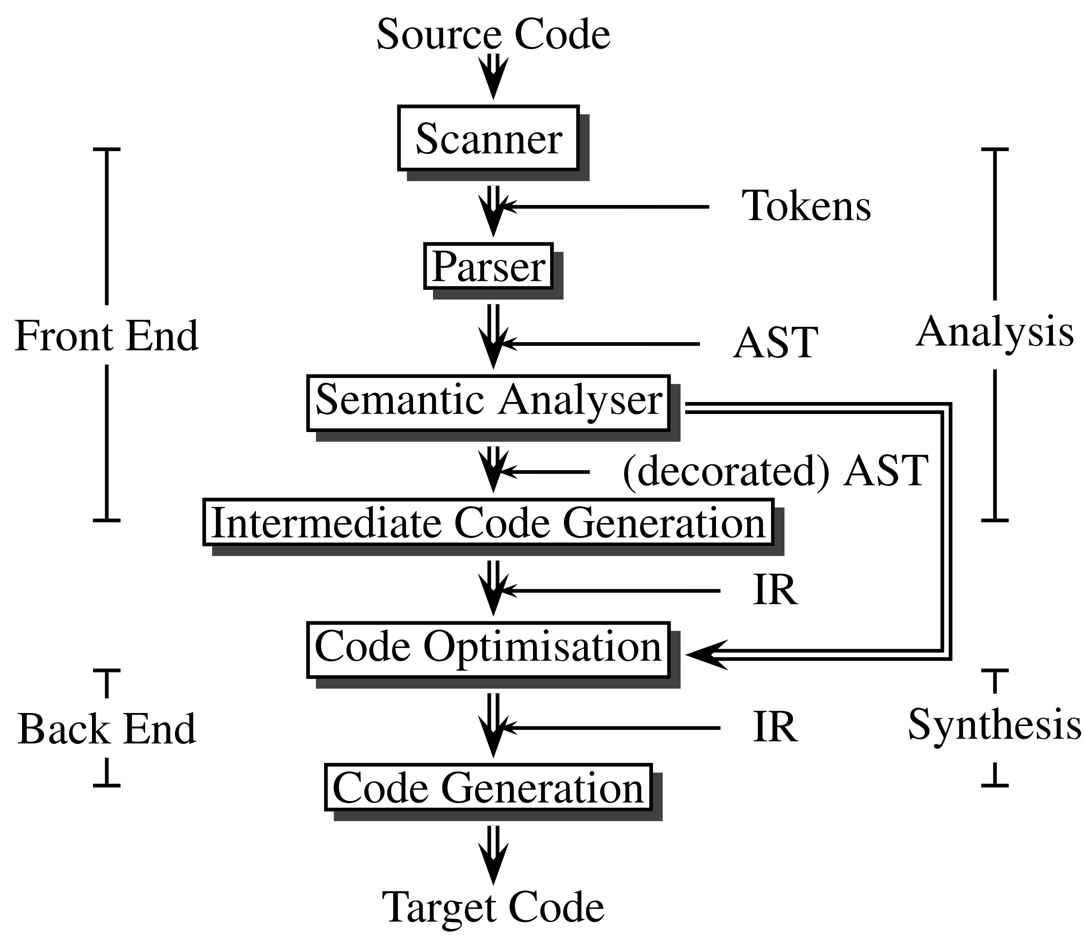
Depth-First Left-To-Right Traversal

- The later phases of a compiler typically involves a depth-first left-to-right traversal of the tree (p 37 Red/p 57 Purple):

```
void traverse(AST n) {
  1 visit(n) // do something at n
  for (each child of m of n from left to right)
    traverse(m);
  2 visit(n)
}
```

- In general, a node can be visited or processed
  - 1 before all its children,
  - 2 after all its children, or
  - 3 in between the visits to its children
- This traversal used in all three tree packages

The Typical Structure of A Compiler (Slide 13)



Informally, error handling and symbol table management also called “phases”.

(1) Analysis: breaks up the program into pieces and creates an intermediate representation (IR), and

(2) Synthesis: constructs the target program from the IR

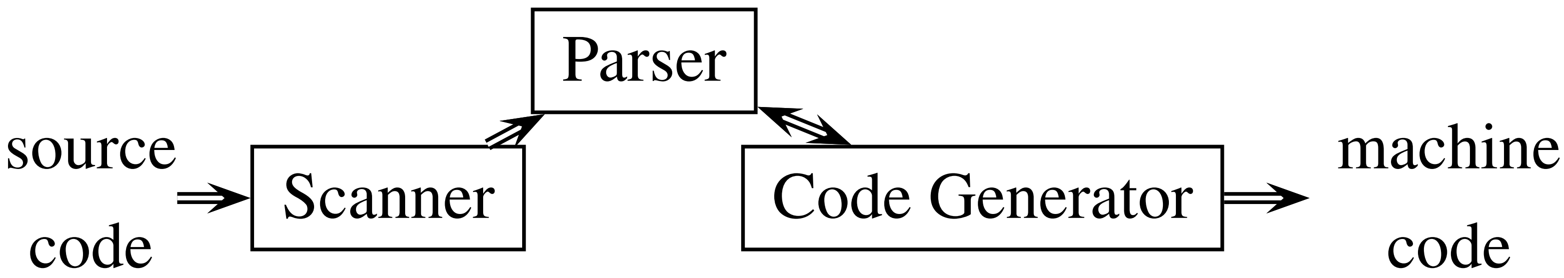
Passes

A pass

1. reads the source program or output from a previous pass,
2. makes some transformations, and
3. then writes output to a file or an internal data structure

Traditionally, a pass is the process of reading a file from the disk and writing a file to the disk. This concept is getting murky now.

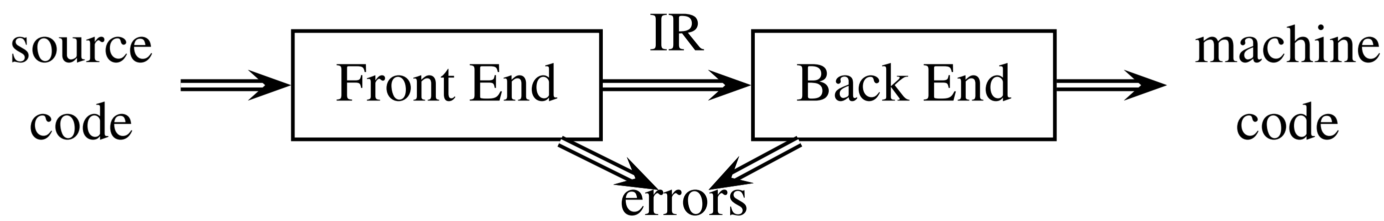
One-Pass Compilers



- Code generation done as soon as a construct is recognised
- Easy to implement
- Code inefficient because optimisations are hardly done
- Difficult to implement for some languages such as PL/1 where variables are used before defined
- An example: Wirth's first Pascal Compilers

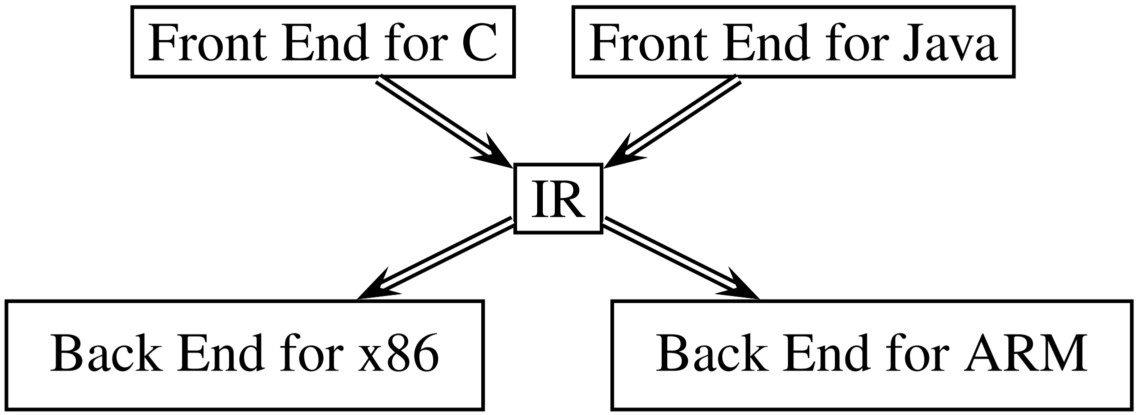
Two-Pass Compilers

- Most production compilers are multi-pass compilers
- The typical structure of a two-pass compiler:



- An example: Ritchie and Johnson's C compilers
- Many assemblers work in two passes
- Why (Intermediate Representation) IR?
  - Simplify retargeting
  - Sophisticated optimisations possible on IR
  - IR can be processed in any order without being constrained by the parsing as in one-pass compilers

Front and Back Ends $\Longrightarrow$ Retargetable Compilers



- Efficient code can be done on IR
- An optimising compiler optimises IR in many passes
- Simplify retargeting

  M languages + N architectures $\Longrightarrow$ M frontends + N backends not MN frontends + NN backends

Why a Physical (or Explicit) Tree?

- Tree is one of intermediate representations (IR)
  - The syntactic structure represented explicit
  - The semantics (e.g., types, addresses, etc.) attached to the nodes
- The question then becomes: “why IR?”

Modern Optimising Compilers

- Optimising the program in multiple passes

$$\text{IR} \Longrightarrow \boxed{\text{opt pass 1}} \xRightarrow{\text{IR}} \cdots \xRightarrow{\text{IR}} \boxed{\text{opt pass } n} \Longrightarrow \text{IR}$$

- Examples: Java bytecode optimisers
  (`http://www.bearcave.com/software/java/comp_java.html`)
- Common optimisations – (covered earlier in COMP4133)
  - Loop optimisation
  - Software pipelining
  - Locality optimisation
  - Inter-procedural analysis and optimisation
  - etc.

Week 3 (2nd Lecture): Abstract Syntax Trees (ASTs)

1. Assignment 3 $\surd$
2. Why a physical tree? $\surd$
3. Parse trees v.s. syntax trees
4. Design of AST classes
5. Use of AST classes
6. Attribute grammar
7. Implementation details specific to Assignment 3

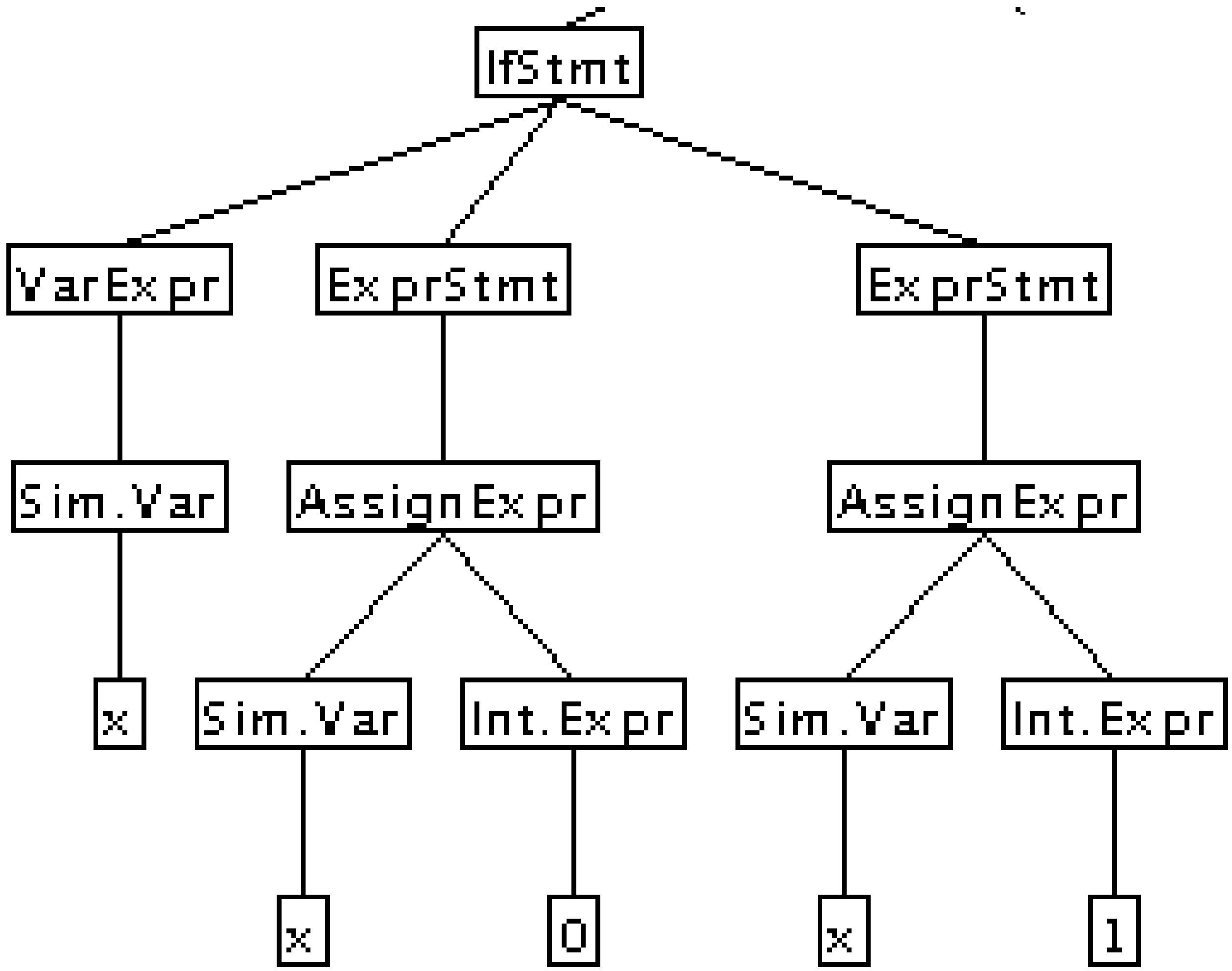
Phrases

- A phrase of a grammar G is a string of terminals labelling the terminal nodes (from left to right) of a parse tree
- An A-phrase of G is a string of terminals labelling the terminal nodes of the subtree whose root is labelled A.
- Formally, given $G = (V_T, V_N, P, S)$, if

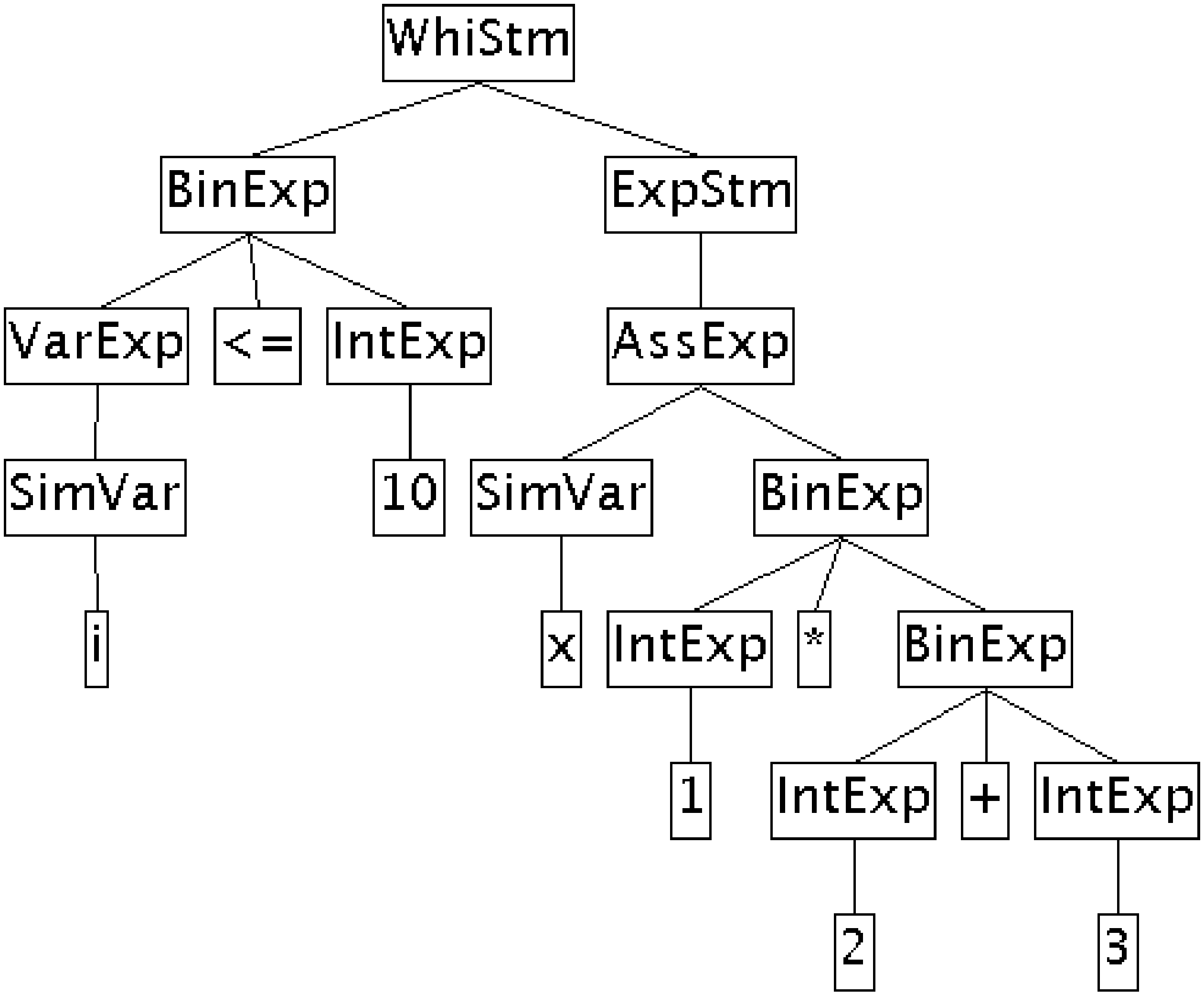
$$\begin{array}{llll} S & \Longrightarrow^* & uwv & uwv \text{ is a sentential form} \\ S & \Longrightarrow^* & uAv & \text{for some } A \in V_N\text{, and} \\ A & \Longrightarrow^+ & w & \text{for some } w \in V_T^+\text{, and} \end{array}$$

  then w is a phrase (which is in fact an A-phrase)
- Examples:
  - An if-phrase has 3 subphrases: an expression and two statements
  - An while-phrase has 2 subphrases: an expression and a statement

An if-phrase has 3 subphrases



An while-phrase has 2 subphrases



Parse Trees (or Concrete Syntax Trees)

- Specifies the syntactic structure of the input
- The underlying grammar is a concrete syntax for the language
- Used for parsing (i.e., deciding if a sentence is syntactically legal)
- Has one leaf for every token in the input and one interior node for every production used during the parse

Syntax Trees or (Abstract Syntax Trees)

- Specifies the phrase structure of the input
- More compressed representation of parse tree
  - Nonterminals used for defining operators precedence and associativity should be confined to the parsing phase
  - Separators (punctuation) tokens are redundant in later phases
  - Keywords implicit in tree nodes
- Abstract syntax can be specified using an attribute grammar
- Used in type checking, code optimisation and generation

The Expression Grammars

- The grammar with left recursion:

Grammar 1: $$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to \textbf{INT} \mid (E) \end{aligned}$$

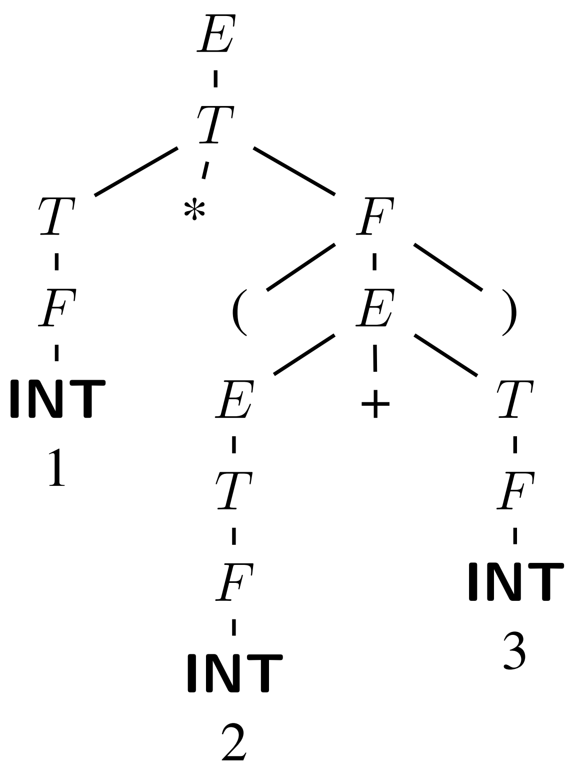
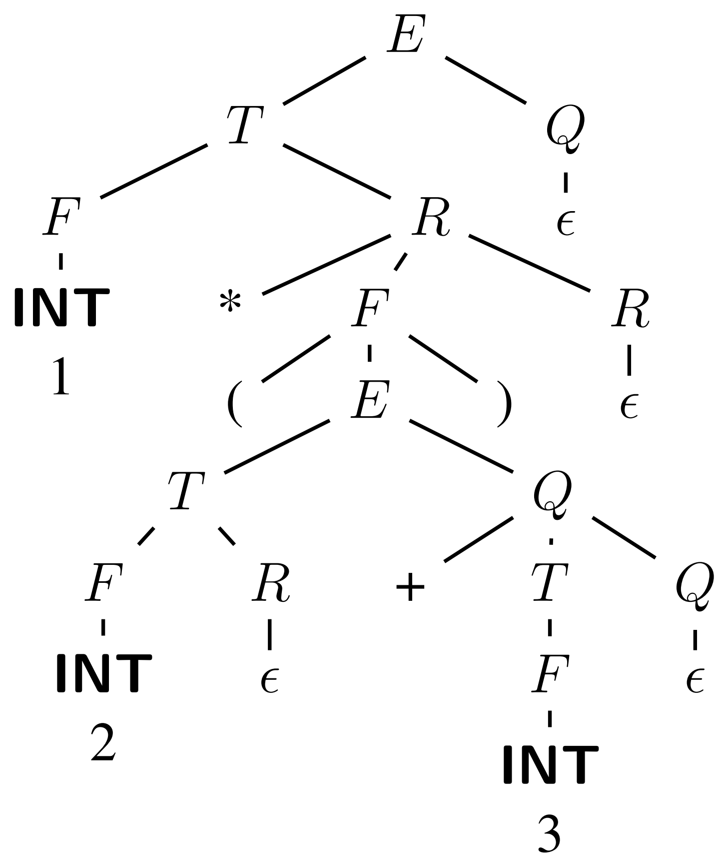
- The transformed grammar without left recursion:

Grammar 2: $$\begin{aligned} E &\to TQ \\ Q &\to +TQ \mid -TQ \mid \epsilon \\ T &\to FR \\ R &\to *FR \mid /FR \mid \epsilon \\ F &\to \textbf{INT} \mid (E) \end{aligned}$$

- An expression grammar (Slide 150):

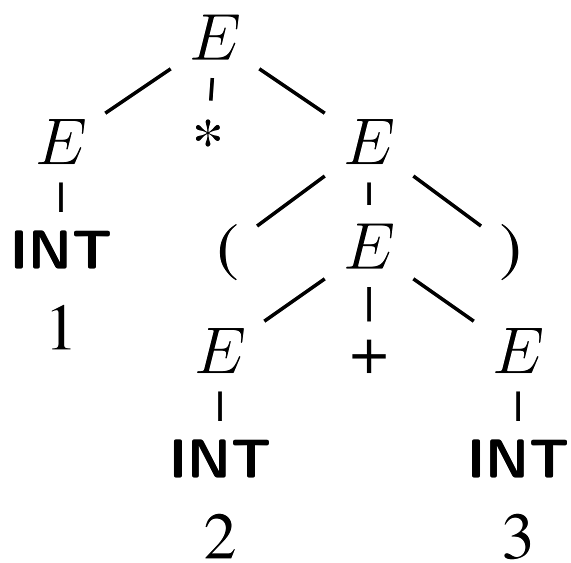
Grammar 3: $E \to E+E \mid E-E \mid E/E \mid E*E \mid (E) \mid \textbf{INT}$

Parse Trees for $1 * (2 + 3)$

Grammar 1	Grammar 2
	

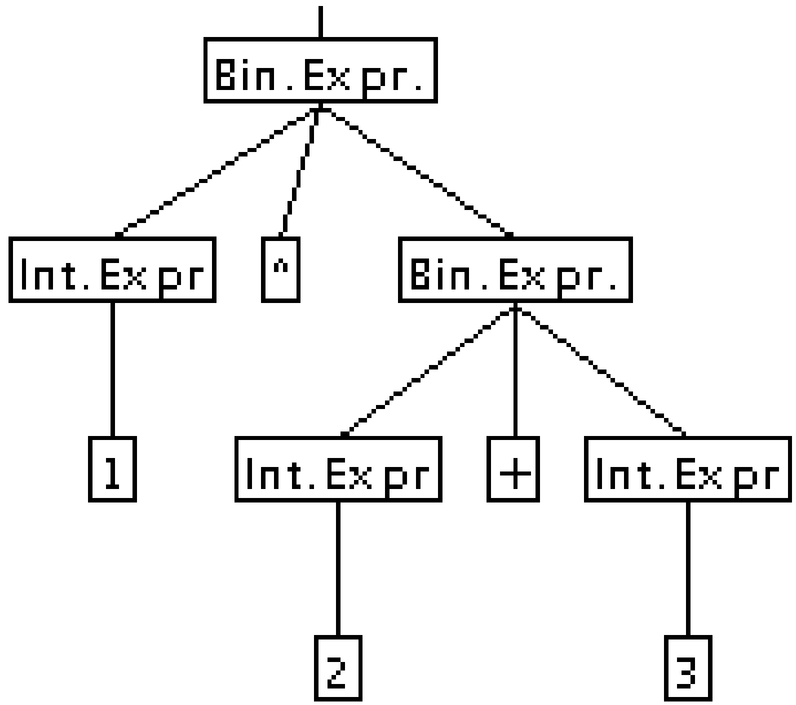
- A depth-first traversal yields the expression being analysed
- Grammar 1 unsuitable for top-down parsing (due to left recursion)
- The tree for Grammar 2 is unnatural

Parse Tree for $1 * (2 + 3)$ Using Grammar 3



- The parse tree is unique for this expression
- But more than one parse tree exist in general (Week 2)
- The (correct) parse trees look more natural but Grammar 3 is ambiguous!

The AST in VC for $1 * (2 + 3)$



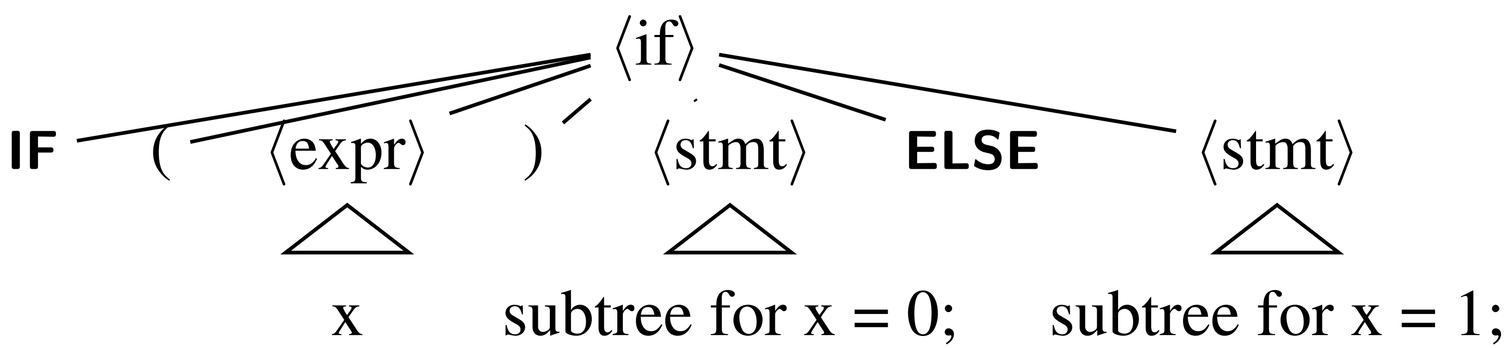
- The separators "(" and ")" are not needed, because the meaning of the expression in the AST is clear
- Nonterminals such as term and factor are not needed, because the operator precedence in the AST is clear

The Parse Tree for a VC If Statement

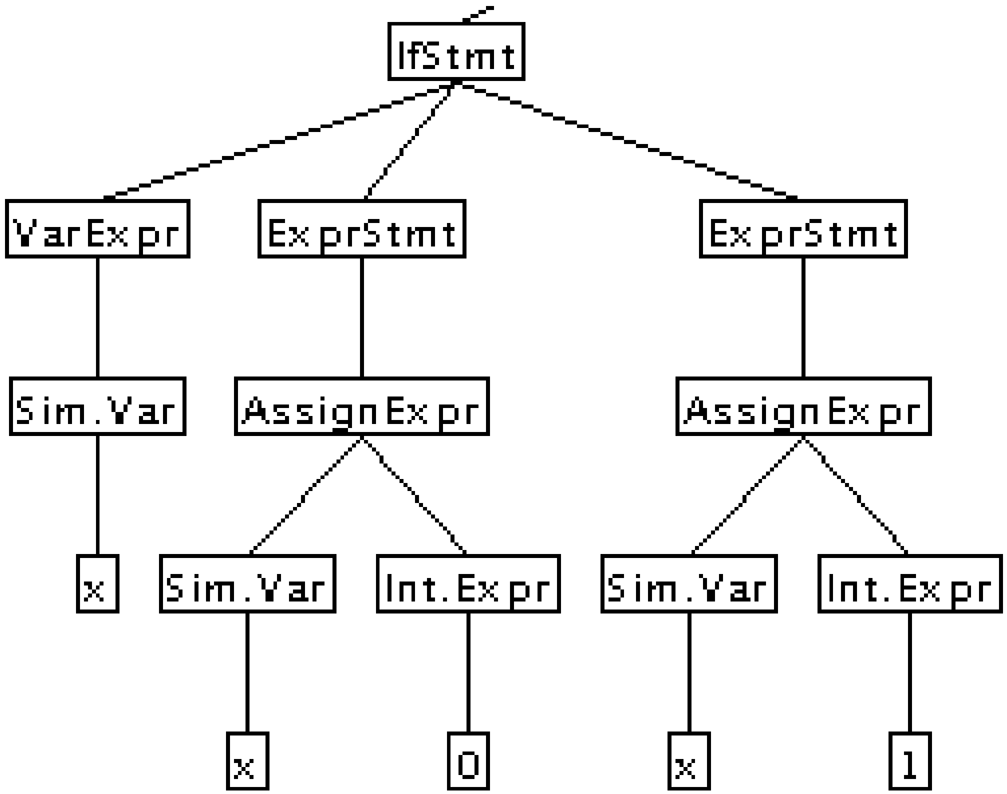
- The If statement:

```
void main(){
   if (x)
      x = 0;
   else
      x = 1;
}
```

- The parse tree



The AST for a VC If Statement



- The separators "(", ")" and ";" are not needed
- Keyword **if** and **else** implicit in the AST nodes

Week 3 (2nd Lecture): Abstract Syntax Trees (ASTs)

1. Assignment 3 $\surd$
2. Why a physical tree? $\surd$
3. Parse trees v.s. syntax trees $\surd$
4. Design of AST classes
5. Use of AST classes
6. Attribute grammar
7. Implementation details specific to Assignment 3

Design of AST Classes

- Can be formally specified using a grammar `http://pdf.aminer.org/000/161/377/the_zephyr_abstract_syntax_description_language.pdf`
- Then the AST classes can be generated automatically
- The structure of AST classes in a compiler:
  - AST.java is the top-level abstract class
  - In general, one abstract class for a nonterminal and one concrete class for each of its production alternatives
- In VC,
  - the EmptyXYZ AST classes introduced to avoid the use of null (nothing fundamental but a design decision made here)
  - Package TreeDrawer assumes no null references

Use of AST Classes

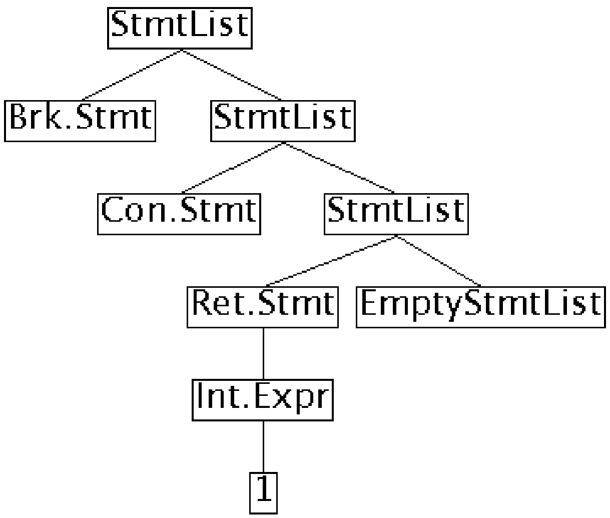
```
SourcePosition pos = new SourcePosition();

Stmt s1 = new BreakStmt(pos);
Stmt s2 = new ContinueStmt(pos);

IntLiteral il = new IntLiteral("1", pos);
IntExpr ie = new IntExpr(il, pos);
Stmt s3 = new ReturnStmt(ie, pos);

List sl = new StmtList(s3, new EmptyStmtList(pos), pos);
sl = new StmtList(s2, sl, pos);
sl = new StmtList(s1, sl, pos);
```

```
break;
continue;
return 1;
```



Use of AST Classes (Cont'd)

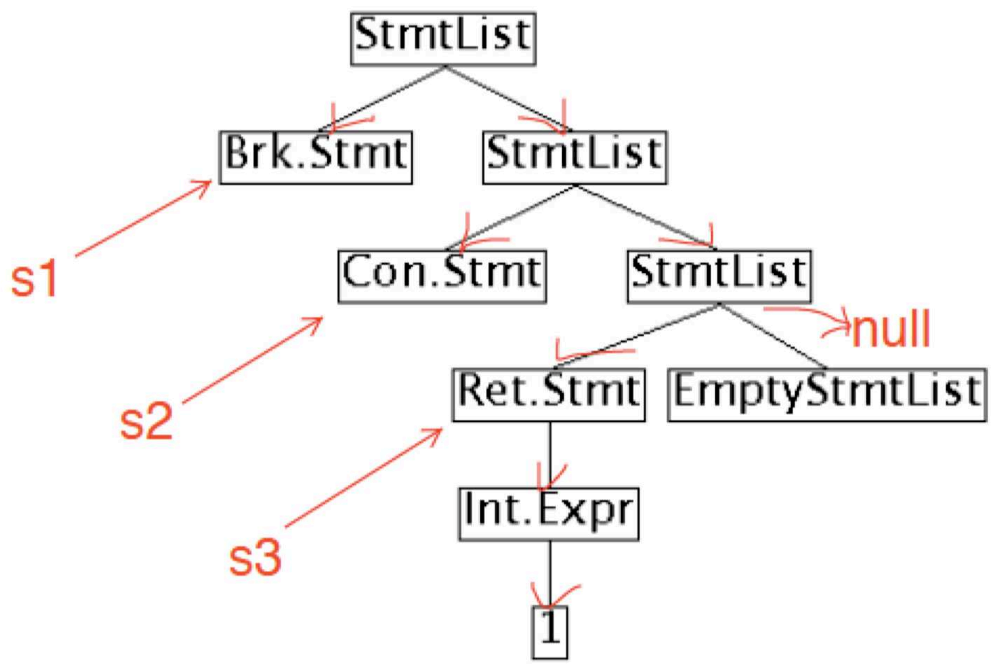
```
SourcePosition pos = new SourcePosition();

Stmt s1 = new BreakStmt(pos);
Stmt s2 = new ContinueStmt(pos);

IntLiteral il = new IntLiteral("1", pos);
IntExpr ie = new IntExpr(il, pos);
Stmt s3 = new ReturnStmt(ie, pos);

List sl = new StmtList(s3, new EmptyStmtList(pos), pos);
sl = new StmtList(s2, sl, pos);
sl = new StmtList(s1, sl, pos);
```

```
break;
continue
return 1
```



How to Test AST Classes

```
import VC.TreeDrawer.Drawer;
import VC.ASTs.*;
import VC.Scanner.SourcePosition;

public class ASTMaker {
  private static Drawer drawer;
  ASTMaker() { }

  List createAST() {
    SourcePosition pos = new SourcePosition();

    Stmt s1 = new BreakStmt(pos);
    Stmt s2 = new ContinueStmt(pos);

    IntLiteral il = new IntLiteral("1", pos);
    IntExpr ie = new IntExpr(il, pos);
    Stmt s3 = new ReturnStmt(ie, pos);

    List sl = new StmtList(s3, new EmptyStmtList(pos), pos);
    sl = new StmtList(s2, sl, pos);
    sl = new StmtList(s1, sl, pos);

    return sl;
    }

  public static void main(String args[]) {
    ASTMaker o = new ASTMaker();
    AST theAST = o.createAST();
    Drawer drawer = new Drawer();
    drawer.draw(theAST);
    }
}
```

Understanding the Visior Design Pattern (Assignments 4 & 5))

- Read Visitor.java – the visitor interface
- Every Visitor class must implement the Visitor interface
- Read AST.java for the abstract visit method
- Every concrete AST A implements the visit method by simply calling the visitor method VisitA in the interface

A understanding of the pattern unnecessary for Assignment 3 but critical for Assignments 4 & 5

Understanding the Visior Design Pattern (Cont'd)

- The free pattern book:

  `http://www.freejavaguide.com/java-design-patterns.pdf`

- Understand the visitor pattern under the Behavioural Patterns before Week 6

- Read this and study the implementation of TreeDrawer

Attribute Grammars

An attribute grammar is a triple:

$$A = (G, V, F)$$

where

- G is a CFG,
- V is a finite set of distinct attributes, and
- F is a finite set of semantic rules (semantic computation and predicate) functions about the attributes.

Note:

- Each attribute is associated with a grammar symbol
- Each semantic rule is associated with a production that makes reference only to the attributes associated with the symbols in the production

Attributes Associated with a Grammar Symbol

A attribute can represent anything we choose:

- a string
- a number
- a type
- a memory location
- etc.

An Attribute Grammar for Converting Infix to Postfix

PRODUCTION	SEMANTIC RULE
$E \rightarrow T$	$[E.t = T.t]$
$\quad \mid \; E_1$ "+" T	$[E.t = E_1.t \parallel T.t \parallel$ "+"$]$
$\quad \mid \; E_1$ "-" T	$[E.t = E_1.t \parallel T.t \parallel$ "-"$]$
$T \rightarrow F$	$[T.t = F.t]$
$\quad \mid \; T_1$ "$*$" F	$[T.t = T_1.t \parallel F.t \parallel$ "*"$]$
$\quad \mid \; T_1$ "/" F	$[T.t = T_1.t \parallel F.t \parallel$ "/"$]$
$F \rightarrow$ **INT**	$[F.t =$ **int**.string-value$]$
$F \rightarrow$ "(" E ")"	$[F.t = E.t]$

- A single string-valued attribute t
- $\parallel$: string concatenation

An Attribute Grammar for Converting Infix to Postfix

PRODUCTION	SEMANTIC RULE
$E \rightarrow T$	$[E.t = T.t]$
$($"+" T |	$[E.t = E.t \parallel T.t \parallel$ "+"$]$
"-" T	$[E.t = E.t \parallel T.t \parallel$ "-"$]$
$)^*$	
$T \rightarrow F$	$[T.t = F.t]$
$($"*" F |	$[T.t = T.t \parallel F.t \parallel$ "*"$]$
"/" F	$[T.t = T.t \parallel F.t \parallel$ "/"$]$
$)^*$	
$F \rightarrow$ **INT**	$[F.t =$ **int**.string-value$]$
$F \rightarrow$ "(" E ")"	$[F.t = E.t]$

- A single string-valued attribute t
- $\parallel$: string concatenation

Tracing the execution of the parser in Slide 272 on 1+2+3 and 1+2*3 to understand this grammar.

The Driver for the Parser in Slide 272

```
/*
 * Expr.java
 *
 */

import VC.Scanner.Scanner;
import VC.Scanner.SourceFile;
import VC.ErrorReporter;

public class Expr {

    private static Scanner scanner;
    private static ErrorReporter reporter;
    private static Parser parser;

    public static void main(String[] args) {

        if (args.length != 1) {
            System.out.println("Usage: java Compiler filename");
            System.exit(1);
        }

        String sourceName = args[0];
        System.out.println("*** " + "The Expression compiler " + " ***");

        SourceFile source = new SourceFile(sourceName);
```

```
        reporter = new ErrorReporter();
        scanner  = new Scanner(source, reporter);
        parser   = new Parser(scanner, reporter);

        parser.parseGoal();

        if (reporter.numErrors == 0)
           System.out.println ("Compilation was successful.");
        else
           System.out.println ("Compilation was unsuccessful.");

    }
}
```

A Parser Implementing the Attribute Grammar in Slide 269

```
public void parseGoal() {
  String Et = parseE();
  if (currentToken.kind != Token.EOF) {
    syntacticError("\"%\" invalid expression", currentToken.spelling);
  } else
    System.out.println("postfix expression is: " + Et);
}

public String parseE() {
  String Tt = parseT();
  String Et = Tt;
  while (currentToken.kind == Token.PLUS
      || currentToken.kind == Token.MINUS) {
      String op = currentToken.spelling;
      accept();
      Tt = parseT();
      Et = Et + Tt + op;
  }
 return Et;
}

String parseT() {
  String Ft = parseF();
  String Tt = Ft;
  while (currentToken.kind == Token.MULT
      || currentToken.kind == Token.DIV) {
      String op = currentToken.spelling;
      accept();
      Ft = parseF();
      Tt = Tt + Ft + op;
```

```
    }
  return Tt;
  }

  String parseF() {
    String Ft = null;
    switch (currentToken.kind) {
    case Token.INTLITERAL:
        Ft = currentToken.spelling;
        accept();
        break;
    case Token.LPAREN:
        accept();
        String Et = parseE();
        Ft = Et;
        match(Token.RPAREN);
        break;
    default:
        syntacticError("\"%\" cannot start F", currentToken.spelling);
        break;
    }
  return Ft;
  }
}
```

An Attribute Grammar for Constructing ASTs

PRODUCTION	SEMANTIC RULE
$E \rightarrow T$	$[E.ast = T.ast]$
$\quad$ ("+" T	$[E.ast = \langle\text{new BinaryExpr}\rangle(E.ast, \text{"+"}, T.ast)]$
$\quad$ "-" T	$[E.ast = \langle\text{new BinaryExpr}\rangle(E.ast, \text{"-"}, T.ast)]$
$\quad$)*	
$T \rightarrow F$	$[T.ast = F.ast]$
$\quad$ ("*" F	$[T.ast = \langle\text{new BinaryExpr}\rangle(T.ast, \text{"*"}, F.ast)]$
$\quad$ "/" F	$[T.ast = \langle\text{new BinaryExpr}\rangle(T.ast, \text{"/"}, F.ast)]$
$\quad$)*	
$F \rightarrow$ **INT**	$[F.ast = \langle\text{new IntExpr}\rangle(\langle\text{new IntLiteral}\rangle(\textbf{int}.\langle\text{str}\rangle))]$
$F \rightarrow$ "(" E ")"	$[F.ast = E.ast]$

- A single attribute ast denoting a reference to a node object
- BinaryExpr, IntExpr, IntLiteral are AST constructors
- A parser for building the AST can be written similarly as the one in Slide 272 except that t is replaced with ast!

Parsing Method for $A \rightarrow \alpha$

private AST_A parseA() {
  AST_A itsAST;
  parse α and constructs itsAST;
  return itsAST;
}

where AST_A is the abstract class for the nonterminal A.

- parseA parses a A-phrase and returns its AST as a result
- The body of parseA constructs the A-phrase's AST by combining the ASTs of its subphrases (or by creating terminal nodes)

The Parsing Method for Statement

```
Stmt parseStmt() throws SyntaxError {
    Stmt sAST = null;
    switch (currentToken.kind) {

    case Token.LCURLY:
      sAST = parseCompoundStmt();
      break;

    case Token.IF:
      sAST = parseIfStmt();
      break;
    ...
}
```

- parseCompoundStmt, parseIfStmt, ... return concrete nodes or objects, which are instances of concrete AST classes, CompoundStmt, IfStmt, ...
- The return type Stmt is abstract; Stmt is the abstract class for the nonterminal stmt in the VC grammar

Implementation Details Specific to Assignment 3

1. ASTs must reflect correctly operator precedence and associativity
2. All lists (StmtList, DeclList, ArgList and ParaList) implemented as:
   - `EBNF:` ⟨StmtList⟩ `->` `(` ⟨something⟩ `)`*
   - `BNF (with right recursion):`

     $\texttt{StmtList}_1$ `->`

     $\quad\epsilon$

     `|` ⟨Stmt⟩

     `|` ⟨Stmt⟩ ⟨StmtList_2⟩
   - The Attrbute Grammar:

     $\texttt{StmtList}_1$ `->`

     $\quad\epsilon$

     $\quad\texttt{StmtList}_1$`.AST = new EmptyStmtList()`

     `|` ⟨Stmt⟩

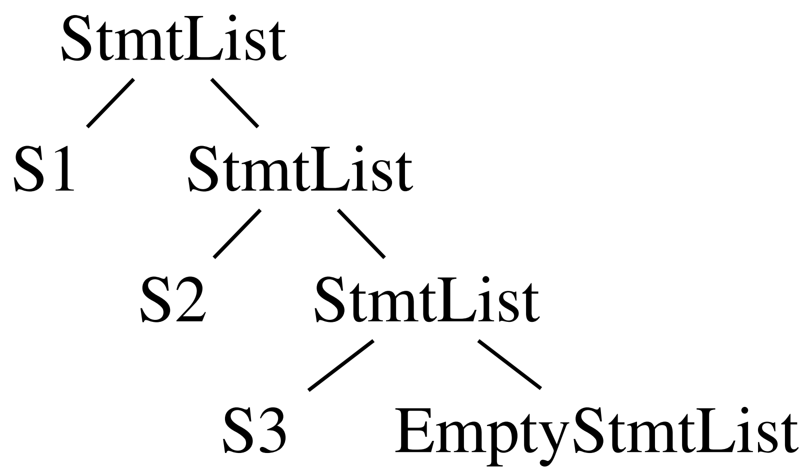
     $\quad\texttt{StmtList}_1$`.AST = new StmtList(Stmt.AST, new EmptyStmtList)`

     `|` ⟨Stmt⟩ ⟨StmtList_2⟩

     $\quad\texttt{StmtList}_1$`.AST = new StmtList(Stmt.AST,` $\texttt{StmtList}_2$`.AST)`
   - See parseStmtList in Parser.java
   - See the supplied test cases for Assignment 3

Implementation Details Specific to Assignment 3 (Cont'd)

S1; S2; S3 $\Longrightarrow$



3. Create EmptyExpr nodes for empty expressions:

```
Expr-Stmt -> Expr? ";"       when ExprStmt = ";"
```

4. All references must not be null – Use EmptyXYZ

Reading

- Assignment 3 spec (to be released soon)
- Syntax trees: §5.2 of Red Dragon (§5.3.1 of Purple Dragon)
- Attribute grammar (or syntax-directed translation):
  - Pages 279 – 287 of (Red) / §5.1 (Purple)
  - Section 2.3 (Red & Purple)

Next Class: Attribute Grammars