# CSE P 501 – Compilers

ASTs, Modularity, and the Visitor Pattern

Hal Perkins

Autumn 2021

# Administrivia

- Scanner due Thursday night
  - Push to Gitlab then tag scanner-final and push tag.
    - GitLab accounts OK? Any other logistics issues?
  - Should be fairly straightforward, but do need to figure out what tokens exist in MiniJava
  - Remember that the scanner doesn't know or care if the token stream makes any sense as a MiniJava program.
  - Will do our best to sanity check over the weekend before parser/AST
- New HW3 (LR constr., LL grammars – today's stuff) out now, due next Monday night
- Parser due in 2 weeks, out now
  - Add parser rules for MiniJava + semantics to build AST
    - Debug grammar rules before adding semantic actions to build the tree
  - Add new visitor to print AST as an indented tree structure
    - Not the same as the AST->source formatter in starter code
    - Needed in any compiler: formatted output of key data structure(s)

# Agenda

- Representation of ASTs as Java objects
- Parser semantic actions and AST generation
- AST operations: modularity and encapsulation
- Visitor pattern: basic ideas and variations
- Some of the “why” behind the “how”

- For the project, see the MiniJava web site and starter code for more details / ideas

# Intermediate Representations

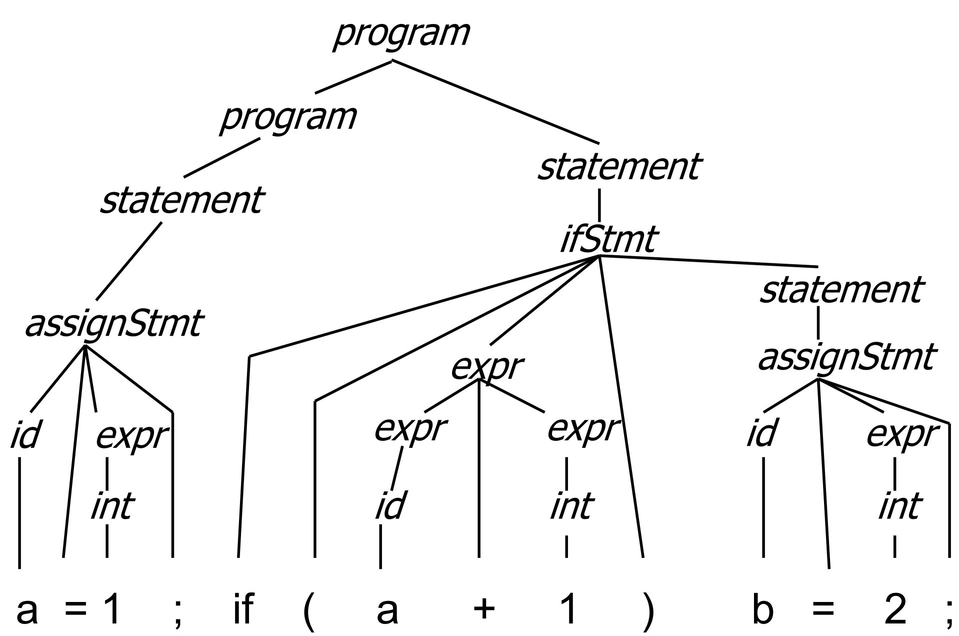
- In most compilers, the parser builds an intermediate representation of the program
  - Typically an AST, as in the MiniJava project
- Rest of the compiler transforms the IR to improve ("optimize") it and eventually translate to final target code
  - Typically will transform initial IR to one or more different IRs along the way
- We'll look at AST's now – other IRs later when we look at optimizations and analysis

# Abstract Syntax Trees (ASTs)

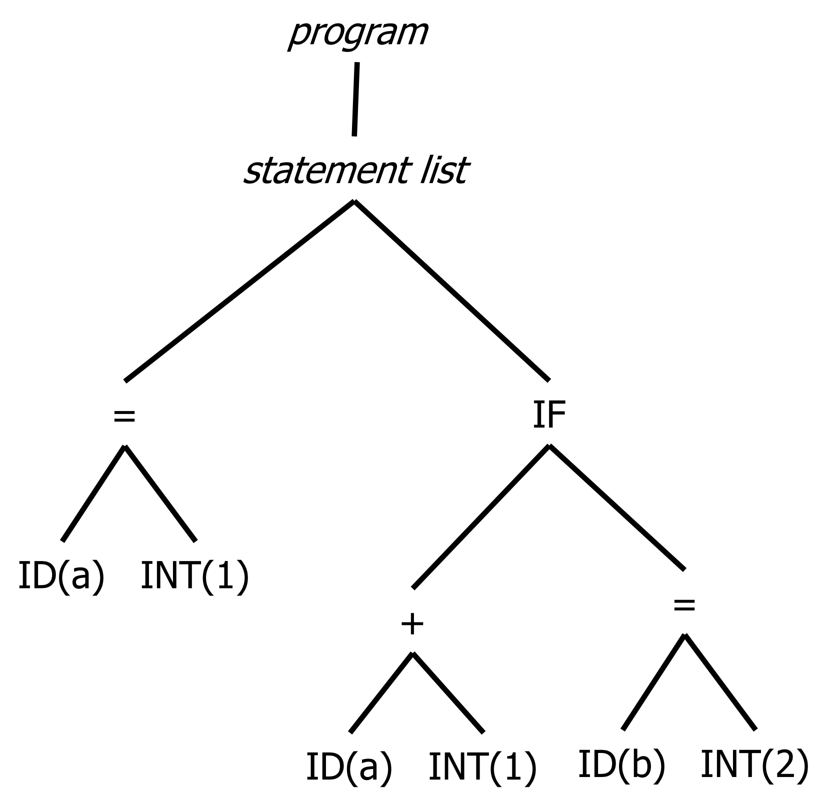
- Idea: capture the essential structure of a program; omit extraneous details
  - i.e, include only what the rest of the compiler needs; omit concrete syntax used only to guide the parse (punctuation, chain productions, etc.)
- Full grammar and derivation needed as part of parsing (it's the control flow for the parser), but a full derivation contains many details that are only needed for parsing, and not after

# Parse Tree / AST example (1)

Full parse tree

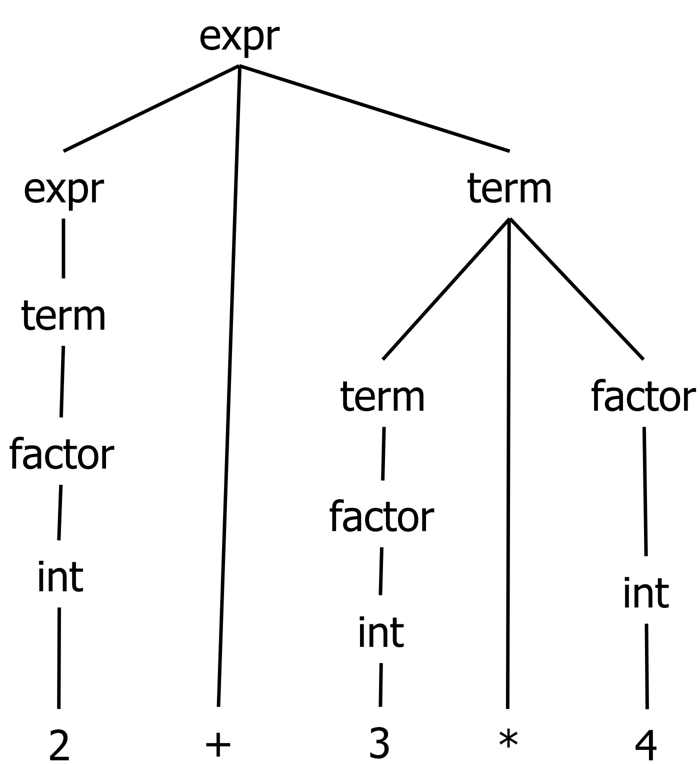



Abstract syntax (AST)

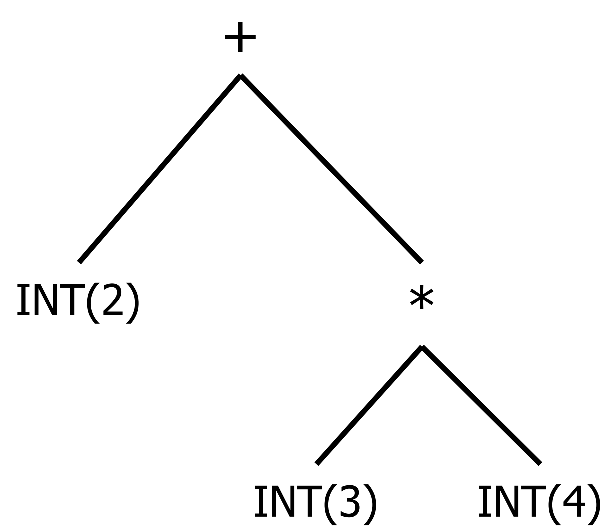


# Parse Tree / AST example (2)

Full parse tree




Abstract syntax (AST)



# Implementing ASTs in Java

- Multiple ways to do this, but typically (and in our our project)
  - Simple tree node objects (basically structs/records)
    - Subtree pointers plus (usually) other useful information like source program locations (e.g., line numbers), links to semantic (symbol table, types) information (later), …
    - But not much more!
    - Basically dumb data structures with public fields, not “smart objects”
  - Use type system and inheritance to factor common information and allow polymorphic treatment of related kinds of nodes

# AST Generation

- Idea: each time the parser recognizes a complete production, it produces as its result an AST node (with links to the subtrees that are the components of the production)
- When we finish parsing, the result of the goal symbol is the complete AST for the program

# MiniJava Starter Code

- AST type hierarchy: root is ASTNode.  Some subclasses:
  - Exp (subclasses: And, Plus, Times, True, Call, …)
  - Statement (subclasses: While, Assign, If, Print, …)
  - Type (abstract rep. of types, *not* source code type declarations – more about that when we get to semantics)
  - Declarations, Classes, others parts of abstract grammar, …
- Additional information in all AST nodes
  - Source code position info (hooks in starter JFlex and CUP rules to capture this, use in error messages, AST printout)
  - accept methods for visitors (more later this lecture)
- Not required to use this AST, but it is *strongly* advised

# Example: AST generation for a Recursive-Descent Parser

```
// parse while (exp) stmt
WhileNode whileStmt() {
    // skip "while ("
    skipToken(WHILE);
    skipToken(LPAREN);

    // parse exp
    ExpNode cond = exp();
```

*(continued next col.)*

```
    // skip ")"
    skipToken(RPAREN);

    // parse stmt
    StmtNode body = stmt();

    // return AST node for while
    return  new WhileNode (cond, body);
}
```

# AST Generation in YACC/CUP

- A result type can be specified for each item in the grammar specification
- Each parser rule can be annotated with a semantic action, which is just a piece of Java code that returns a value of the result type
- The semantic action is executed when the rule is reduced

# YACC/CUP Parser Specification

- CUP code

```
non terminal StmtNode stmt, whileStmt;
non terminal ExpNode exp;
…
stmt ::= …
        | WHILE LPAREN exp:e RPAREN stmt:s
          {:  RESULT = new WhileNode(e,s);  :}
        ;
```

  – See the starter code for examples showing how to capture additional things in the AST like line numbers

# Operations on ASTs

- Once we have the AST, we may want to:
  - Print a readable dump of the tree
  - Print a parseable (source-code) version of the tree (so-called pretty-printing)
  - Do static semantic analysis:
    - Type checking
    - Verify that things are declared and initialized properly
    - Etc. etc. etc. etc.
  - Perform optimizing transformations on the tree
  - Generate code from the tree, or
  - Generate another IR from the tree for further processing

# Modularity

- Classic slogans:
  - Do one thing well
  - Minimize coupling, maximize cohesion
  - Isolate operations/abstractions in modules
  - Hide implementation details

- Okay, so where in the MiniJava compiler does the typechecker module belong?

# Where do the Operations Go?

- Pure “object-oriented” style
  - Really, really, really smart AST nodes
  - Each node knows how to perform every operation on itself

```
public class WhileNode extends StmtNode {
  public WhileNode(…);
  public typeCheck(…);
  public StrengthReductionOptimize(…);
  public DeadCodeEliminationOptimize(…);
  public generateCode(…);
  public prettyPrint(…);
  …
}
```

# Critique

- This is nicely encapsulated – all details about a WhileNode are hidden in that class
- But it is poor modularity
- What happens if we want to add a new optimization (or any other) operation?
  - Have to modify every node class ☹
- Worse: the details of any particular operation (optimization, type checking) are scattered across the node classes

# Modularity Issues

- Smart nodes make sense if the set of operations is relatively fixed, but we expect to need flexibility to add new kinds of nodes
- Example: graphics system
  - Operations: draw, move, iconify, highlight
  - Objects: textbox, scrollbar, canvas, menu, dialog box, window, plus new objects defined as the system evolves
- Another example: objects in a game or simulation

# Modularity in a Compiler

- Abstract syntax does not change frequently over time – language changes are usually incremental
  ∴ Kinds of nodes are relatively fixed
- As a compiler evolves, it is common to modify or add operations on the AST nodes
  – Want to modularize each operation (type check, optimize, code gen) so its parts are together in the source code
  – Want to avoid having to change node classes when we modify or add an operation on the tree

# Two Views of Modularity

| | draw | move | iconify | highlight | transmogrify |
|---|---|---|---|---|---|
| circle | X | X | X | X | X |
| text | X | X | X | X | X |
| canvas | X | X | X | X | X |
| scroll | X | X | X | X | X |
| dialog | X | X | X | X | X |
| … | | | | | |

| | Type check | Optimize | Generate x86 | Flatten | Print |
|---|---|---|---|---|---|
| IDENT | X | X | X | X | X |
| exp | X | X | X | X | X |
| while | X | X | X | X | X |
| if | X | X | X | X | X |
| Binop | X | X | X | X | X |
| … | | | | | |

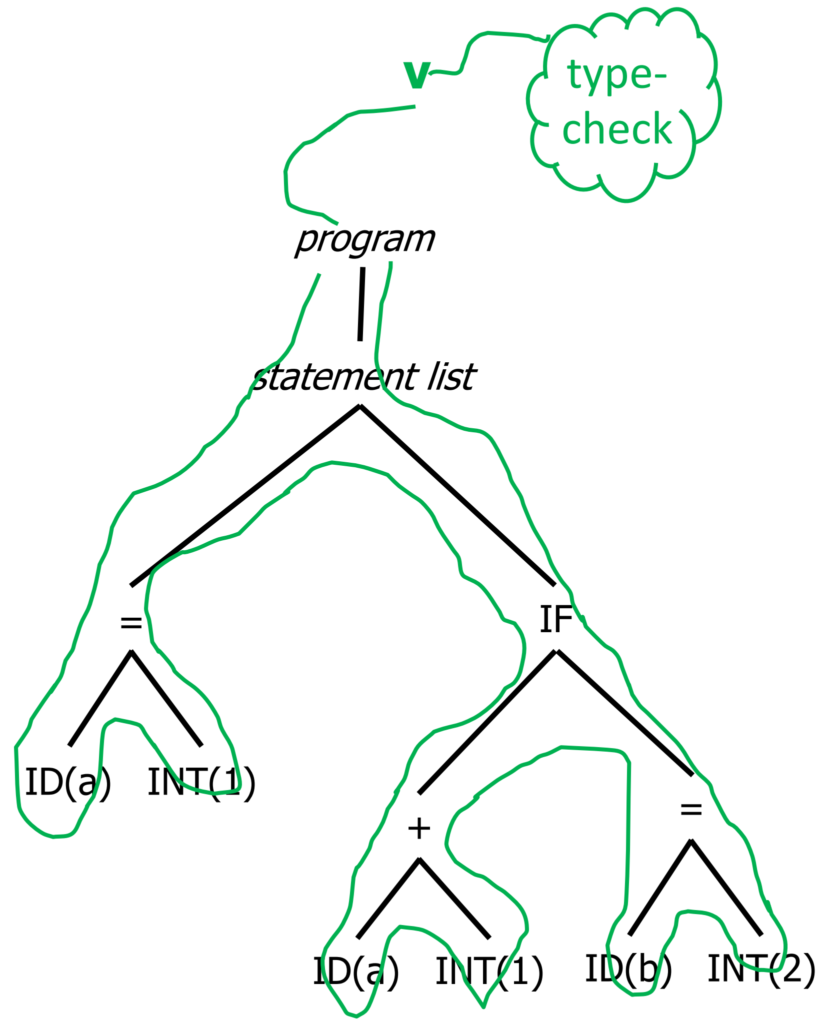
# Visitor Pattern

- Idea: Package each operation (optimization, print, code gen, …) in a separate visitor class (module)
- Create exactly one instance of each visitor class (a singleton!)
  - Sometimes called a "function object"
  - Contains all of the methods for that particular operation, one for each kind of AST node
- Include a generic "accept visitor" method in every node class
- To perform an operation, pass the appropriate "visitor object" around the AST during a traversal

# Here's the idea

To type-check this AST:

1. Create an object (instance) v of the Type-Check visitor class
2. Pass the type-check object to the root note accept(visitor) method
3. Each node passes the visitor object around the tree by calling accept(v) in subtrees to type-check the subtree, and then combine results (a tree traversal)
4. When each node "accepts" the visitor, it arranges to call the visitor method that knows how to type-check *that* particular kind of node



# Visitor issue: avoiding instanceof

- We'd like to avoid huge if-elseif nests in the visitor to discover the node types as it is passed around the tree

```
void checkTypes(ASTNode p) {
    if (p instanceof WhileNode) { … }
    else if (p instanceof IfNode) { … }
    else if (p instanceof BinExp) { … }
…
}
```

# Visitor "Double Dispatch"

- Include a "visit" method for every AST node type in each Visitor

```
void visit(WhileNode);
void visit(ExpNode);
etc.
```

- Include an accept(Visitor v) method in each AST node class
- When Visitor v is passed to an AST node, the node's accept method calls v.visit(this)
  - Selects correct Visitor method for this node
  - Often called "double dispatch", but really single dispatch + overloading

# Visitor Interface

```
interface Visitor {
    // overload visit for each AST node type
    public void visit(WhileNode s);
    public void visit(IfNode s);
    public void visit(BinExp e);
    …
}
```

- Every separate Visitor class implements this interface
- Aside: The result type can be whatever is convenient, doesn't have to be void, although that is common
- Note: could also give methods unique names e.g., visitWhile, visitIf, visitBinExp, etc. instead of overloading visit(…). Best to follow existing code if either convention already adopted, otherwise individual preference.

# Accept Method in Each AST Node Class

- Every AST class overrides accept(Visitor)
- Example

```
public class WhileNode extends StmtNode {
    …
    // accept a visit from a Visitor object v
    @Override
    public void accept(Visitor v) {
        v.visit(this);   // dynamic dispatch on "this" (WhileNode)
    }
    …
}
```

- Key points
  - Visitor object passed as a parameter to WhileNode
  - WhileNode calls visit, which calls visit(WhileNode) automatically because of overloading – i.e., the correct method for this kind of node
- Note: if visitor methods have unique names instead of overloading visit(…) then WhileNode would call something like v.visitWhile(this).

# Composite Objects (1)

- How do we handle composite objects?
- One possibility: the accept method passes the visitor down to subtrees before (or after) visiting itself

```
public class WhileNode extends StmtNode {
  Expr exp; Stmt stmt;   // children
  …
  // accept a visit from visitor v
  public void accept (Visitor v) {
    this.exp.accept(v);
    this.stmt.accept(v);
    v.visit(this);
  }
```

# Composite Objects (2)

- Another possibility: the visitor can control the traversal inside the visit method for that particular kind of node

```
public void visit(WhileNode w) {
    w.expr.accept(this);
    w.stmt.accept(this);
}
```

# So which to choose?

- Possibilities:
  - Node objects drive the traversal and pass the visitors around the tree in standard ways
  - Visitor object drives the traversal (the visitor has access to the node, including references to child subtrees)
- In a compiler:
  - First choice handles many common cases
  - Big compilers often have multiple visitor schemes (e.g., several different traversals defined in Node interface – postorder, inorder, … – plus custom traversals in some visitors)
  - For MiniJava: keep it simple and start with supplied examples, but if you really need to do something different, you can
    - (i.e., keep an open mind, but not so open that you create needless complexity)

# Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - ∴ May need to expose more node state than we might have done otherwise
    - i.e., lots of public fields in AST node objects
  - Overall a good tradeoff – better modularity

    (plus, the nodes should be relatively simple data objects anyway – not hiding much of anything)

# Visitor Actions and State

- A visitor function has a reference to the node it is visiting (the parameter)

  ∴ can access and manipulate subtrees directly

- Visitor object can also include local data (state) shared by methods in the visitor
  - This data is effectively "global" to the methods in the visitor object, and can be used to store and pass around information accumulated by the visit methods

```
public class TypeCheckVisitor extends NodeVisitor {
    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <local state>;        // all typecheck visitor methods can read/write this
}
```

# Why is it so complicated?

- What we're really trying to do: 2-argument dynamic dispatch
  - Pick correct method to execute based on dynamic types of both the node and the visitor
- But Java and most O-O languages only support single dispatch
  - So we use single dispatch plus overloading to get the effect we want

# References

- For Visitor pattern (and many others)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995 (the classic; examples are in old C++ and Smalltalk)
  - *Object-Oriented Design & Patterns*, Horstmann, A-W, 2nd ed, 2006 (uses Java)
- Specific information for MiniJava AST and visitors in Appel textbook & online

# Coming Attractions

- Static Analysis
  - Type checking & representation of types
  - Non-context-free rules (variables and types must be declared, etc.)
- Symbol Tables
- & more

- Later, more about compiler IRs when we get to optimizations