CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

Semantic Analysis

Prompt: What does semantic analysis involve?

- Analyzing the *semantics* (meaning of the program), not simply its structure
- Creating symbol tables and type checking
  1. An important concept in P/L's is the ability to *name* objects
     - Variables, methods, types, classes, etc.
  2. The *type system* of a P/L allows the programmer to make verifiable assertions that can be checked at compile time (instead of runtime)
     - Different P/L's have different approaches to type checking

MiniJava/Ram23

Terminology:

- In MiniJava/Ram23, we must *declare* all named objects
  - Variables must be declared with a type
  - Not all languages have this requirement (e.g. Python doesn't)

Overview of Type Systems

- Most P/Ls assign a type to every value (whether a literal, constant, or variable)
  - Integer, floating-pt, Boolean, string, etc.
  - *Higher-order types:* enums, structs, etc.
- Objectives of Type Systems:
  1. Correctness – raise warnings or errors when possible at compile time
  2. Performance - generate efficient code (e.g. variables that are declared as constants and used many times can possibly be stored in registers)
  3. Expressiveness – type system inference (e.g. single println function for both ints and strings)

Overview of P/Ls:

A P/L is commonly classified as:

- Safe or unsafe
- Static or dynamic
- Explicit or implicit

- **Unsafe Languages**
- C programming language, can modify any word in memory (low-level access needed to build software like OS)
- Can go outside the bounds of an array

- **Safe Languages:**
- Java, C#, Python
- Boundaries of arrays are checked at runtime and potentially yield `IndexOutofBoundsException`

Dynamically-typed Languages:

- Type information stored in memory alongside the data and available at runtime

```
x = 5 // int
x = [3,1] // list
```

- Examples: Python, Lisp, Java's `instanceof` operator
- All (or almost all) type checking happens during program execution
- Easier to rapidly prototype

Statically-typed Languages:

- Variables have types (`int`, `char`, `String` ...)
- Can't assign any other type to it later.
- Example: Java
- Can catch type errors *before* program runs
- All (or almost all) type checking is done as part of compilation
- Avoids overhead of runtime checks and basic machine code doesn't need to retain type information

Explicitly-typed Languages:

- Programmer is responsible for indicating the types of variables explicitly (e.g. Java)
- `int x = 5;`
- `3 + 5 + 7` // legal
- `"Hello" + "World"` // legal
- `"Hello" + 5` (can't add strings and numbers) // illegal

Implicitly-typed Languages:

- interpreter/compiler can perform behind-the-scenes conversions to make operations work
- `double x = 23;`
- `var` keyword in Java (new!)

Symbol Tables

Records all information needed about every declared named item (variables, functions, constants, etc.)

Implementation of Symbol Tables

How to implement?

Seems easy, right? Just make a `HashTable(k,v)`?

- Keys are the identifiers
- Values are the values/objects and types

Gets tricky because of scope.

- (Will assume static scoping, not dynamic scoping for this discussion.)
- Global declarations are fine; local declarations are problematic.

Example

```
{
    int x = 1;
    int y = 2;
    {
        double x = 3.14;
        y += (int)x;
    }
    y += x;
}
```

- Legal in C
- Illegal in Java and MiniJava
- Why won’t a normal hashtable work?
  - The same identifier may refer to different things in different parts of the program.

Another Example

```
class C {
    int a, int b, int c;
    public void m() {
        System.out.println(a + c);
        int j = a + b;
        String a = "hello";
        System.out.println(a);
        System.out.println(j);
        System.out.println(b);
    }
}
```

Notation:

- id → meaning
- + operator: binding in RHS overrides those on left
- *Observe:* not commutative: X + Y ≠ Y + X

- σ_0 (base environment)
- $\sigma_1 = \sigma_0 + \{ a \rightarrow int, b \rightarrow int, c \rightarrow int \}$
- $\sigma_2 = \sigma_1 + \{ j \rightarrow int \}$
- $\sigma_3 = \sigma_2 + \{ a \rightarrow String \}$

More Terminology

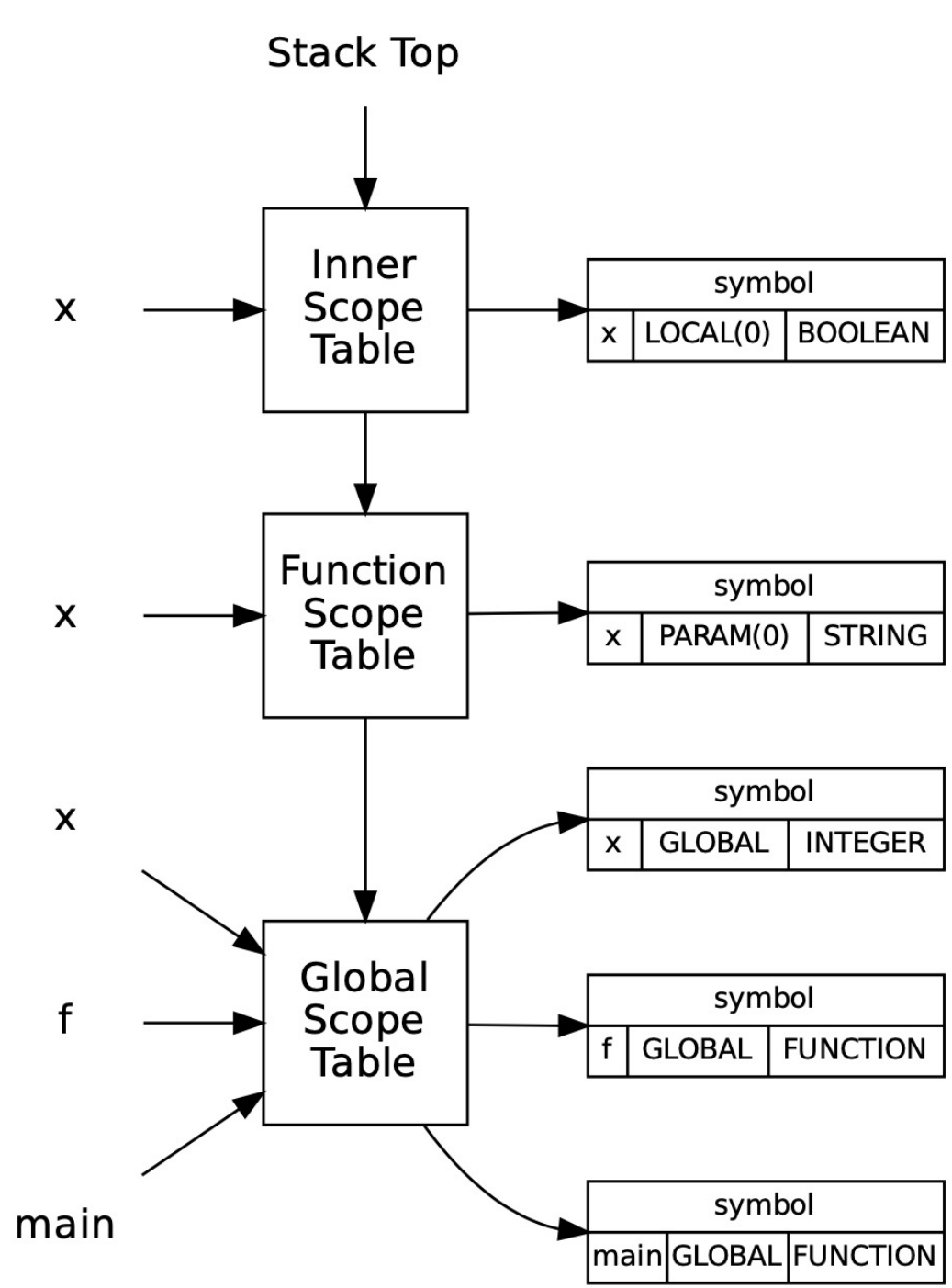
(in the context of *declarations*)

- Scope: local portion of a program that some name is visible
  - (variable, method, class, …)
- Symbol Tables / Environments:
  - Symbol Table: structure that will map identifiers to their types and locations
  - Environment: collection of symbol tables
- Uses: non-defining occurrences of identifiers
  - Lookup identifier in symbol table
- Bindings: identifiers are bound to meanings in the symbol table

```
x: integer = 10;

f: function void ( x: string ) =
{
    print x, "\n";
    {
        x: boolean = false;
        print x, "\n";
    }
}

main: function void () =
{
    print x, "\n";
    f("hello");
}
```



Implementation of Symbol Tables

- Will build symbol table using Visitor design pattern
  - Alternatively could have built during parsing phase using semantic actions (rather than visitors)
- Entries in symbol table contain information about an identifier, such as its lexeme, type, position in storage, “offset” (where to find in memory), etc.
- Symbol tables typically need to support multiple declarations of the same identifier within a program.

Example: Handling Nested Scopes

```
static int w;      // level 0
int x;
void example (int a, int b)   {
    int c;         // level 1
    {
        int b, z; // level 2a
        ...
    }
    {
        int a, x; // level 2b
        ...
        {
            int c, x;    // level 3
            b = a + b + c + w;
            // refers to b_1 = a_2b + b_1 + c_3 + w_0
            // (subscripts are the level)
        }
    }
}
```

Names Declared at each Scope:

0 - w, x, example

1 - a, b, c

2a - b, z

2b - a, x

3 - c, x

Concept: new symbol table (usually a hash table) for every scope.

- “sheaf of tables” – linked together in an order that corresponds to the lexical nesting levels

Operations:

- **Insert**: declarations enter information into current table
- **Lookup**: check current table for variable reference; recursively check lower levels until success of failure (reached the bottom)
- **Initialize Scope**: increments current level; creates new symbol table for that level; links table to enclosing level's table
- **Finalize Scope**: changes current level pointer to point to scope surrounding the current level
  - can preserve table in memory if needed for later use

Example: Sequence of Calls

```
static int w;      // level 0
int x;
void example (int a, int b)   {
    int c;         // level 1
    {
        int b, z; // level 2a
        ...
    }
    {
        int a, x; // level 2b
        ...
        {
            int c, x;    // level 3
            b = a + b + c + w;
            // refers to b_1 = a_2b + b_1 + c_3 + w_0
            // (subscripts are the level)
        }
    }
}
```

1. Init Scope
2. Insert w
3. Insert x
4. Insert example
5. Init Scope
6. Insert a
7. Insert b
8. Insert c
9. Init Scope
10. Insert b
11. Insert z
12. Finalize Scope
13. Init Scope
14. Insert a
15. Insert x
16. Init Scope
17. Insert c
18. Insert x
19. Lookup b
20. Lookup a
21. Lookup b
22. Lookup c
23. Lookup w
24. Finalize Scope
25. Finalize Scope
26. Finalize Scope
27. Finalize Scope

Managing Namespaces (Object-Oriented)

- Need tables for code being compiled.
- Need tables for *external classes* known and referenced in the code.
- Need tables for *inheritance hierarchy*

Example

Performing a lookup for identifier `foo` when compiling method `m` in class `C`

1. First, lookup lexically scoped symbol table for `m`
2. If fails, recursively backtrack and check inheritance hierarchy
3. If fails, check global symbol table

Ram23

- No nesting of scopes in Ram23 and all declarations are at the *beginning* of a method.
- Simplifies construction of symbol tables.

After the Break

Type Checking

Semantic Analysis

1. Symbol Table Construction
2. Type Checking

Type Checking

- Make use of already constructed symbol table.
  - (Two passes through AST?)
- What should the symbol table contain?

Class-level	Method-level	Variable-level
id (the class name)	id	Id
parent? (for simplicity, no inheritance)	parameters	type
variables (globals)	local variables	
methods	return type	

```
class B {
    C f;  int [] j;   int q;
    public int start(int p, int q) {
         int ret;   int a;
         /* ... */
         return ret;
    }
    public boolean stop(int p) {
              /* ... */
        retun false;
    }
}

class C {
   /* ... */
}
```

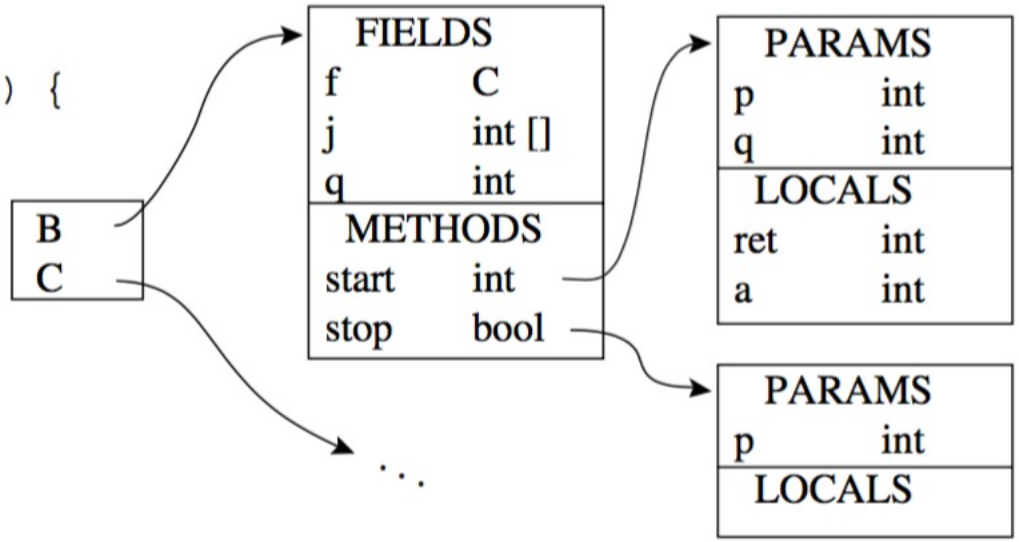


FIGURE 5.7. A MiniJava Program and its symbol table

Remember: MiniJava specifies that there is no nesting of scopes and variables are declared up front.

Today, semantic analysis proceeds in multiple phases:

1. First build symbol table
2. Then perform type checking

Goal is to catch *all* remaining errors before proceedings to back-end compiler phases.

Why Two Passes?

- In a one-pass analysis, we might run into a class or method call that is defined, but not yet entered into the symbol table.
  - (e.g. `Helper()` call in `main` method in ComplexMath.java)

```
class ComplexMath{
    public static void main(String[] a) {
       println(new Helper().go());
    }
}
class Helper {
    public integer go(){
       integer x;
       println(5 * 4 + 3);
       ...
    }

    public bool stop(int a) {
       ...
    }
}
```

- Older compilers insisted on one semantic pass, instead of two.
- *Language requirement:* constraint that methods must be defined before their use.

Step 1: Building a Symbol Table Using a Visitor

`visit` method for when we get to a variable declaration.

```
class ErrorMsg {
    boolean anyErrors;
    void complain(String msg) {
        anyErrors = true;
        System.out.println(msg);
    }
}

// Type t;
// Identifier i;
public void visit(VarDecl n) {
    Type t = n.t.accept(this);
    String id = n.i.toString();

    if (currMethod == null) {
        if (!currClass.addVar(id,t))
            error.complain(id + "is already defined in " + currClass.getId());
    } else if (!currMethod.addVar(id,t))
        error.complain(id + "is already defined in " +currClass.getId()+"."+currMethod.getId());
}
```

MiniJava Abstract Syntax

```
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()
```

What is currMethod and currClass?

```
// Type t;
// Identifier i;
public void visit(VarDecl n) {
    Type t = n.t.accept(this);
    String id = n.i.toString();

    if (currMethod == null) {
        if (!currClass.addVar(id,t))
            error.complain(id + "is already defined in " +
currClass.getId());
    } else if (!currMethod.addVar(id,t))
        error.complain(id + "is already defined in " + currClass.getId() +
"." + currMethod.getId());
}
```

Brainstorming Software Hierarchy of a Symbol Table

1. BuildSymbolTableVisitor.java
   - SymbolTable st;
   - private Class currClass;
   - private Method currMethod
   - “extends” -- nodes like Plus are not interesting to the symbol table creator
     - visit(Plus n) is not extended, defined by TypeDepthVisitor
   - TypeDepthVisitor implements Visitor
     - TypeDepthVisitor performs a "naive" depth-first traversal
2. SymbolTable.java
3. Class.java
4. Method.java
5. Variable.java

```
class SymbolTable {
    Hashtable < String, Class > ht;
    boolean addClass(String id, String parent)
    Class getClass(String id)
    boolean containsClass(String id)
    Type getVarType(Method m, Class id, String id)
    Method getMethod(String mid, String cid)
    Type getMethodType(String mid, String cid)
}
```

```
class Variable {
    String id;
    Type type;
}
```

HANDOUT:

- SymbolTableCode.pdf

```
class Class {
    String id;
    Hashtable method;
    Hashtable globals;
    String parent;
    Type type;
    boolean addMethod(String id, Type type)
    Method getMethod(String id)
    boolean addVar(String id, Type type)
    Variable getVar(String id)
    boolean containsVar(String id)
    boolean containsMethod(String id)
}
```

```
class Method {
    String id;
    Type t;
    Vector params;
    Hashtable vars;
    boolean addParam(String id, Type t)
    Variable getParamAt(int i)
    boolean addVar(String id, Type t)
    boolean containsVar(String id)
    boolean containsParam(String id)
}
```

Step 2: Write Another Visitor to Perform Typechecking

Functions:

- Traverse the AST
- Perform type checking *at appropriate nodes*
- Examples:
  1. At Plus node, can check that operands are integers
     - Defined in starter code in TypeCheckExpVisitor extends TypeDepthFirstVisitor
  2. At VarDecl node, there is nothing to type check
     - Defined in starter code in TypeCheckVisitor extends DepthFirstVisitor

```
// Exp e1,e2;
public Type visit(Plus n) {
    if (! (n.e1.accept(this) instanceof IntegerType) )
        error.complain("Left side of Plus must be of type integer");
    if (! (n.e2.accept(this) instanceof IntegerType) )
        error.complain("Right side of Plus must be of type integer");
    return new IntegerType();
}
```

Type check logic for Plus:

- Plus object has two Exp fields
- Call accept() method of each Exp, which will return the type of the Exp
- Check return types
- Return that Plus will result in an Integer

```
// Exp e1,e2;
public Type visit(Plus n) {
    if (! (n.e1.accept(this) instanceof IntegerType) )
        error.complain("Left side of Plus must be of type integer");
    if (! (n.e2.accept(this) instanceof IntegerType) )
        error.complain("Right side of Plus must be of type integer");
    return new IntegerType();
}
```

Prompt: In an error, we still return `IntegerType`*. Why?*

As many errors as possible should be detected during the compilation process. Don’t just abort after the first one.

What Else Should We Type Check?

- `If` statement condition evaluates to type `Boolean`.
- Type mismatch in assignment (trickier if extension of classes is allowed).
- Operands of logical operators must be type `Boolean`.
- Method calls:
  - type checking the return type
  - checking parameter types against arguments
- …

Overall, requirements for type checking depend on the language.