

Compiler Construction

Semantic Analysis II

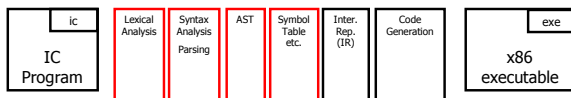
Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University

PAs

- PA2
 - Deadline extension Dec 29
 - Fix PA1 grader's notes in your IC.lex
- PA1
 - Escape characters should stay as they are
 - `<STRING> \n { str.append("\n") }`

2

IC compiler Compiler



We saw:

- Scope
- Symbol tables

Today:

- Type checking
- Recap

3

Examples of type errors

```
(int) a; a = (true);
```

assigned type doesn't match declared type

```
1 < (true)
```

relational operator applied to non-int type

```
void foo(int x) {  
  int x;  
  foo(5,7);  
}
```

argument list doesn't match formal parameters

```
class A {...}  
class B extends A {  
  void foo() {  
    A a;  
    B b;  
    b = a;  
  }  
}
```

a is not a subtype of b

4

Types

- Type
 - Set of values computed during program execution
 - `boolean = {true, false}`
 - `int = {-232, 232}}`
 - `void = {}`
- Type safety
 - Types usage adheres formally defined typing rules

5

Type judgments

- `e : T`
 - Formal notation for type judgments
 - `e` is a well-typed expression of type `T`
 - `2 : int`
 - `2 * (3 + 4) : int`
 - `true : bool`
 - `"Hello" : string`

6

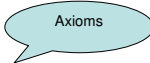
Type judgments

- $E \vdash e : T$
 - Formal notation for type judgments
 - In the context E , e is a well-typed expression of T
 - $b:\text{bool}, x:\text{int} \vdash b:\text{bool}$
 - $x:\text{int} \vdash 1 + x < 4:\text{bool}$
 - $\text{foo}:\text{int} \rightarrow \text{string}, x:\text{int} \vdash \text{foo}(x) : \text{string}$
- Type context
 - set of type bindings $\text{id} : T$ (symbol table)

7

Typing rules

$\frac{\text{Premise}}{\text{Conclusion}} \text{ [Name]}$

$\frac{}{\text{Conclusion}} \text{ [Name]}$


8

Typing rules for expressions

$\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}} \text{ [+]}$

$\frac{}{E \vdash \text{true} : \text{bool}}$

$\frac{}{E \vdash \text{false} : \text{bool}}$

$\frac{}{E \vdash \text{int-literal} : \text{int}}$

$\frac{}{E \vdash \text{string-literal} : \text{string}}$

$\frac{}{E \vdash \text{null} : \text{null}}$

$\frac{}{E \vdash \text{new } T() : T}$

AST leaves

9

Some IC expression rules 1

$$\frac{}{E \vdash \text{true} : \text{bool}} \quad \frac{}{E \vdash \text{false} : \text{bool}}$$

$$\frac{}{E \vdash \text{int-literal} : \text{int}} \quad \frac{}{E \vdash \text{string-literal} : \text{string}}$$

$$\frac{E \vdash e1 : \text{int} \quad E \vdash e2 : \text{int}}{E \vdash e1 \text{ op } e2 : \text{int}} \quad \text{op} \in \{ +, -, /, *, \% \}$$

$$\frac{E \vdash e1 : \text{int} \quad E \vdash e2 : \text{int}}{E \vdash e1 \text{ rop } e2 : \text{bool}} \quad \text{rop} \in \{ <=, <, >, >= \}$$

$$\frac{E \vdash e1 : T \quad E \vdash e2 : T}{E \vdash e1 \text{ rop } e2 : \text{bool}} \quad \text{rop} \in \{ =, != \}$$

10

Some IC expression rules 2

$$\frac{E \vdash e1 : \text{bool} \quad E \vdash e2 : \text{bool}}{E \vdash e1 \text{ lop } e2 : \text{bool}} \quad \text{lop} \in \{ \&\&, \|\| \}$$

$$\frac{E \vdash e1 : \text{int}}{E \vdash -e1 : \text{int}} \quad \frac{E \vdash e1 : \text{bool}}{E \vdash !e1 : \text{bool}}$$

$$\frac{E \vdash e1 : T[]}{E \vdash e1.length : \text{int}} \quad \frac{E \vdash e1 : T[] \quad E \vdash e2 : \text{int}}{E \vdash e1[e2] : T} \quad \frac{E \vdash e1 : \text{int}}{E \vdash \text{new } T[e1] : T[]}$$

$$\frac{}{E \vdash \text{new } T() : T} \quad \frac{\text{id} : T \in E}{E \vdash \text{id} : T}$$

11

Type-checking algorithm

1. Construct types
 1. Add basic types to type table
 2. Traverse AST looking for user-defined types (classes, methods, arrays) and store in table
 3. Bind all symbols to types
2. Traverse AST bottom-up (using visitor)
 1. For each AST node find corresponding rule (there is only one for each kind of node)
 2. Check if rule holds
 1. **Yes:** assign type to node according to consequent
 2. **No:** report error

12

Algorithm example

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : \text{bool}}{E \vdash e_1 \text{ } \&\&\text{ } e_2 : \text{bool}}$$

$$\frac{E \vdash e_1 : \text{bool}}{E \vdash !e_1 : \text{bool}}$$

$$\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 \text{ } \>\text{ } e_2 : \text{bool}}$$

$$\frac{}{E \vdash \text{false} : \text{bool}}$$

$$\frac{}{E \vdash \text{int-literal} : \text{int}}$$

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : \text{bool}}{E \vdash e_1 \text{ } \&\&\text{ } e_2 : \text{bool}}$$

$$\frac{E \vdash e_1 : \text{bool}}{E \vdash !e_1 : \text{bool}}$$

$$\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 \text{ } \>\text{ } e_2 : \text{bool}}$$

$$\frac{}{E \vdash \text{false} : \text{bool}}$$

$$\frac{}{E \vdash \text{int-literal} : \text{int}}$$

45 > 32 && !false

13

Type-checking visitor

```

class TypeChecker implements PropagatingVisitor<Type, SymbolTable> {
    public Type visit(ArithBinopExp e, SymbolTable symtab) throws Exception {
        Type lType = e.left.accept(this, symtab);
        Type rType = e.right.accept(this, symtab);
        if (lType != TypeTable.intType())
            throw new TypeError("Expecting int type, found " +
                lType.toString(), e.getLine());
        if (rType != TypeTable.intType())
            throw new TypeError("Expecting int type, found " +
                rType.toString(), e.getLine());
        // we only get here if no exceptions were thrown
        e.type = TypeTable.intType();
    }
    ...
}
    
```

14

Statement rules

- Statements have type **void**
- Judgments of the form $E \vdash S$
 - In environment E , S is well-typed

$$\frac{E \vdash e : \text{bool} \quad E \vdash S}{E \vdash \text{while} (e) S}$$

$$\frac{E \vdash e : \text{bool} \quad E \vdash S}{E \vdash \text{if} (e) S}$$

$$\frac{E \vdash e : \text{bool} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash \text{if} (e) S_1 \text{ else } S_2}$$

$$\frac{}{E \vdash \text{break}}$$

$$\frac{}{E \vdash \text{continue}}$$

15

Checking return statements

- Special entry $\{\text{ret}:T_r\}$ represents return value
 - Add to symbol table when entering method
 - Lookup entry when hit return statement

$$\frac{E \vdash e:T \quad \text{ret}:T_r \in E \quad \text{Ⓢ}T_r}{E \vdash \text{return } e;}$$

T subtype of T'

$$\frac{\text{ret}:void \in E}{E \vdash \text{return};}$$

16

Subtyping

- Inheritance induces subtyping relation
 - Type hierarchy is a tree
 - Subtyping rules:

$$\frac{A \text{ extends } B \{...\}}{A \leq B} \quad \frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C} \quad \frac{}{\text{null} \leq A}$$

- Subtyping does not extend to array types
 - A subtype of B then A[] is not a subtype of B[]

17

Type checking with subtyping

- $S \leq T$
 - S may be used whenever T is expected
 - An Expression E from type S also has type T

$$\frac{E \vdash e : S \quad S \leq T}{E \vdash e : T}$$

18

IC rules with subtyping

$$\frac{E \vdash e1 : T1 \quad E \vdash e2 : T2 \quad T1 \leq T2 \text{ or } T2 \leq T1 \quad \text{op} \{=, \neq\}}{E \vdash e1 \text{ op } e2 : \text{bool}}$$

19

Semantic analysis flow

- Parsing and AST construction
 - Combine library AST with IC program AST
- Construct and initialize global type table
- Phase 1: Symbol table construction
 - Construct class hierarchy and check that hierarchy is a tree
 - Construct remaining symbol table hierarchy
 - Assign enclosing-scope for each AST node
- Phase 2: Scope checking
 - Resolve names
 - Check scope rules using symbol table
- Phase 3: Type checking
 - Assign type for each AST node
- Phase 4: Remaining semantic checks

20

Class hierarchy for types

```
abstract class Type {...}
class IntType extends Type {...}
class BoolType extends Type {...}
class ArrayType extends Type {
  Type elemType;
}
class MethodType extends Type {
  Type[] paramTypes;
  Type returnType;
  ...
}
class ClassType extends Type {
  ICClass classAST;
  ...
}
...
```

21

Type comparison

- Option 1: use a unique object for each distinct type
 - Resolve each type expression to same object
 - Use reference equality for comparison (==)
- Option 2: implement a method `t1.equals(t2)`
 - Perform deep (structural) test
- For object-oriented languages also need sub-typing:
`t1.subtypeOf(t2)`

22

Type table implementation

```
class TypeTable {
    // Maps element types to array types
    private Map<Type, ArrayType> uniqueArrayTypes;
    private Map<String, ClassType> uniqueClassTypes;

    public static Type boolType = new BoolType();
    public static Type intType = new IntType();
    ...

    // Returns unique array type object
    public static ArrayType arrayType(Type elemType) {
        if (uniqueArrayTypes.containsKey(elemType)) {
            // array type object already created - return it
            return uniqueArrayTypes.get(elemType);
        }
        else {
            // object doesn't exist - create and return it
            ArrayType arrt = new ArrayType(elemType);
            uniqueArrayTypes.put(elemType, ArrayType);
            return arrt;
        }
    }
    ...
}
```

23

Recap

Semantic analysis flow example

```

class A {
  int x;
  int f(int x) {
    boolean y; ...
  }
}

class B extends A {
  boolean y;
  int t;
}

class C {
  A o;
  int z;
}
    
```

25

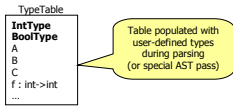
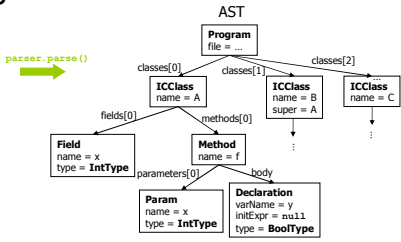
Parsing and AST construction

```

class A {
  int x;
  int f(int x) {
    boolean y; ...
  }
}

class B extends A {
  boolean y;
  int t;
}

class C {
  A o;
  int z;
}
    
```



26

Defined types and type table

```

class A {
  int x;
  int f(int x) {
    boolean y; ...
  }
}

class B extends A {
  boolean y;
  int t;
}

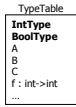
class C {
  A o;
  int z;
}
    
```

```

abstract class Type {
  String name;
  boolean subtypeof(Type t) {...}
}
class IntType extends Type {...}
class BoolType extends Type {...}
class ArrayType extends Type {
  Type elemType;
}
class MethodType extends Type {
  Type[] paramTypes;
  Type returnType;
}
class ClassType extends Type {
  ICClass classAST;
}
    
```

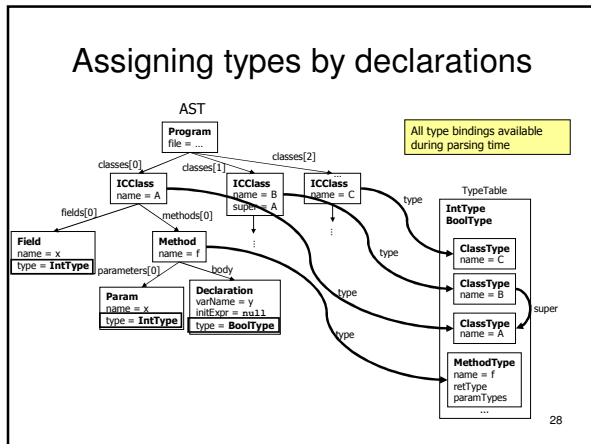
```

class TypeTable {
  public static Type boolType = new BoolType();
  public static Type intType = new IntType();
  ...
  public static ArrayType arrayType(Type elemType) {...}
  public static ClassType classType(String name, String super,
    ICClass ast) {...}
  public static MethodType methodType(String name, Type retType,
    Type[] paramTypes) {...}
}
    
```

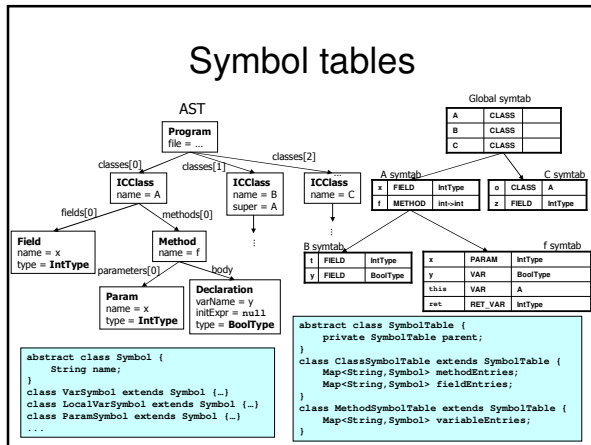


27

Assigning types by declarations



Symbol tables



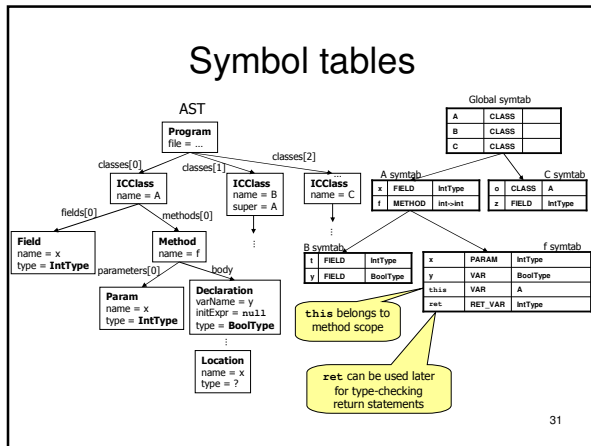
Scope nesting in IC

```

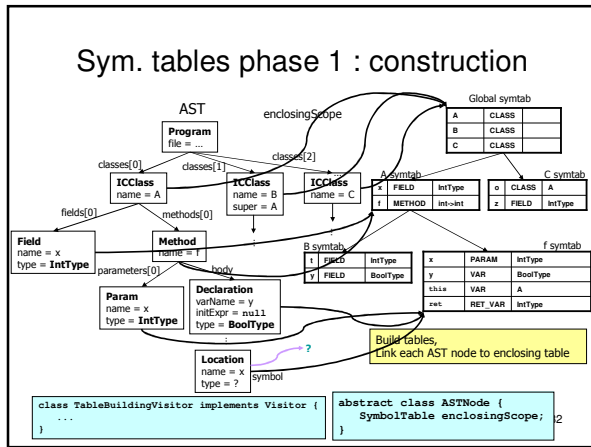
class GlobalSymbolTable extends SymbolTable {}
class ClassSymbolTable extends SymbolTable {}
class MethodSymbolTable extends SymbolTable {}
class BlockSymbolTable extends SymbolTable {}
    
```

Symbol	Kind	Type	Properties	
				Global names of all classes
				Class fields and methods
				Method formals + locals
				Block variables defined in block

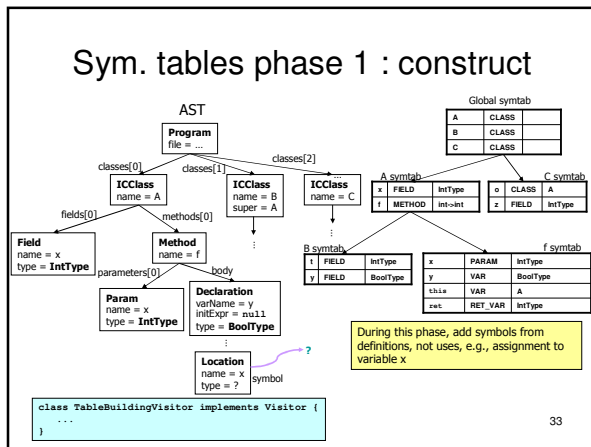
Symbol tables



Sym. tables phase 1 : construction



Sym. tables phase 1 : construct



```
public Type visit(While whileStatement) {
    Type conditionType =
        whileStatement.getCondition().accept(this);

    whileStatement.setType(PrimitiveTypes.VOID);
    if (conditionType != PrimitiveTypes.BOOLEAN)
        Error
        ++loopDepth;
    whileStatement.getOperation().accept(this);
    --loopDepth;
    return null;
}
```

37

```
public Type visit(Break breakStatement) {
    if (loopDepth == 0)
        error;
    setType(PrimitiveTypes.VOID);
    return null;
}
```

38
