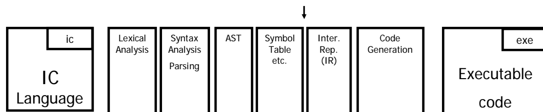
Winter 2006-2007 Compiler Construction T8 – semantic analysis recap + IR part 1

Mooly Sagiv and Roman Manevich
School of Computer Science
Tel-Aviv University

Announcements

- In PA3 use **`-print-ast`** option not **`-dump-ast`** (same option as in PA2)
- Compiler usage format:
  `java IC.Compiler <file.ic> [options]`
  - Check that there is a file argument, don't just crash
- Don't do semantic analysis in IC.cup
- Exception handling
  - Catch all exceptions you throw in main and handle them (don't re-throw)

Today



- Today:
  - Semantic analysis recap
  - Intermediate representation
    - HIR and LIR
    - Beginning IR lowering

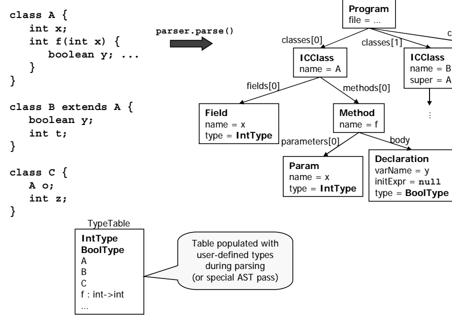
Semantic analysis flow example

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

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

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

4

Parsing and AST construction

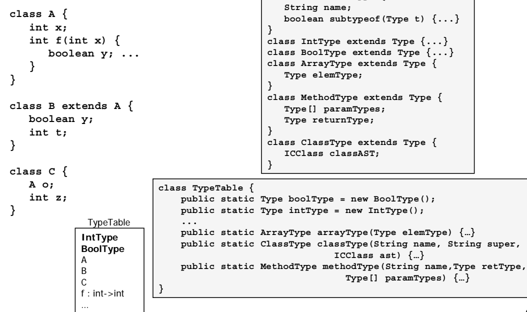
```
class A {
   int x;
   int f(int x) {
      boolean y; ...
   }
}

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

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



5

Defined types and type table

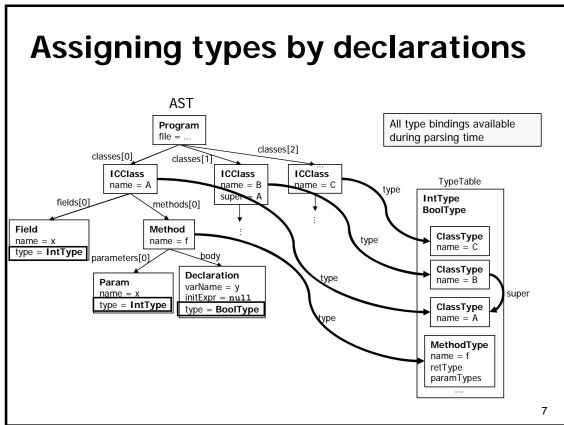
```
class A {
   int x;
   int f(int x) {
      boolean y; ...
   }
}

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

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



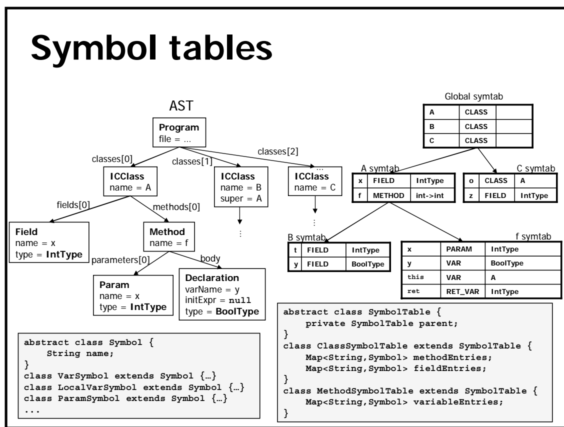

```
abstract class Type {
   String name;
   boolean subtypeof(Type t) {...}
}
class IntType extends Type {...}
class BoolType extends Type {...}
class ArrayType extends Type {
   Type elemType;
}
class MethodType extends Type {
   Type[] paramTypes;
   Type returnType;
}
class ClassType extends Type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

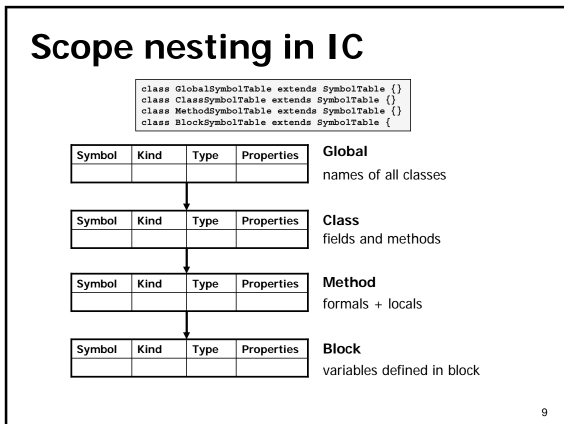
6



Assigning types by declarations
AST
All type bindings available during parsing time
Program
file = …
classes[0]
classes[1]
classes[2]
ICClass
name = A
ICClass
name = B
super = A
ICClass
name = C
fields[0]
methods[0]
Field
name = x
type = IntType
Method
name = f
parameters[0]
body
Param
name = x
type = IntType
Declaration
varName = y
initExpr = null
type = BoolType
type
TypeTable
IntType
BoolType
ClassType
name = C
ClassType
name = B
ClassType
name = A
super
MethodType
name = f
retType
paramTypes
7

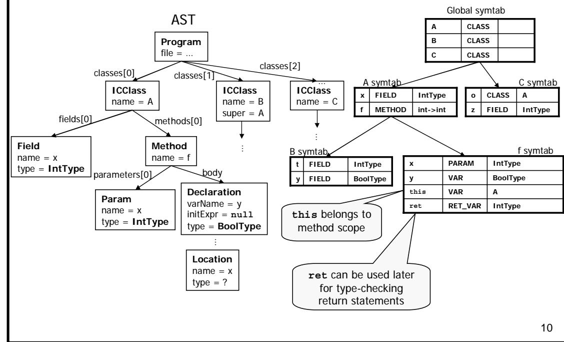


Symbol tables
AST
Program
file = …
classes[0]
classes[1]
classes[2]
ICClass
name = A
ICClass
name = B
super = A
ICClass
name = C
fields[0]
methods[0]
Field
name = x
type = IntType
Method
name = f
parameters[0]
body
Param
name = x
type = IntType
Declaration
varName = y
initExpr = null
type = BoolType
Global symtab
A CLASS
B CLASS
C CLASS
A symtab
x FIELD IntType
f METHOD int->int
C symtab
o CLASS A
z FIELD IntType
B symtab
t FIELD IntType
y FIELD BoolType
f symtab
x PARAM IntType
y VAR BoolType
this VAR A
ret RET_VAR IntType
abstract class Symbol {
String name;
}
class VarSymbol extends Symbol {…}
class LocalVarSymbol extends Symbol {…}
class ParamSymbol extends Symbol {…}
...
abstract class SymbolTable {
private SymbolTable parent;
}
class ClassSymbolTable extends SymbolTable {
Map<String,Symbol> methodEntries;
Map<String,Symbol> fieldEntries;
}
class MethodSymbolTable extends SymbolTable {
Map<String,Symbol> variableEntries;
}

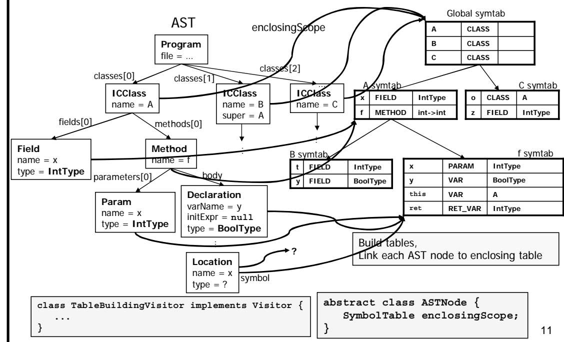


Scope nesting in IC
class GlobalSymbolTable extends SymbolTable {}
class ClassSymbolTable extends SymbolTable {}
class MethodSymbolTable extends SymbolTable {}
class BlockSymbolTable extends SymbolTable {
Symbol Kind Type Properties
Global
names of all classes
Symbol Kind Type Properties
Class
fields and methods
Symbol Kind Type Properties
Method
formals + locals
Symbol Kind Type Properties
Block
variables defined in block
9

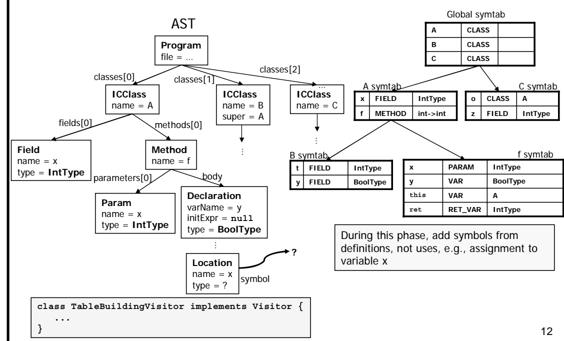
Symbol tables



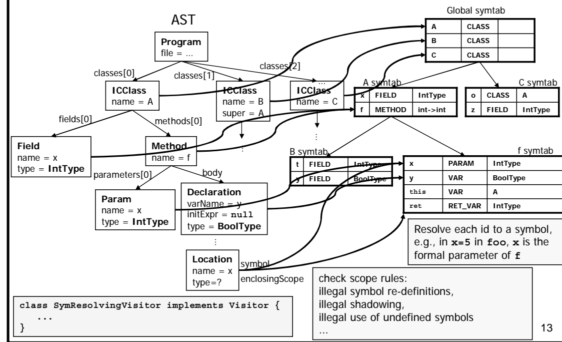
Sym. tables phase 1 : construction



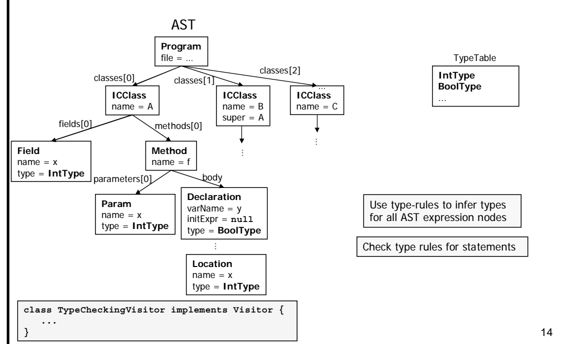
Sym. tables phase 1 : construct



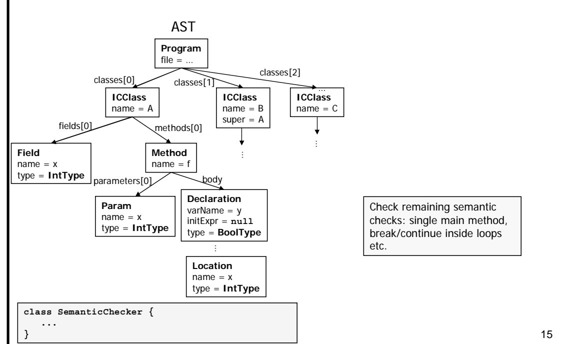
Sym. tables phase 2 : resolve



Type-check AST



Miscellaneous semantic checks



How to write PA3

1. Implement skeleton of type hierarchy + type table
   1. Modify IC.cup/Library.cup to use types
   2. Check result using `-print-ast` option
2. Implement `Symbol` classes and `SymbolTable` classes
3. Implement symbol table construction
   - Check using `-dump-symtab` option
4. Implement symbol resolution
5. Implement checks
   1. Scope rules
   2. Type-checks
   3. Remaining semantic rules

16

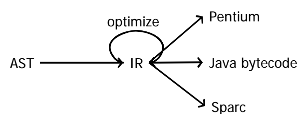
Class quiz

- Classify the following events according to compile time / runtime / other time
  (Specify exact compiler phase and information used by the compiler)
  1. `x` is declared twice in method `foo`
  2. Grammar G is ambiguous
  3. Attempt to dereference a null pointer `x`
  4. Number of arguments passed to `foo` is different from number of parameters
  5. reduce/reduce conflict between two rules
  6. Assignment to `a+5` is illegal since it is not an l-value
  7. The non-terminal X does not derive any finite string
  8. Test expression in an `if` statement is not Boolean
  9. `$r` is not a legal class name
  10. Size of the activation record for method `foo` is 40 bytes

17

Intermediate representation

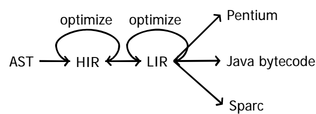
- Allows language-independent, machine independent optimizations and transformations
- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)



18

Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages



19

What's in an AST?

- Administration
  - Declarations
  - For example, class declarations
  - Many nodes do not generate code
- Expressions
  - Data manipulation
- Flow of control
  - If-then, while, switch
  - Target language (usually) more limited
    - Usually only jumps and conditional jumps

20

High-level IR (HIR)

- Close to AST representation
  - High-level language constructs
- Statement and expression nodes
  - Method bodies
  - Only program's computation
- Statement nodes
  - if nodes
  - while nodes
  - statement blocks
  - assignments
  - break, continue
  - method call and return

21

High-level IR (HIR)

- Expression nodes
  - unary and binary expressions
  - Array accesses
  - field accesses
  - variables
  - method calls
  - New constructor expressions
  - length-of expressions
  - Constants

In this project we can make do with HIR=AST

22

Low-level IR (LIR)

- An abstract machine language
  - Generic instruction set
  - Not specific to a particular machine
- Low-level language constructs
  - No looping structures, only jumps/conditional jumps
- We will use – two-operand instructions
  - Advantage – close to Intel assembly language
- Other alternatives
  - Three-address code: a = b OP c
    - Has at most three addresses (or fewer)
    - Also named quadruples: (a,b,c,OP)
  - Stack machine (Java bytecodes)

23

Arithmetic / logic instructions

- Abstract machine supports variety of operations

  a = b OP c   a = OP b
- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR
- Comparisons: EQ, NEQ, LEQ, GE, GEQ
- Unary operations: MINUS, NEG

24

Data movement

- Copy instruction: a = b
- Load/store instructions:
  a = *b   *a = b
- Address of instruction a=&b
  - Not used by IC
- Array accesses:
  a = b[i]   a[i] = b
- Field accesses:
  a = b.f   a.f = b

25

Branch instructions

- Label instruction
  label L
- Unconditional jump: go to statement after label L
  jump L
- Conditional jump: test condition variable a; if true, jump to label L
  cjump a L
- Alternative: two conditional jumps:
  tjump a L   fjump a L

26

Call instruction

- Supports call statements
  call f(a1,...,an)
- And function call assignments
  a = call f(a1,...,an)
- No explicit representation of argument passing, stack frame setup, etc.

27

Register machine instructions

Instruction	Meaning
Load_Const c, Rn	Rn = c
Load_Mem x, Rn	Rn = x
Store_Reg Rn, x	x = Rn
Add_Reg Rm, Rn	Rn = Rn + Rm
Subtr_Reg Rm, Rn	Rn = Rn – Rm
Mult_Reg Rm, Rn	Rn = Rn * Rm
…	

Note 1: rightmost operand = operation destination
Note 2: two register instr - second operand doubles as source and destination

28

Example

```
x = 42;
while (x > 0) {
  x = x - 1;
}
```

```
Load_Const 42,R1
Store_Mem R1,x
test_label:
Load_Mem x,R1
Compare_greater R1,0
False_Jump end_label
Load_Mem x,R1
Load_Const 1,R2
Subtr_Reg R1,R2
Store_Reg R2,x
Jump test_label
end_label:
```

(warning: code shown is a naive translation)

29

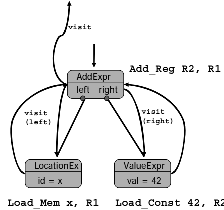
Translation (IR Lowering)

- How to translate HIR to LIR?
- Assuming HIR has AST form
  (ignore non-computation nodes)
  - Define how each HIR node is translated
  - Recursively translate HIR (HIR tree traversal)
- TR[e] = LIR translation of HIR construct e
  - A sequence of LIR instructions
  - Temporary variables = new locations
    - Use temporary variables to store intermediate values during translation

30

Translating expressions – Example

```
TR[x + 42]
```




```
Load_Mem x, R1
Load_Const 42, R2
Add_Reg R2, R1
```


Translating expressions

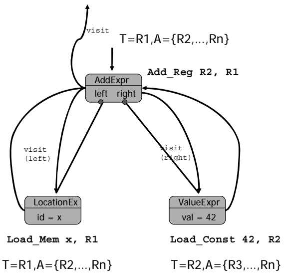
- (HIR) AST Visitor
  - Generate LIR sequence for each visited node
  - Propagating visitor – register information
- When visiting a expression node
  - A single Target register designated for storing result
  - A set of available auxiliary registers
  - TR[node, target, available set]
- Leaf nodes
  - Emit code using target register
  - No auxiliaries required
- What about internal nodes?

Translating expressions

- Internal nodes
  - Process first child, store result in target register
  - Process second child
    - Target is now occupied by first result
    - Allocate a new register Target2 from available set for result of second child
  - Apply node operation on Target and Target2
  - Store result in Target
  - All initially available register now available again
  - Result of internal node stored in Target (as expected)

Translating expressions – example

`TR[x + 42,T,A]`




```
Load_Mem x, R1
Load_Const 42, R2
Add_Reg R2, R1
```

34

See you next week

35