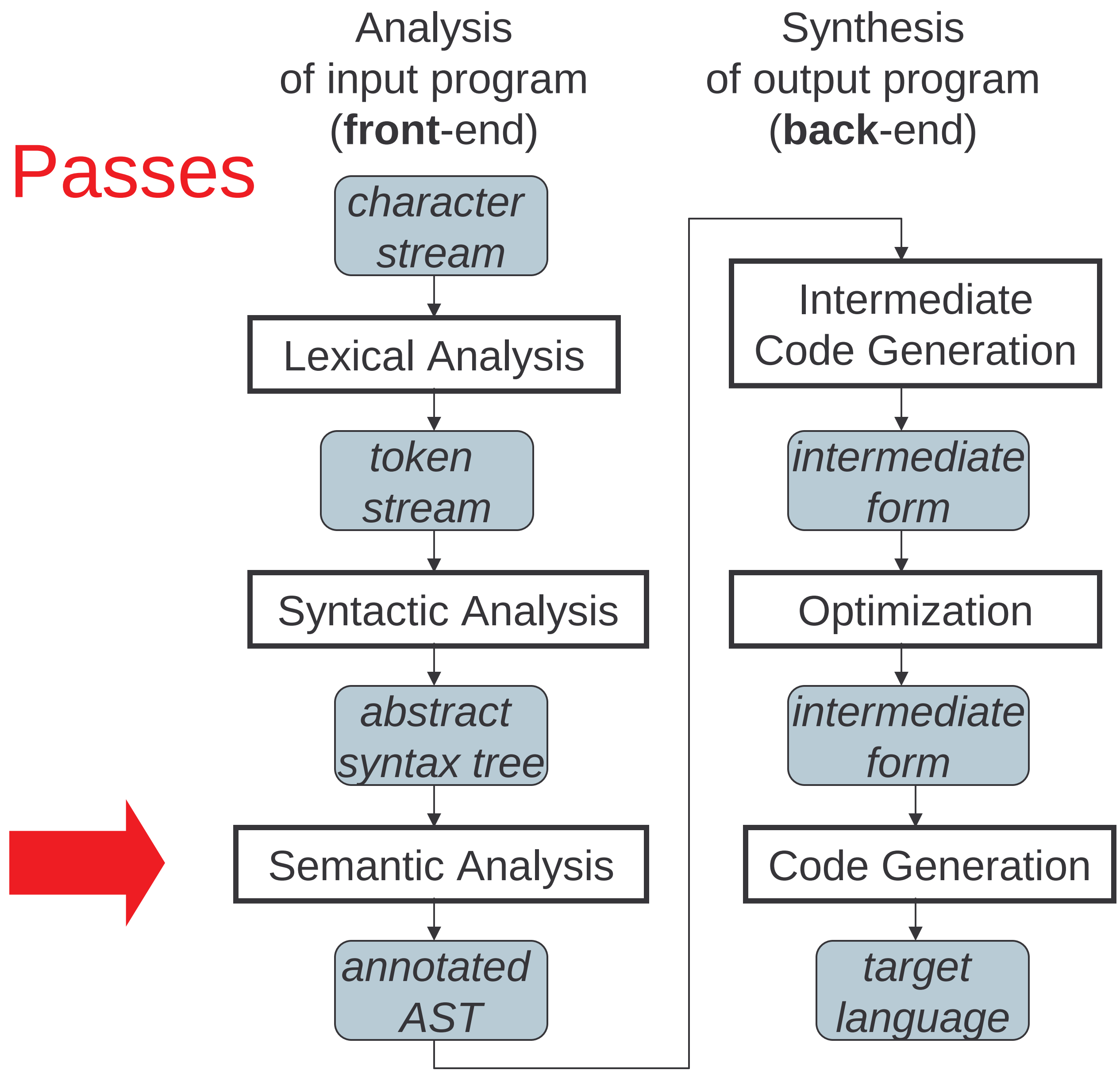
Semantic Analysis

Having figured out the program's structure, now figure out what it means

Compiler Passes



Semantic Analysis/Checking

Semantic analysis: the final part of the analysis half of compilation

- afterwards comes the synthesis half of compilation

Purposes:

- perform final checking of legality of input program, “missed” by lexical and syntactic checking
- name resolution, type checking, break stmt in loop, ...
- “understand” program well enough to do synthesis
- Typical goal: relate assignments to & references of particular variable

Symbol Tables

Key data structure during semantic analysis, code generation

Build in semantic pass

Stores info about the names used in program

- a map (table) from names to info about them
- each symbol table entry is a binding
- a declaration adds a binding to the map
- a use of a name looks up binding in the map
- report a type error if none found

An Example

```
class C {
   int x;
   boolean y;
   int f(C c) {
       int z;
       ...
       ...z...c...new C()...x...f(..)...
   }
}
```

A Bigger Example

```
class C {
   int x;
   boolean y;
   int f(C c) {
       int z;
       ...
       {
          boolean x;
          C z;
          int f;
          ...z...c...new C()...x...f(...)...
       }
     ...z...c...new C()...x...f(...)...
    }
 }
```

Nested Scopes

Can have same name declared in different scopes

- Want references to use closest textually-enclosing declaration
  - static/lexical scoping, block structure
  - closer declaration shadows declaration of enclosing scope

Simple solution:

- – one symbol table per scope
- – each scope's symbol table refers to its lexically enclosing scope's symbol table
- – root is the global scope's symbol table
- – look up declaration of name starting with nearest symbol table, proceed to enclosing symbol tables if not found locally

All scopes in program form a tree

Name Spaces

Sometimes we can have same name refer to different things, but still unambiguously. Example:

```
class F {
  int F(F F) {
     // 3 different F’s are available here!
     ... new F() ...
     ... F = ...
     ... this.F(...) ...
  }
}
```

In MiniJava: three name spaces

- classes, methods, and variables

- We always know which we mean for each name reference, based on its syntactic position

Simple solution: symbol table stores a separate map for each name space

Information About Names

- Different kinds of declarations store different information about their names
  - must store enough information to be able to check later references to the name
- A variable declaration:
    - its type
    - whether it's final, etc.
    - whether it's public, etc.
    - (maybe) whether it's a local variable, an instance variable, a global variable, or ...

Information About Names (Continued)

- A method declaration:
  - its argument and result types
  - whether it's static, etc.
  - whether it's public, etc.
- A class declaration:
  - its class variable declarations
  - its method and constructor declarations
  - its superclass

Generic Type Checking Algorithm

- To do semantic analysis & checking on a program, recursively type check each of the nodes in the program's AST,each in the context of the symbol table for its enclosing scope
  - going down, create any nested symbol tables & context needed
  - recursively type check child subtrees
  - on the way back up, check that the children are legal in the context of their parents
- Each AST node class defines its own type check method, which fills in the specifics of this recursive algorithm
- Generally:
  - declaration AST nodes add bindings to the current symbol table
  - statement AST nodes check their subtrees
  - expression AST nodes check their subtrees and return a result type

MiniJava Type Check Implementation

In the `SymbolTable` subdirectory:

Various `SymbolTable` classes, organized into a hierarchy:

```
SymbolTable
  GlobalSymbolTable
  NestedSymbolTable
    ClassSymbolTable
    CodeSymbolTable
```

- Support the following operations (and more):
  - `declareClass, lookupClass`
  - `declareInstanceVariable,`
    `declareLocalVariable,`
    `lookupVariable`
  - `declareMethod, lookupMethod`

Class, Variable and Method Information

- `lookupClass` returns a `ClassSymbolTable`
  - includes all the information about the class's interface
- `lookupVariable` returns a `VarInterface`
  - stores the variable's type
- A hierarchy of implementations:

```
VarInterface
  LocalVarInterface
  InstanceVarInterface
```

- `lookupMethod` returns a `MethodInterface`
  - stores the method's argument and result types

Key AST Type Check Operations

```
void Program.typecheck()
    throws TypecheckCompilerExn;
```

– typecheck the whole program

```
void Stmt.typecheck(CodeSymbolTable)
    throws TypecheckCompilerExn;
```

– Type check a statement in the context of the given symbol table

```
ResolvedType Expr.typecheck(CodeSymbolTable)
    throws TypecheckCompilerExn;
```

– type check an expression in the context of the given symbol table, returning the type of the result

Forward References

Typechecking class declarations is tricky: need to allow for forward references from the bodies of earlier classes to the declarations of later classes

```
class First {
   Second next;  // must allow this forward ref
   int f() {
   ... next.g() ...  // and this forward ref
   }
}
class Second {
   First prev;
   int g() {
      ... prev.f() ...
   }
}
```

Supporting Forward References

Simple solution:

type check a program's class declarations in multiple passes

- first pass: remember all class declarations

  {`First` --> class{?}, `Second` -->class{?}}

- second pass: compute interface to each class, checking class types in headers

  {`First` --> class{`next:Second`},

  `Second` -->class{`prev:First`}}

- third pass: check method bodies, given interfaces

Supporting Forward References [continued]

```
void
  ClassDecl.declareClass(GlobalSymbolTable)
      throws TypecheckCompilerExn;
```

- declare the class in the global symbol table

```
void ClassDecl.computeClassInterface()
      throws TypecheckCompilerExn;
```

- fill out the class's interface, given the declared classes

```
void ClassDecl.typecheckClass()
      throws TypecheckCompilerExn;
```

- type check the body of the class, given all classes' interfaces

Example Type Checking Operation

```
class VarDeclStmt {
  String name;
  Type type;

   void typecheck(CodeSymbolTable st)
      throws TypecheckCompilerExn {
  st.declareLocalVar(type.resolve(st), name);
  }
}
```

- `resolve` checks that a syntactic type expression is a legal type, and returns the corresponding resolved type
- `declareLocalVar` checks for duplicate variable declaration in this scope

Example Type Checking Operation

```
class AssignStmt {
   String lhs;
   Expr rhs;
   void typecheck(CodeSymbolTable st)
      throws TypecheckCompilerException {
     VarInterface lhs_iface = st.lookupVar(lhs);
     ResolvedType lhs_type = lhs_iface.getType();
     ResolvedType rhs_type = rhs.typecheck(st);
     rhs_type.checkIsAssignableTo(lhs_type);
     }
   }
```

`lookupVar` checks that the name is declared as a var

`checkIsAssignableTo` verifies that an expression yielding the rhs type can be assigned to a variable declared to be of lhs type

- initially, rhs type is equal to or a subclass of lhs type

Example Type Checking Operation

```
class IfStmt {
   Expr test;
   Stmt then_stmt;
   Stmt else_stmt;
   void typecheck(CodeSymbolTable st)
        throws TypecheckCompilerException {
     ResolvedType test_type = test.typecheck(st);
     test_type.checkIsBoolean();
     then_stmt.typecheck(st);
     else_stmt.typecheck(st);
   }
}
```

- `checkIsBoolean` checks that the type is a boolean

Example Type Checking Operation

```
class BlockStmt {
   List<Stmt> stmts;
   void typecheck(CodeSymbolTable st)
        throws TypecheckCompilerException {
      CodeSymbolTable nested_st =
          new CodeSymbolTable(st);
      foreach Stmt stmt in stmts {
          stmt.typecheck(nested_st); }
   }
}
```

- (Garbage collection will reclaim `nested_st` when done)

Example Type Checking Operation

```
class IntLiteralExpr extends Expr {
  int value;

  ResolvedType typecheck(CodeSymbolTable st)
       throws TypecheckCompilerException {
    return ResolvedType.intType();
  }
}
```

`ResolvedType.intType()` returns the resolved int type

Example Type Checking Operation

```
class VarExpr extends Expr {
  String name;

  ResolvedType typecheck(CodeSymbolTable st)
        throws TypecheckCompilerException {
     VarInterface iface = st.lookupVar(name);
     return iface.getType();
  }
}
```

Example Type Checking Operation

```
class AddExpr extends Expr {
   Expr arg1;
   Expr arg2;

   ResolvedType typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      ResolvedType arg1_type =
                   arg1.typecheck(st);
      ResolvedType arg2_type =
                   arg2.typecheck(st);
      arg1_type.checkIsInt();
      arg2_type.checkIsInt();
      return ResolvedType.intType();
    }
}
```

Polymorphism and Overloading

Some operations are defined on multiple types

Example: assignment statement: lhs = rhs;

- works over any lhs & rhs types, as long as they're compatible
- works the same way for all such types

Assignment is a **polymorphic** operation

Another example: equals expression: expr1 == expr2

- works if both exprs are ints or both are booleans (but nothing else, in MiniJava)
- compares integer values if both are ints, compares boolean values if both are booleans
- works differently for different argument types

Equality testing is an **overloaded** operation

Polymorphism and Overloading [continued]

- Full Java allows methods & constructors to be overloaded, too
    - different methods can have same name but different argument types
- Java 1.5 supports (parametric) polymorphism via generics: parameterized classes and methods

An Example Overloaded Type Check

```
class EqualExpr extends Expr {
   Expr arg1;
   Expr arg2;
   ResolvedType typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      ResolvedType arg1_type = arg1.typecheck(st);
      ResolvedType arg2_type = arg2.typecheck(st);
      if (arg1_type.isIntType() &&
          arg2_type.isIntType()) {
         //resolved overloading to int version
         return ResolvedType.booleanType();
      } else if (arg1_type.isBooleanType() &&
                 arg2_type.isBooleanType()) {
         //resolved overloading to boolean version
        return ResolvedType.booleanType();
      } else {
      throw new TypecheckCompilerException("bad
  overload");
}}}
```

Type Checking Extensions in Project [1]

Add resolved type for `double`

Add resolved type for arrays

- parameterized by element type

Questions:

- when are two array types equal?
- when is one a subtype of another?
- when is one assignable to another?

Add symbol table support for static class variable declarations

- `StaticVarInterface` class
- `declareStaticVariable` method

Type Checking Extensions in Project [2]

Implement type checking for new statements and expressions:

`IfStmt`

- else stmt is optional

`ForStmt`

- loop index variable must be declared to be an int
- initializer & increment expressions must be ints
- test expression must be a boolean

`BreakStmt`

- must be nested in a loop

`DoubleLiteralExpr`

- result is double

`OrExpr`

- like `AndExpr`

Type Checking Extensions in Project [3]

`ArrayAssignStmt`

- array expr must be an array
- index expr must be an int
- rhs expr must be assignable to array's element type

`ArrayLookupExpr`

- array expr must be an array
- index expr must be an int
- result is array's element type

`ArrayLengthExpr`

- array expr must be an array
- result is an int

`ArrayNewExpr`

- length expr must be an int
- element type must be a legal type
- result is array of given element type

Type Checking Extensions in Project [4]

Extend existing operations on ints to also work on doubles

Allow unary operations taking ints (`NegateExpr`) to be overloaded on doubles

Allow binary operations taking ints (`AddExpr, SubExpr, MulExpr, DivExpr, LessThanExpr,LessEqualExpr, GreaterEqualExpr, GreaterThanExpr,EqualExpr, NotEqualExpr`) to be overloaded on doubles

- also allow *mixed* arithmetic: if operator invoked on an int and a double, then implicitly coerce the int to a double and then use the double version

Extend `isAssignableTo` to allow ints to be assigned/passed/returned to doubles, via an implicit coercion

Type Checking Terminology

Static vs. dynamic typing

- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

Strong vs. weak typing

- strong: guarantees no illegal operations performed
- weak: can't make guarantees

Caveats:

- Hybrids common
- Mistaken usage also common
- "untyped," "typeless" could mean dynamic or weak

	static	dynamic
strong		
weak		

Type Equivalence

When is one type equal to another?

- implemented in MiniJava with `ResolvedType.equals(ResolvedType)` method

"Obvious" for atomic types like `int`, `boolean`, class types

What about type "constructors" like arrays?

```
int[] a1;
int[] a2;
int[][] a3;
boolean[] a4;
Rectangle[] a5;
Rectangle[] a6;
```

Type Equivalence

Parameterized types in Java 1.5:

```
List<int>l1; List<int>l2; List<List<int>>l3;
```

In C:

```
int* p1; int* p2;
struct {int x;} s1; struct {int x;} s2;
typedef struct {int x;} S; S s3; S s4;
```

Name vs Structural Equivalence

Name equivalence:

two types are equal iff they came from the same textual occurrence of a type constructor

- implement with pointer equality of ResolvedType instances
- special case: type synonyms (e.g. typedef) don’t define new types
- e.g. class types, struct types in C, datatypes in ML

Structural equivalence:

two types are equal iff they have same structure

- if atomic types, then obvious
- if type constructors:
  - same constructor
  - recursively, equivalent arguments to constructor
- implement with recursive implementation of equals, or by canonicalization of types when types created then use pointer equality
- e.g. atomic types, array types, record types in ML

Type Conversions and Coercions

- In Java, can **explicitly convert** an object of type double to one of type int
  - can represent as unary operator
  - typecheck, codegen normally
- In Java, can **implicitly coerce** an object of type int to one of type double
  - compiler must insert unary conversion operators, based on result of type checking

Type Casts

In C and Java, can explicitly **cast** an object of one type to another

- sometimes cast means a conversion (casts between numeric types)
- sometimes cast means just a change of static type without doing any computation (casts between pointer types or pointer and numeric types)

In C: safety/correctness of casts not checked

- allows writing low-level code that's type-unsafe
- more often used to work around limitations in C's static type system

In Java: downcasts from superclass to subclass include run-time type check to preserve type safety

- static typechecker allows the cast
- codegen introduces run-time check
- Java's main form of dynamic type checking

Programming language	static / dynamic	strong / weak	safety	Nominative / structural
Ada	static	strong	safe	nominative
assembly language	none	weak	unsafe	structural
BASIC	static	weak	safe	nominative
C	static	weak	unsafe	nominative
C++	static	strong	unsafe	nominative
C# [2]	static	strong	safe	nominative
Clipper	dynamic	weak	safe	duck
D	static	strong	unsafe	nominative
Fortran	static	strong	safe	nominative
Haskell	static	strong	safe	structural
Io	dynamic	strong	safe	duck
Java	static	strong	safe	nominative
Lisp	dynamic	strong	safe	structural
ML	static	strong	safe	structural
Objective-C [1]	dynamic	weak	safe	duck
Pascal	static	strong	safe	nominative
Perl 1-5	dynamic	weak	safe	nominative
Perl 6	hybrid	strong	safe	duck
PHP	dynamic	strong	safe	?
Pike	static	weak	safe	nominative