

CSE P 501 Su04

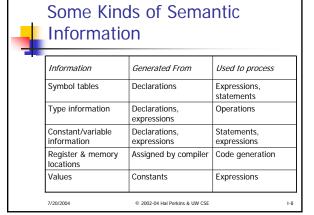


### Semantic Analysis

- Some key ideas
  - Extract types and other information from the program
  - Check language rules that go beyond the contextfree grammar
- Key data structures: symbol tables
  - For each identifier in the program, record its attributes (kind, type, etc.)
  - Later: assign storage locations (stack frame offsets) for variables

20/2004

© 2002-04 Hal Perkins & UW CSE





### Semantic Checks

- For each language construct we want to know
  - What semantic rules should be checked according to the language definition (type compatibility, etc.)
  - For an expression, what is its type (used to check whether the expression is used in the right context)
  - For declarations in particular, what information needs to be captured to be used elsewhere

7/20/2004

© 2002-04 Hal Perkins & UW CSE



# A Sampling of Semantic Checks (0)

- Name: id
  - id has been declared and is in scope
  - Inferred type of id is its declared type
  - Memory location assigned by compiler
- Constant: v
  - Inferred type and value are explicit

7/20/2004

© 2002-04 Hal Perkins & UW CSE



# A Sampling of Semantic Checks (1)

- Binary operator: exp<sub>1</sub> op exp<sub>2</sub>
  - exp<sub>1</sub> and exp<sub>2</sub> have compatible types
    - Identical, or
    - Well-defined conversion to a common type
  - Inferred type is a function of the operator and operands

7/20/2004

© 2002-04 Hal Perkins & UW CSE



# A Sampling of Semantic Checks (2)

- Assignment: exp<sub>1</sub> = exp<sub>2</sub>
  - exp<sub>1</sub> is assignable (not a constant or expression)
  - exp<sub>1</sub> and exp<sub>2</sub> have compatible types
    - Identical, or
    - exp<sub>2</sub> can be converted to exp<sub>1</sub> (e.g., char to int), or
    - Type of exp<sub>2</sub> is a subclass of type of exp<sub>1</sub> (can be decided at compile time)

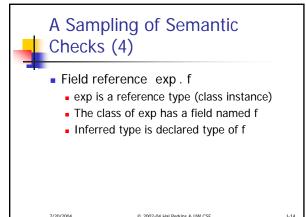
Inferred type is type of exp<sub>1</sub>

1-12

I-10

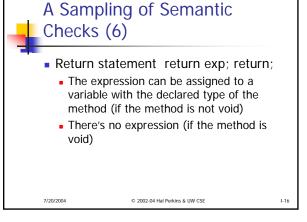
# A Sampling of Semantic Checks (3) Cast: (exp<sub>1</sub>) exp<sub>2</sub> exp<sub>1</sub> is a type exp<sub>2</sub> either Has same type as exp<sub>1</sub> Can be converted to type exp<sub>1</sub> (e.g., double to int) Is a superclass of exp<sub>1</sub> (in general requires a

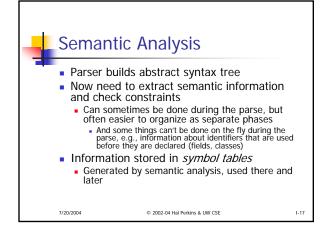
runtime check)
Inferred type is exp<sub>1</sub>

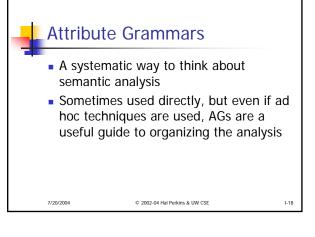


# A Sampling of Semantic Checks (5) • Method call exp.m(e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>) • exp is a reference type (class instance) • The class of exp has a method named m • The method has *n* parameters • Each argument has a type that can be assigned to the associated parameter • Inferred type is given by method declaration (or is void)

© 2002-04 Hal Perkins & UW CSE





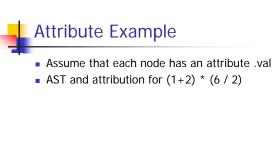




### **Attribute Grammars**

- Idea: associate attributes with each node in the (abstract) syntax tree
- Examples of attributes
  - Type information
  - Storage location
  - Assignable (e.g., expression vs variable, or Ivalue vs rvalue for C/C++ programmers)
  - Value (for constant expressions)
  - etc. ...
- Notation: X.a if a is an attribute of node X

20/2004 © 2002-04 Hal Perkins & UW CSE



2004 © 2002-04 Hal Perkins & UW CSE

## Inherited and Synthesized Attributes

- Given a production X ::= Y<sub>1</sub> Y<sub>2</sub> ... Y<sub>n</sub>
- A synthesized attribute is X.a is a function of some combination of attributes of Y<sub>i</sub>'s (bottom up)
- An inherited attribute Y<sub>i</sub>.b is a function of some combination of attributes X.a and other Y<sub>i</sub>.c (top down)

7/20/2004 © 2002-04 Hal Perkins & UW CSE



# Informal Example of Attribute Rules (1)

- Attributes for simple arithmetic language
- Grammar

program ::= decl stmt decl ::= int id; stmt ::= exp = exp; exp ::= id | exp + exp | 1

7/20/2004 © 2002-04 Hal Perkins & UW CSE

1-22



# Informal Example of Attribute Rules (2)

- Attributes
  - env (environment, e.g., symbol table); inherited by stmt, synthesized by decl
  - type (expression type); synthesized
  - kind (variable [Ivalue] vs value [rvalue]); synthesized

20/2004 © 2002-04 Hal Perkins & UW CSE

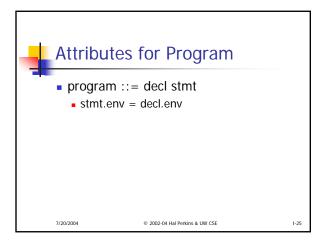


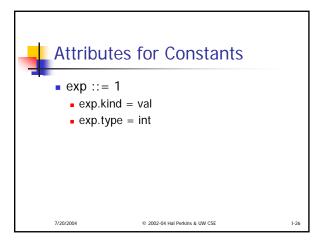
I-21

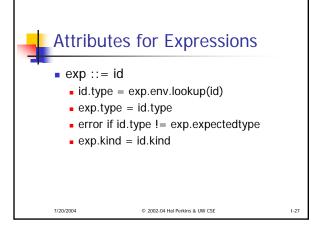
### Attributes for Declarations

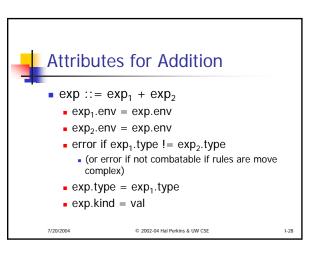
- decl ::= int id;
  - decl.env = {identifier, int, var}

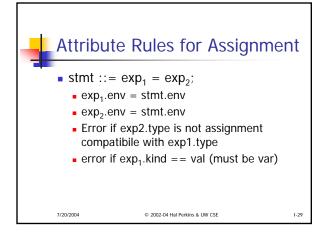
/20/2004 © 2002-04 Hall Perkins & UW CSE

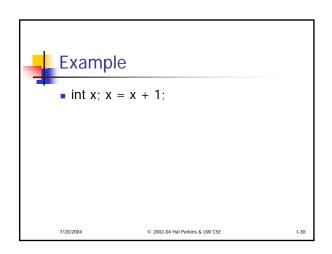














### **Extensions**

- This can be extended to handle sequences of declarations and statements
  - Sequence of declarations builds up combined environment with information about all declarations
  - Full environment is passed down to statements and expressions

7/20/2004

© 2002-04 Hal Perkins & LIW CSE



### **Observations**

- These are equational (functional) computations
- This can be automated, provided the attribute equations are non-circular
- Problems
  - Non-local computation
  - Can't afford to literally pass around copies of large, aggregate structures like environments (i.e., copy rules)

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### In Practice

- Attribute grammars give us a good way of thinking about how to structure semantic checks
- Symbol tables will hold environment information
- Add fields to AST nodes to refer to appropriate attributes (symbol table entries for identifiers, expression types, etc.)
  - Put in appropriate places in inheritance tree statements don't need types, for example

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### Symbol Tables

- Map identifiers to <type, kind, location, other properties>
- Operations
  - Lookup(id) => information
  - Enter(id, information)
  - Open/close scopes

7/20/2004

© 2002-04 Hal Perkins & UW CSE



# Aside: Implementing Symbol Tables

- Topic in classical compiler course: implementing a hashed symbol table
- These days: use the Java collection classes (or equivalent in C#, C++, etc.)
  - Map (HashMap) will solve most of the problems
  - List (ArrayList) for ordered lists (parameters, etc.)

7/20/2004

© 2002-04 Hal Perkins & UW CSE



1-33

### Symbol Tables for MiniJava (1)

- Global
  - Single global table to map class names to per-class symbol tables
    - Created in a pass over class definitions in AST
    - Used in remaining parts of compiler to check field/method names and extract information about them

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### Symbol Tables for MiniJava (2)

- Global
  - 1 Symbol table for each class
    - 1 entry for each method/field declared in the class
      - Contents: type information, public/private, storage locations (later), etc.
    - In full Java, multiple symbol tables per class since methods and fields can have the same names in a class

7/20/2004

© 2002-04 Hal Perkins & UW CSF



### Symbol Tables for MiniJava (3)

- Global (cont)
  - All global tables persist throughout the compilation
    - And beyond in a real Java or C# compiler...
      - (e.g., symbolic information in Java .class files)

7/20/2004

© 2002-04 Hal Perkins & LIW CSF



### Symbol Tables for MiniJava (4)

- Local symbol table for each method
  - 1 entry for each local variable or parameter
    - Contents: type information, storage locations (later), etc.
  - Needed only while compiling the method; can discard when done

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### **Beyond MiniJava**

- What we aren't dealing with: nested scopes
  - Inner classes
  - Nested scopes in methods reuse of identifiers in parallel or (if allowed) inner scopes
- Basic idea: new symbol table for inner scopes, linked to surrounding scope's table
  - Look for identifier in inner scope; if not found look in surrounding scope (recursively)

1-40

• Pop back up on scope exit

7/20/2004 © 2002-04 Hal Perkins & UW CSE



### **Engineering Issues**

- In practice, want to retain O(1) lookup
  - Use hash tables with additional information to get the scope nesting right
- In multipass compilers, symbol table information needs to persist after analysis of inner scopes for use on later passes
  - See a good compiler textbook for details

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### **Error Recovery**

- What to do when an undeclared identifier is encountered?
  - Only complain once (Why?)
  - Can forge a symbol table entry for it once you've complained so it will be found in the future
  - Assign the forged entry a type of "unknown"
  - "Unknown" is the type of all malformed expressions and is compatible with all other types to avoid redundant error messages

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### "Predefined" Things

- Many languages have some "predefined" items
- Include initialization code in the compiler to create symbol table entries for these
  - Rest of compiler generally doesn't need to know the difference between "predeclared" items and ones found in the program

7/20/2004 © 2002-04 Hal Perkins & UW CSE



### Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char
- Compound/Constructed Types
  - Built up from other types (recursively)
  - Constructors include arrays, records/ structs/classes, pointers, enumerations, functions, modules, ...

7/20/2004 © 2002-04 Hal Perkins & UW CSE I-44



### Type Representation

- Create a shallow class hierarchy
   abstract class Type { ... } // or interface
   class ClassType extends Type { ... }
   class PrimType extends Type { ... }
  - Should not need too many of these

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### **Base Types**

- For each base type (int, boolean, others in other languages), create a single object to represent it
  - Symbol table entries and AST nodes for expressions refer to these to represent type info
- Useful to create a "void" type to tag functions that do not return a value
- Also useful to create an "unknown" type for errors

7/20/2004 © 2002-04 Hal Perkins & UW CSE



### Compound Types

- Basic idea: represent with an object that refers to component types
  - Limited number of these correspond directly to type constructors in the language (record/struct, array, function,...)

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### Class Types

class Id { fields and methods } class ClassType extends Type {

Type baseClassType; // ref to base class
Map fields; // type info for fields
Map methods; // type info for methods

}

 (Note: may not want to do this literally, depending on how you chose to represent symbol tables for classes. i.e., class symbol tables might be useful as the representation of the class type.)

7/20/2004

© 2002-04 Hal Perkins & UW CSE



### **Array Types**

For Java this is simple: only possibility is # of dimensions and element type

```
class ArrayType extends Type {
  int nDims;
  Type elementType;
}
```

7/20/2004 © 2002-04 Hal Perkins & UW CSE



### Array Types for Pascal

- Pascal allows arrays to be indexed by any discrete type
  - array[indexType] of elementType
- Element type can be any other type, including an array

```
class PascalArrayType extends Type {
    Type indexType;
    Type elementType;
}
```

0/2004 © 2002-04 Hal Perkins & UW CSE



### Methods/Functions

Type of a method is its result type plus an ordered list of parameter types
 class MethodType extends Type {
 Type resultType; // type or "void"
 List parameterTypes;
 }

7/20/2004 © 2002-04 Hal Perkins & UW CSE



### Type Equivalance

- For base types this is simple
  - Types are the same if they are identical
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically or when requested by programmer (casts)

1-52

7/20/2004 © 2002-04 Hal Perkins & UW CSE



# Type Equivalence for Compound Types

- Two basic strategies
  - Structural equivalence: two types are the same if they are the same kind of type and their component types are equivalent, recursively
  - Name equivalence: two types are the same only if they have the same name, even if their structures match
- Different language design philosophies

7/20/2004

© 2002-04 Hal Perkins & UW CSE



I-51

# Type Equivalence and Inheritance

Suppose we have

class Base { ... }
class Extended extends Base { ... }

- A variable declared with type Base has a compile-time type of Base
- During execution, that variable may refer to an object of class Base or any of its subclasses like Extended (or can be null, which is compatible with all class types)
  - Sometimes called the runtime type

20/2004 © 2002-04 Hal Perkins & UW CSE



### **Useful Compiler Functions**

- Create a handful of methods to decide different kinds of type compatibility
  - Types are identical
  - Type t1 is assignment compatibile with t2
  - Parameter list is compatible with types of expressions in the call
- Normal modularity reasons: isolates these decisions in one place and hides the actual type representation from the rest of the compiler

7/20/2004

© 2002-04 Hal Perkins & UW CSE



# Implementing Type Checking for MiniJava

- Create multiple visitors for the AST
- First passe(s): gather information
  - Collect global type information for classes
  - Could do this in one pass, or might want to do one pass to collect class information, then a second one to collect per-class information about fields, methods
- Next set of passes: go through method bodies to check types, other semantic constraints

/2004 © 2002-04 Hal Perkins & UW CSE



### **Coming Attractions**

- Need to start thinking about translating to object code (actually x86 assembly language for this project)
- Next: x86 overview/review
- Then
  - Runtime representation of classes, objects, data, and method stack frames
  - Assembly language code for higher-level language statements

7/20/2004

© 2002-04 Hal Perkins & UW CSE

I Hal Perkins & UW CSE

I-57