


CSE 582 – Compilers

Static Semantics

Hal Perkins
Autumn 2002


10/29/2002 © 2002 Hal Perkins & UW CSE 1-1



Agenda

- n Static semantics
- n Types
- n Attribute grammars
- n Representing types
- n Symbol tables
 - n Reminder about Java container classes
 - n "Predefined" things

10/29/2002 © 2002 Hal Perkins & UW CSE 1-2



What do we need to know to compile this?


```

class C {
    int a;
    C(int initial) {
        a = initial;
    }
    void setA(int val) {
        a = val;
    }
}

class Main {
    public static void main(){
        C c = new C(17);
        c.setA(42);
    }
}

```


10/29/2002 © 2002 Hal Perkins & UW CSE 1-3



Beyond Syntax

- n There is a level of correctness that is not captured by a grammar
 - n Has a variable been declared?
 - n Are types consistent in an expression?
 - n In the assignment $x=y$, is y assignable to x ?
 - n Does a method call have the right number and types of parameters?
 - n In a selector $p.q$, is q a method or field of class p ?
 - n Is variable x guaranteed to be initialized before it is used?
 - n Etc. etc. etc.


10/29/2002 © 2002 Hal Perkins & UW CSE 1-4



What else do we need to know to generate code?

- n Where are fields allocated in an object?
- n How big are objects?
- n Where are local variables stored when a method is called?
- n Which methods are associated with an object/class?

10/29/2002 © 2002 Hal Perkins & UW CSE 1-5



Types

- n Role of types in programming languages
 - n Run-time safety
 - n Compile-time error detection
 - n Improved expressiveness (operator overloading, for example)

10/29/2002 © 2002 Hal Perkins & UW CSE 1-6

Semantic Analysis

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-7

Some Kinds of Semantic Information

Information	Generated From	Used to process
Symbol tables	Declarations	Expressions, statements
Type information	Declarations, expressions	Operations
Constant/variable information	Declarations, expressions	Statements, expressions
Register & memory locations	Assigned by compiler	Code generation
Values	Constants	expressions

10/29/2002 © 2002 Hal Perkins & UW CSE 1-8

A Sampling of Semantic Checks (0)

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-9

A Sampling of Semantic Checks (1)

- Binary operator: $exp_1 \text{ op } exp_2$
 - exp_1 and exp_2 have compatible types
 - Identical, or
 - Well-defined conversion to a common type
 - Result type is a function of the operator and operands

10/29/2002 © 2002 Hal Perkins & UW CSE 1-10

A Sampling of Semantic Checks (2)

- Assignment: $exp_1 = exp_2$
 - exp_1 is assignable (not a constant or expression)
 - exp_1 and exp_2 have compatible types
 - Identical, or
 - exp_2 can be converted to exp_1 (e.g., char to int), or
 - Type of exp_2 is a subclass of type of exp_1 (can be decided at compile time)
 - Result type is type of exp_1
 - Location where value is stored is assigned by the compiler

10/29/2002 © 2002 Hal Perkins & UW CSE 1-11

A Sampling of Semantic Checks (3)

- Cast: $(exp_1) exp_2$
 - exp_1 is a type
 - exp_2 either
 - Has same type as exp_1
 - Can be converted to type exp_1 (e.g., double to int)
 - Is a subclass of exp_1 (usually requires a runtime check)
 - Result type is exp_1

10/29/2002 © 2002 Hal Perkins & UW CSE 1-12

A Sampling of Semantic Checks (4)

- Field reference $exp_1.exp_2$
 - The type of exp_1 is a class
 - exp_2 is an identifier
 - exp_1 has a field or method named exp_2
 - Result type is declared type of exp_2

10/29/2002

© 2002 Hal Perkins & UW CSE

1-13

A Sampling of Semantic Checks (5)

- Method call $m(e_1, e_2, \dots, e_n)$
 - The method has n parameters
 - Each argument has a type that can be assigned to the associated parameter
 - Result type is given by method declaration (or is void)

10/29/2002

© 2002 Hal Perkins & UW CSE

1-14

Semantic Analysis

- Parser builds abstract syntax tree
- Now need to extract semantic information and check constraints
 - Can sometimes be done during the parse, but often easier to organize as a separate phase
 - And some things can't be done on the fly during the parse, e.g., information about identifiers that are used before they are declared (fields, classes)
- Information stored in *symbol tables*
 - Generated by semantic analysis, used later

10/29/2002

© 2002 Hal Perkins & UW CSE

1-15

Attribute Grammars

- A systematic way to think about semantic analysis
- Sometimes used directly, but even if ad hoc techniques are used, AGs are a useful guide to organizing the analysis

10/29/2002

© 2002 Hal Perkins & UW CSE

1-16

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 lvalue vs rvalue for C/C++ programmers)
 - Value (for constant expressions)
 - etc. ...
- Notation: $X.a$ if a is an attribute of X

10/29/2002

© 2002 Hal Perkins & UW CSE

1-17

Attribute Example

- Assume that each node has an attribute *.val*
- AST and attribution for $(1+2) * (6 / 2)$

10/29/2002

© 2002 Hal Perkins & UW CSE

1-18

Inherited and Synthesized Attributes

- n Given a production $X ::= Y_1 Y_2 \dots Y_n$
- n A *synthesized* attribute is $X.a$ is a function of some combination of attributes of Y_i 's (bottom up)
- n An *inherited* attribute $Y_i.b$ is a function of some combination of attributes $X.a$ and other $Y_j.c$ (top down)

10/29/2002

© 2002 Hal Perkins & UW CSE

1-19

Informal Example of Attribute Rules (1)

- n Attributes for simple arithmetic language
- n Grammar
 - program ::= decl stmt
 - decl ::= int id;
 - stmt ::= exp = exp ;
 - exp ::= id | exp + exp | 1

10/29/2002

© 2002 Hal Perkins & UW CSE

1-20

Informal Example of Attribute Rules (2)

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

10/29/2002

© 2002 Hal Perkins & UW CSE

1-21

Attributes for Declarations

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

10/29/2002

© 2002 Hal Perkins & UW CSE

1-22

Attributes for Program

- n program ::= decl stmt
 - n stmt.env = decl.env

10/29/2002

© 2002 Hal Perkins & UW CSE

1-23

Attributes for Constants

- n exp ::= 1
 - n exp.kind = val
 - n exp.type = int

10/29/2002

© 2002 Hal Perkins & UW CSE

1-24

Attributes for Expressions

- $\text{exp} ::= \text{id}$
 - $\text{id.type} = \text{exp.env.lookup}(\text{id})$
 - $\text{exp.type} = \text{id.type}$
 - error if $\text{id.type} \neq \text{exp.expectedtype}$
 - $\text{exp.kind} = \text{id.kind}$

10/29/2002

© 2002 Hal Perkins & UW CSE

1-25

Attributes for Addition

- $\text{exp} ::= \text{exp}_1 + \text{exp}_2$
 - $\text{exp}_1.\text{env} = \text{exp.env}$
 - $\text{exp}_2.\text{env} = \text{exp.env}$
 - $\text{exp}_1.\text{expectedtype} = \text{exp.expectedtype}$
 - $\text{exp}_2.\text{expectedtype} = \text{exp.expectedtype}$
 - error if $\text{exp}_1.\text{type} \neq \text{exp}_2.\text{type}$
 - $\text{exp.type} = \text{exp}_1.\text{type}$
 - $\text{exp.kind} = \text{val}$

10/29/2002

© 2002 Hal Perkins & UW CSE

1-26

Attribute Rules for Assignment

- $\text{stmt} ::= \text{exp}_1 = \text{exp}_2;$
 - $\text{exp}_1.\text{env} = \text{stmt.env}$
 - $\text{exp}_2.\text{env} = \text{stmt.env}$
 - $\text{exp}_2.\text{expectedtype} = \text{exp}_1.\text{type}$
 - error if $\text{exp}_1.\text{kind} \neq \text{var}$

10/29/2002

© 2002 Hal Perkins & UW CSE

1-27

Example

- $\text{int } x; x = x + 1;$

10/29/2002

© 2002 Hal Perkins & UW CSE

1-28

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

10/29/2002

© 2002 Hal Perkins & UW CSE

1-29


Observations

- These are equational (functional) computations
- In principle, this could 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)

10/29/2002

© 2002 Hal Perkins & UW CSE


1-30



In Practice

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


10/29/2002 © 2002 Hal Perkins & UW CSE 1-31



Symbol Tables

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


10/29/2002 © 2002 Hal Perkins & UW CSE 1-32



Aside: Implementing Symbol Tables in Java

- Classic topic in 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.)


10/29/2002 © 2002 Hal Perkins & UW CSE 1-33



Symbol Tables for JFlat (1)

- Global
 - 1 Symbol table per class
 - 1 entry for each method/field
 - 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


10/29/2002 © 2002 Hal Perkins & UW CSE 1-34



Symbol Tables for JFlat (2)

- Global (cont)
 - Single global table to map class names to class symbol tables
 - Created in pass over class definitions
 - Used in remaining parts of compiler to check field/method names and extract information
 - All global tables persist throughout the compilation
 - And beyond in a real Java or C# compiler...

10/29/2002 © 2002 Hal Perkins & UW CSE 1-35



Symbol Tables for JFlat (3)

- 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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-36

Symbol Tables Beyond JFlat

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-37

Engineering Issues

- n In practice, want to retain $O(1)$ lookup
 - n Use hash tables with additional information to get the scope nesting right
- n In multipass compilers, symbol table information needs to persist after analysis of inner scopes

10/29/2002 © 2002 Hal Perkins & UW CSE 1-38

Error Recovery

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-39

"Predefined" Things

- n JFlat, like all other languages has some "predefined" items
 - n Class JFSystem, in our case
- n Include startup code in the compiler to create symbol table entries for these
 - n Rest of compiler generally doesn't need to know the difference between "predeclared" items and ones found in the program

10/29/2002 © 2002 Hal Perkins & UW CSE 1-40

Type Systems

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-41

Type Representation

- n Create a shallow class hierarchy


```
abstract class Type { ... }
class ClassType extends Type { ... }
```

 - n Should not need too many of these

10/29/2002 © 2002 Hal Perkins & UW CSE 1-42

Base Types

- For each base type (int, boolean, others in other languages), create a single object to represent
 - Use references to these objects to represent these types
- Useful to create a "void" type to tag functions that do not return a value

10/29/2002 © 2002 Hal Perkins & UW CSE 1-43

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,...)

10/29/2002 © 2002 Hal Perkins & UW CSE 1-44

Class Types

- ```

class Id { fields and methods }
class ClassType extends Type {
 Type parentClassType; // ref to base class
 Set fields; // type info for fields
 Set methods; // type info for methods
}

```

  - (Note: may not want to do this literally, depending on how you chose to represent symbol tables for classes.)

10/29/2002 © 2002 Hal Perkins & UW CSE 1-45

## Array Types

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

```

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

```

10/29/2002 © 2002 Hal Perkins & UW CSE 1-46

## 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;
}

```

10/29/2002 © 2002 Hal Perkins & UW CSE 1-47

## Functions/Methods

- Type of a function is its result type plus an ordered list of parameter types
 

```

class MethodType extends Type {
 Type resultType; // type or "void"
 List parameterTypes;
}

```

10/29/2002 © 2002 Hal Perkins & UW CSE 1-48

## Type Equivalence

- 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)

10/29/2002 © 2002 Hal Perkins & UW CSE 1-49

## 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 subtypes 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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-50

## 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*

10/29/2002 © 2002 Hal Perkins & UW CSE 1-51

## Useful Compiler Functions

- You will want methods like this in the objects representing types
 

```
// return true if this type is assignable to
// type other, otherwise false
boolean assignableTo(Type other) { ... }
```
- Other useful methods might be ones that report whether one type is the same as another

10/29/2002 © 2002 Hal Perkins & UW CSE 1-52

## Coming Attractions

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

10/29/2002 © 2002 Hal Perkins & UW CSE 1-53