Compiler Construction

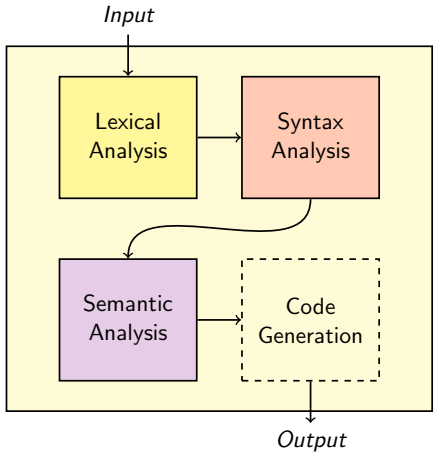
Semantic analysis

Emma Söderberg, Lennart Andersson

Revision 2012-02-08

2012

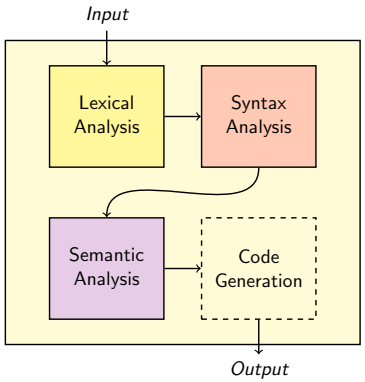
Today



Semantic Analysis

- Name Analysis
- Type Analysis
- Error Checking

Error Checking



Lexical:

- Unknown tokens?, ..

Syntax

- Missing tokens?, Order? ..

Semantic:

- Are all identifiers declared?
- Are all expressions of a legal type?
- Does this procedure call have the correct number of arguments?
- ...

Semantic Analysis

se-man-tic: *of or relating to meaning in language*

Webster's Dictionary

Name Analysis

- Bind identifiers to the appropriate declaration

Type Analysis

- Compute types of expressions

Errors?

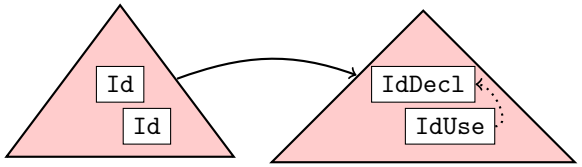
A:
```
int b = a;
int a = 0;
```

B:
```
2 < 4 < 3
```

C:
```
void m(int a) {
  int a = 0;
}
```


Name Analysis

Bind each `IdUse` to the `IdDecl` to which it refers.



Previously: One AST class for identifiers
Now: Two different AST classes for identifiers:

- `IdDecl` – a declared occurrence
  an identifier that names a declaration
- `IdUse` – an applied occurrence
  an identifier that refers to a declaration

Which identifiers are `IdDecl`s? Which are `IdUse`s?

```
class GraphicalObject {
  Position pos;
  Position getPos() { return pos; }
}

class Circle extends GraphicalObject {
  float radius;
  float area() {
    return Math.PI*radius*radius;
  }
}
```


Scope (Synlighetsområde)

How do we know which `IdDecl` an `IdUse` refers to?

```
int a = 4;
{
  int a = 3;
  print(a);
}
```

```
class A { int a; }
class B extends A {
  void m() {
    print(a);
    int a = 0;
  }
}
```

```
class A {
  void m() {}
}
class B extends A {
  void m2() {
    m();
  }
  void m() {}
}
```

The scope of a declaration: The part of a program where the name of a declaration is visible

Blocks

Block:

- a syntactic unit with declarations and statements
- may require memory allocation during execution (once or several times)

Block structure (nesting):

- a block can have inner blocks (recursively)
- declarations in a block are visible also in the inner blocks

Anonymous

```
void m() {
  {
  }
}
```

Class in method

```
void m() {
  class A {
  }
}
```

Method in method

```
void m() {
  void n() {
  }
}
```


Special "blocks"

```
"{"
  class A {
    static int a = B.b;
  }
  class B {
    static int b = A.a;
  }
"}"
```

Global declarations

- can be viewed as belonging to an outermost block

Static fields

- A global block can be created for each class to hold its static fields

Scope rules (visibility rules)

Govern how `IdUses` are bound to `IdDecls`

Typical factors (differ in different languages):

combination how can blocks be combined?

name collisions what happens if the same name is declared in many blocks?

declaration order does it affect the bindings?

method overloading can there be several methods of the same name, but with different argument types? What are the binding rules?

parameters how do they relate to local variables?

return values are they named explicitly?

visibility restrictions private, public, ...

qualified access access via another name

How can blocks be combined?

Block structure

- declarations in an outer block are visible also in an inner block

Inheritance

- declarations in a class are visible also in subclasses

Combined block structure and inheritance

- e.g., a method in a subclass can access instance variables in a superclass

```
{
  int a = 0;
  {
    int b = a;
  }
}
```

```
class A {
  void m() {}
}
class B extends A {
  void n() { m() }
}
```

```
class A {
  int a = 0;
}
class B extends A {
  void m() {
    int b = a;
  }
}
```

Name collisions

```
{
  int x;
  {
    int x;
  }
}
```

Shadowing (skuggning)
— inner declarations shadow outer declarations of the same name

Forbidden shadowing
Some languages prohibit inner blocks to declare a name that is already present in an outer block.

Declaration order

```
class A {
  void m() {
    n();
  }
  void n() {
    m();
  }
}
```

Homogeneous blocks
— the order between the declarations is irrelevant

- Declarations in Java classes
- All declarations in Algol

```
void m() {
  int a = 0;
  int b = a;
}
```

Declare-before-use
— a name must be declared before it is used

- Declarations in Java methods
- Declarations in C
- Declarations in Pascal

Overloaded method names

Overloaded method

- The same method name
- Different signatures (method signature — name, parameter types, return type)
- Bind to the method with the most specific signature (with respect to the static types)

Which method is called?

```
void visit(Expr node) { ... }
void visit(Add node) { ... }

Expr e = new Add();
visit(e);
```

Which method is called?

```
e.accept(visitor);
```

Parameters

Usually, parameters can be seen as special local variables

```
void m(int x, y) {
  int s = 2;
  x = s + y;
  ...
}
```

Usually, it is an error to declare a local variable with the same name

```
void m(int x, y) {
  int x; // Multiple declaration of x
  ...
}
```


Return value

A value can be returned by a special `return` statement (C, Java, ...)

```
int m() { return 3; }
```

A value can be returned by using the function name as a variable (Algol, Pascal, ...)

```
int m() { m = 3; }
```

A value can be returned by begin the value of the last expression (Scala, ...)

```
int m() { 3+7; }
```


Visibility modifiers

```
class A {
  private int a;
  protected int b;
}
class B extends A {
  int a = b;
}
```

Explicit modifiers

- private, protected, public (Java)
- hidden, protected (Simula)
- friend, ... (C++)
- ...

What rules hold for classes/methods/fields without modifiers?

Qualified access

```
class A {
  B m() { .. }
}
class B {
  C c = .. ;
}
class C {
  int d = 0;
}
```

Indirect access via another name

```
A a = new A();
int e = a.m().c.d
```

Other name issues

Can the same name be used for declarations of different kind?

- E.g., fields and methods in Java: `int c`, `int c() ...`
- what about `class c ...`?

Do all names share the same lexical definition?
or are there different kinds of identifiers?

- In Smalltalk, attributes must start with a lower case letter, classes with an upper case letter.
- In Java it is just a convention that class names should start with a capital letter.

If this is Java, what is wrong?

```
class c {
  int c = 0;
  void c(int c) {
  }
}
```


AST based name analysis

Bindings

- Represent the binding as a reference variable in the IdUses

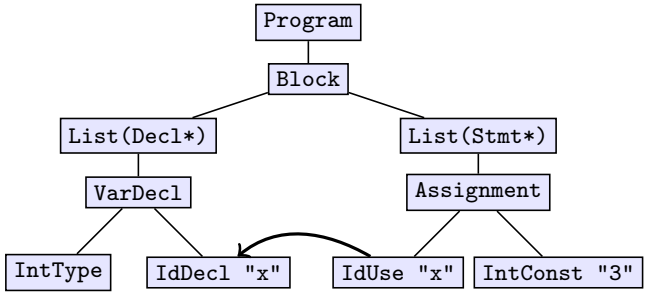
Algorithm

- Traverse the AST
- Keep track of visible declarations in a symbol table
- Look up the declaration for each IdUse

Symbol table

- maps names to declarations
- a stack can be used to handle block structure

Bindnings



Distinguish between identifier uses and declarations in the AST
Replace `IdExpr` by

- `IdDecl` — declared identifier occurrence
- `IdUse` — applied identifier occurrence
  (has a reference to an `IdDecl` node)

Example

```
Program ::= Block;
Block ::= Decl* Stmt*;
abstract Decl;
VarDecl: Decl ::= Type IdDecl;
abstract Type;
IntType: Type ::=;
IdDecl ::= <ID>;
abstract Stmt;
AssignStmt: Stmt ::= IdUse Expr;
BlockStmt: Stmt ::= Block;
abstract Expr;
IdUse: Expr ::= <ID>;
IntConst: Expr ::= <INT>;
Add: Expr ::= Left:Expr Right:Expr;
```

```
begin
  int x;
  x = 3;
end;

begin
  int x;
  int y;
  x = 3;
  begin
    int z;
    z = x + 5;
    y = z + 1;
  end;
end;
```


Properties of this simple language

- declarations appear before statements (cannot be mixed like in C or Java)
- No IdUses inside the declaration part
- Blocks can be nested

```
begin
  int x;
  x = 3;
end;

begin
  int x;
  int y;
  x = 3;
  begin
    int z;
    z = x + 5;
    y = z + 1;
  end;
end;
```


Implementation of name analysis ...

```
aspect NameAnalysis {
  void Program.nameAnalysis() {
    SymbolTable table = new SymbolTable();
    nameAnalysis(table);
  }

  void ASTNode.nameAnalysis(SymbolTable table) {
    for (k=0; k<getNumChild(); k++) {
    getChild(k).nameAnalysis(table);
  }

  void Block.nameAnalysis(SymbolTable table) {
    table.enterblock();
    getDeclList().addDecls(table);
    getStmtList().nameAnalysis(table);
    table.exitblock();
  }
```


. . . Implementation of name analysis

```
  void ASTNode.addDecls(SymbolTable table) {
    for (k=0; k<getNumChild(); k++) {
      getChild(k).addDecls(table);
    }
  }

  void IdDecl.addDecls(SymbolTable table) {
    table.add(getID(), this);
  }

  IdDecl IdUse.decl;

  void IdUse.nameAnalysis(SymbolTable table) {
    decl = table.lookup(getID());
  }
}
```

Implementation of the symbol table

- a stack of hashmaps (one hashmap for each block)
- a new hashmap is pushed when entering a block
- a hashmap is popped when exiting a block
- the symbol table is empty after the name analysis

Symbol table API

Constructor summary

`SymbolTable()`

Method summary

`void`	`add(String symbol, Object meaning)`	Adds the symbol and its associated meaning to the top table
`boolean`	`alreadyDeclared(String symbol)`	Returns true if the symbol is already in the top table
`int`	`blockLevel()`	Returns the current block level (= the number of dictionaries on the stack)
`void`	`enterBlock()`	Adds a new table to the stack
`void`	`exitBlock()`	Removes the top table from the stack
`Object`	`lookup(String symbol)`	Returns the meaning of symbol

Implementation of SymbolTable ...

```
public class SymbolTable {
  private class LinkedHashMap
                  extends HashMap<String, Object>() {

    LinkedHashMap next;
    LinkedHashMap(LinkedHashMap next) {
      this.next = next;
    }

    Object lookup(String symbol) {
      Object result = get(symbol);
      if (result != null || next == null)
        return result;
      else return next.lookup(symbol);
    }
  }
}
```


... Implementation of SymbolTable

```
public class SymbolTable {
    private LinkedHashMap top = null;
    public void add(String symbol, Object meaning) {
        top.put(symbol, meaning);
    }
    public Object lookup(String symbol) {
        return top.lookup(symbol);
    }
    public void enterBlock() {
        top = new LinkedHashMap(top);
    }
    public void exitBlock() {
        top = top.next;
    }
}
```

Variations

The declaration part could contain IdUses, e.g.,

- variables with initial values
- names of user defined types (e.g., classes and interfaces)

The declaration part could contain blocks, e.g.,

- methods, inner classes

Mixed declarations and statements

- as in C and Java

Order between the declarations

- Decl-before-use like in C and Java methods

Efficient handling of symbols

Use object comparison

```
s1 == s2
```

instead of string comparison

```
s1.equals(s2)
```

How?

- maintain a pool of unique `String` objects
- make sure all identifiers are represented by one of these `String` objects

There is support for this in Java

```
class String {
  String intern()
  ...
}
```


Example

The JJT specification

```
void IdDecl() #IdDecl: { Token t; }
{
  t = <ID>
  { jjtThis.setID(t.image.intern()); }
}

void IdUse() #IdUse: { Token t; }
{
  t = <ID>
  { jjtThis.setID(t.image.intern()); }
}
```

Now, `SymbolTable.lookup` can use object comparisons!

Use of symbol tables

Simplistic 1-pass compiler

- no tree is built — the name analysis is performed during parsing
- the *meaning* of a declaration is represented by a small data structure. For a variable it typically contains type and allocation info.
- the name analysis is tangled with other compilation aspects, e.g., type analysis, allocation information, code generation
- complex scope rules require complex declaration structures and several passes

Use of symbol tables

AST-based use

- The *meaning* is represented by AST nodes
- Easy to traverse the tree in any order and multiple times if needed, handling complex scope rules
- After the `IdUse`⟶`IdDecl` bindings are computed, the symbol table is no longer needed.
- Other computations, e.g., type analysis, use the bindings.
- The name analysis can easily be separated from other analyses

Type analysis

- Compute the type for each expression
- Check that each expression has correct type in its context

Representation of simple types

A constant object for each simple type

```
interface SemType {
  public static final SemType unknownType = ...;
  public static final SemType intType = ...;
  public static final SemType boolType = ...;
}
```

`unknownType` is used for undeclared variables and erroneous expressions.

Structured types

How can user-defined types (classes, records, arrays ...) be represented?
Use the AST

- Use the corresponding AST node, e.g., the `ClassDecl` node or the `RecordDecl` node to represent the type
- Let these AST nodes implement the `SemType` interface

Recursive types

Structured types can be recursive, e.g.,

```
class A {
  B b;
}
class B {
  A a;
}
```

Representation

- The recursion is represented as a circular structure: the AST for one class contains a node that has a reference to the AST for the other class, and vice versa.
- Avoid infinite loops in type computations.

Subtypes

Like in OO languages, e.g.,

```
class A {
  ...
}
class B extends A {
  ...
}
...
A a = new B(); // type correct
B b = new A(); // incorrect
```

Representation

- The type hierarchy can be represented in the AST. The node for a class can have a reference to the node for its superclass.

Computation of types for expressions

```
aspect ExprTypes {
  abstract SemType Expr.type();

  SemType Add.type() {
    return SemType.intType;
  }

  SemType IdUse.type() {
    if (decl == null) {       // undeclared identifier
      return SemType.unknownType;
    } else {
      return decl.type();
    }
  }
}
```


How can an `IdDecl` node know its type?

Alternative 1

- Extend the name analysis so that a type attribute is added to each `IdDecl` node

Alternative 2 (easiest)

- The `IdDecl` node could *ask* its parent node

Alternative 3

- Bindings could be represented as references to `Decl` nodes instead of as references to `IdDecl` nodes.
- The `Decl` node can more easily return a suitable `SemType`.
- But for declaration lists (`int a, b, c`) this solution does not work.

Implementation of "Ask the parent"

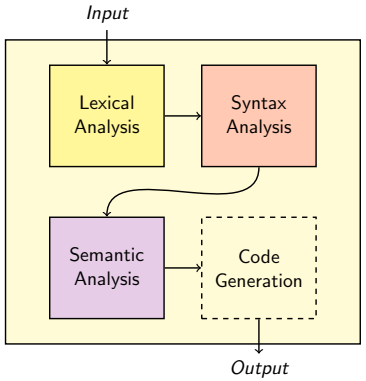
```
aspect IdDeclType {
  SemType IdDecl.type() {
    return getParent().type();
  }

  SemType ASTNode.type() { error(); return null; }

  SemType VarDecl.type() {
    return getType().type();
  }

  SemType IntType.type() {
    return SemType.intType;
  }
}
```


Examples of errors



Lexical errors

```
@, $, 44.8, ...
```

Syntactic errors

```
if { ( true ) }
```

Semantic errors

```
int a = true;
```

Run-time errors

```
int a = 10/0;
```


Semantic error checking

Report errors that are discovered during semantic analysis

- variables that are not declared
- more than one declaration of the same name in the same block
- erroneous type in an expression
- wrong number of arguments to a method call

Semantic error checking

Avoid reporting errors that depend on other errors

```
int b = a + 5;
```

an undeclared variable should not also give type checking errors

```
boolean a = true;
int b = (a * 10) + 5;
```

a type checking error in an expression should not give additional type checking errors in enclosing expressions

Semantic error checking

Accurate error messages
A good compiler shows the location of the error in the source code.

JavaCC/JJTree
JJTree can save tokens with row and column attributes for each node. Have a look in SourceCode.jadd and Parser.jjt in CalcAnalysis.zip for lab 5.

Class for error handling

Example

```
class CompileTimeErrors {
  void add(ASTNode node, String message)
  // node is the AST node closest to the error
  // message is an appropriate error message
  { ...
  }
  void sort() { ... } // according to row, col
  void print(Stream s) {
    ...
  }
}
```


Type checking Add, Sub, Mul, ...?

Several AST classes with the same structure and the same type checking and error handling. Use inheritance to capture the common behavior:

```
abstract Expr;
abstract BinExpr: Expr ::= Left:Expr Right:Expr;
abstract BinIntExpr: BinExpr;
Add: BinIntExpr;
Sub: BinIntExpr;
Mul: BinIntExpr;
Div: BinIntExpr;
```

Implement type checking in `BinIntExpr`

Example: type checking

```
aspect TypeChecking {
  void ASTNode.typeCheck(CompileTimeErrors errs) {
    for (int k=0; k < getNumChild(); k++) {
      getChild(k).typeCheck(errs);
    }
  }
  void BinIntExp.typeCheck(CompileTimeErrors errs) {
    SemType t1 = getLeft().type();
    SemType t2 = getRight().type();
    if ( t1 != SemType.intType &&
         t1 != SemType.unknownType ||
         t2 != SemType.intType &&
         t2 != SemType.unknownType) {
           errs.add(this, "non integer argument
              to binary integer operator");
    }
  }
}
```

Summary

Name analysis

- Results in `IdUse`→`IdDecl` bindings
- Many variants on scope rules in different languages

Type analysis and representation of types

- simple types (int, float, ...)
- structured recursive types (objects, records, ...)
- hierarchical types (inheritance)

Error checking

- many different kinds of errors