## Semantic Analysis

- determination of "meaning" of syntactically valid program
- two main sub-phases:
  - scope analysis
    - determining definition for each use of a name
  - type analysis
    - determining type of entity used/expected in each context





## Scope Analysis

- names identify different entities:
  - classes
  - variables
  - methods
  - parameters
- some entities are predefined (e.g. standard types), some are declared
- scope rules:
  - for each use of a name (reference) there is a unique declaration that defines that reference





## Block Structured Languages

- in Algol-like languages declarations and the statements over which they have effect are grouped into blocks.
- blocks include:
  - program
  - procedures
  - block statement (e.g. begin end)
- blocks may be nested
- purpose of block is to confine the usage of the declared entity to the statements within the block

## Scope Rules (Block Structure)

- all identifiers declared in the same block must have unique names
  - variations: within kinds (classes) of names
- where is an entity known?
  - entire block?
  - roughly from declaration to end of block ("defined before use")
  - from where within declaration?
    - from the end of its declaration to the end of the block
    - a procedure declared in a block is known from the end of its header to the end of the declarative block
- hole-in-scope
  - entities declared in different blocks may have the same name
  - consider a block `Q` which defines an entity named `x`. If `Q` contains a block `R` that defines a another entity named `x`, the first entity (i.e. declared in `Q`) is not known in the scope of the second entity (i.e. declared in `R`)
  - qualification



```
begin
   :
  …x…
   :
  int x;
   :
  …x…
   :
  static final int y = y;
   :
end
```

```
begin
   :
procedure p ( )
  begin
    :
    p();
    :
  end;
    :
end
```

```
Q: begin
     :
     int x;
     :
     …x…
     :
   R: begin
        :
        float x;
        :
        …x…
        :
      end;
      :
      …x…
      :
   end
```

## Stack Symbol Table

- these rules suggest an algorithm (binding algorithm) which determines, for each use of a name, the declaration to which it refers (i.e. its binding)

```
loop
    search for declaration of name in current block
exit when found
    move to smallest encompassing block
end
```

- scope analysis done by maintaining a stack of blocks
- at block entry, new block pushed on stack
- at block exit, block is popped from stack
- block for predefined entities is pushed onto stack at initialization
- when a declaration occurs, entry representing it put in current block
  - before adding entry to stack, check that it is not already declared in current block (found ⇒ multiple declaration error)
- when a reference is encountered, stack is searched, block by block, from top down until an entry is found (not found ⇒ undeclared)





## Java-like Languages

- basically block-structured
  - classes contain blocks
- each compilation is part of a larger unit
  - must save some of symbol table after compilation
  - mechanism for deciding which saved symbol tables to use during compilation
    - packages and import
- overriding vs overloading vs hiding
  - overloading and overriding for instance methods
    - same signature implies override, different signature implies overload
  - hiding for class methods and all variables
    - hole-in-scope
- in Java
  - doesn't require "defined before use"
  - classes, constructors, methods and block-statements define blocks which may be nested
  - different kinds of identifiers and methods with different signatures may be overloaded within a block





## Java Scope Rules

1. Look for a declaration of the name in the bock in which the code resides. If one exists, this is the defining declaration. This rule applies to both formal parameter declarations and local declarations.
2. If no such declaration exists, apply step 1 again, looking in the immediately enclosing block. Continue until there is no enclosing block.
3. If no such declaration exists, apply step 1 in the superclass of the class. Continue until there is no superclass.
4. If no such declaration exists, check the public declarations of public classes from all imported packages (including `java.lang`)
5. If no such declaration exists, the name is undeclared and the reference is in error.

## Symbol Table Representation

- each scope (block or class) contains entries for items declared in the scope
- algorithms
  - locate a name starting in this scope (`find`)
  - check to see if name already defined in this scope (`alreadyDefined`)
  - define name in this scope (`define`)
  - locate superclass (`getSuperclass`)
- interface `ADeclaration` & class `Declarations`
  - things that can be declared
- interface `Scope`
  - implementations of specific scope rules
- class symbol tables are saved on disk
- nested scopes have reference to enclosing scope (stack symbol table)
- every syntactic unit is processed within some scope





## Global Scope

- global scope
  - represents system (package)
  - predefined names
  - declared classes
- representation is a `Map`
  - i.e. key (name) maps to data (declarative info)
- initialization
  - add entries for all predefined names (including classes)
- class `Global`
  - loading scope
    - `global` in `Compiler`
    - create if not defined
  - saving scope
    - write `global` object & re-read next time
      - only when no errors
- new classes stored in `global`





## Scope Errors

- name already defined in current block when defining then multiple declaration error (except classes in global scope)
- name not defined when searching then undeclared name error
- misspelling can lead to multiple errors, e.g.:

```
int  tabel;
  :
…table…
  :
…table…
```

- error recovery:
  - when undeclared name encountered, add it to symbol table in current scope
    - undeclared superclass, treat as no superclass
  - since global only written when no error, effect is temporary
- missing name, e.g.:

```
int  ;
```

  - special name (`Compiler.missingName`) but don't define

## Scope Analysis

- extend parser for scope analysis
- `SyntacticUnit` references `scope` for unit
  - additional parameter on constructor
  - used in looking up names
  - `expectId`
    - replaces `expect(IDENTIFIER,…)` and returns name
    - returns `Compiler.missingName` when no id found
  - `checkAndDefine`
    - duplicate declarations
    - handles missing names
  - `findDcl`
    - undeclared identifiers
    - declares in scope
  - `semanticError`
    - scope & type errors





## Examples

- `VarDcl`
- `Body`
- `Name`





## Kind and Type Analysis

- various kinds of declared entities
  - class, method, variable
- declarations define types for entities
  - type of variable
  - return type of method
  - classes define a type
- kind checking
  - correct kind for context
- type checking
  - the required (*a priori*) type (from context)
  - the actual (*a posteriori*) type (from expression)
  - comparison of *a priori* and *a posteriori* for type checking and/or coercion
    - type equivalence
    - type (assignment) compatibility

## Kinds of Entities

- entities belong to categories (kinds)
- different kinds have different attributes
  - variables: type
  - methods: return type, parameters
- have interfaces defining each kind of entity
- implementations as subclasses of `Declarations`
  - symbol table entries





## Kind Interfaces

- `AType`
  - type equivalence
  - type compatibility
- `AVariable`
  - type
- `AnArray`
  - element type
- `AMethod`
  - function/procedure
  - result type
  - parameter list
  - signature match
- `AConstructor`
  - essentially a method
- `AClass`
  - superclass
  - constructor

## Error Recovery

- undeclared identifiers
- missing names
- kind errors
- `Universals`
  - implements all kind interfaces
    - multiple inheritance
  - provides a scope
  - `Compiler.universalDcl`
  - substitutes where kind/type missing or incorrect





## Kind Checking

- correct kind of identifier for context
- `find` returns `ADeclaration`
  - usually need to recover kind
- `SyntacticUnit` kind helper methods
  - convert to correct kind
  - generate kind errors and do error recovery
  - e.g.
    - `asClassDcl`





## Type Checking

- implementation classes for each kind of declaration
- for declarations
  - record attributes using object of appropriate kind
- for references
  - find declaration (scope checking)
  - verify kind (kind checking)
  - check type (type compatibility)

## Types

- `Types`
  - abstract
  - default type equivalence and compatibility
- type equivalence
  - structural equivalence
  - name equivalence





## Primitive Types

- `Primitives`
- understood by compiler, i.e. no further attributes
- as `AType`
  - equivalence
  - compatibility
    - conversion





## Variables

- 3 kinds of variables
  - instance variables
  - formal parameters
  - local variables
  - scope
  - parameters part of parameter list
- attributes
  - type
- `Variables`

## Arrays

- attributes?
  - element type
  - dimensionality
  - number of elements
    - dimension bounds
- as `ADeclaration`
  - anonymous type (i.e. no name)
- as `AType`
  - equality
  - compatibility
    - which attributes?
- as `Scope`
  - `length` attribute
  - no superclass





## Classes

- attributes?
  - superclass
  - constructor (not known ad declaration time)
  - members (filled in as defined)
- as `AType`
  - equality
  - compatibility (`this := t`)
    - same class or superclass of other class
- as `Scope`
  - `find`
    - members first
    - if has superclass, superclass next
    - no superclass, check globals





## Methods

- attributes?
  - encompassing scope (class)
  - result type (if function)
  - parameter list
  - local variables
- signature conformance
  - parameter list one-to-one type equality
  - type equivalent result type
- as `Scope`
  - superclass is encompassing class's superclass
  - `alreadyDefined`
    - locals or parameter list
  - `find`
    - locals or parameter list first
    - then encompassing scope
- a constructor is a procedure method

## Parameter Lists

- attributes?
  - enclosing scope
    - method's enclosing scope
  - ordered list of declarations (variables)
    - `Iterable-iterator`
  - is empty parameter list OK?
- as `Scope`
  - superclass is same as for method
  - `find`
    - search list first
    - then enclosing scope
  - `alreadyDefined`
    - in param list
  - `define`
    - in parameter list in order





## Error Recovery

- `Universals`
- implements all kinds
- as `Scope`
  - no superclass
  - nothing already defined
  - everything defined as universal
  - `define` doesn't do anything
- as `AType`
  - equals all types
  - compatible with all types
- as `AVariable`
  - type is `universal`





- as `AnArray`
  - element type is `universal`
- as `AMethod`
  - both a function and a procedure
  - result type is `universal`
  - dummy parameter list
- as `AClass`
  - constructor not set
  - returns dummy constructor

## Dummy Parameter List

- extends `Parameters`
- as `Scope`
  - everything defined as `universal`
  - nothing already defined
  - define doesn't do anything
- as `Parameters`
  - OK to have no parameters
  - `iterator` returns as many universals as needed
    - `DummyIterator`





## Type Checking

- identifier declarations
  - create appropriate kind of object as implementation of `ADeclaration`
  - define in correct scope
- identifier references
  - look up in appropriate scope (use `findDcl`)
  - check kind (use `asXxxx`)
  - extract attributes as required for type checking
- type identifiers
  - `expectTypeId` in `SyntacticUnit`
    - replaces `expectId` where type identifier expected
    - returns type declaration
- examples
  - `VarDcl`
  - `Name`
  - `Qualifier`





## Expressions

- type checking
  - correct type of operands for operator
  - both operands of same type (usually)
  - result is type of expression
  - when type mismatch: error and result is `universal`
- predefined types
  - `Compiler.typeXxxx` is reference to type declaration in `global`
    - must define/establish when loading globals
  - literals
    - type is from literal type
- subscripts must be `typeInt`
  - some languages is attribute of array type
- example
  - `Term`

## Statements

- type checking on expressions in statements
  - e.g. `if`, `while`:
    - boolean expression
    - use `Compiler` constants
  - assignment: type compatibility across `:=`
  - method calls: parameter type compatibility
- example
  - `IfStmt`





## Method Calls

- type checking
  - correct number of actual parameters
  - match between actual and formal parameters re:
    - type compatibility
    - usage in some languages
- procedure vs function methods
- example
  - `ArgumentList`





## Testing

- at least 3 sets of tests:
  - valid constructs
    - test all alternatives (e.g. all forms of factor)
  - kind errors
    - all places where kind errors are reported
    - validate error recovery
  - type errors
    - all places where type errors are reported
    - validate error recovery
- all old tests (e.g. scope and syntactic analysis tests) should still run