Introduction To Semantic Analysis and Scope

Abertay University

v15.3.12

Agenda.

- Strategy and Architecture.
- The Symbol Table.
- Scope.
- Handling a Declaration.

(Long lecture – may break into two parts.)

Strategy and Architecture.

- What Is Semantic Analysis?
- Context is Everything.
- Implementation Strategies.
- Implementation with Ardkit.
  (with some caveats)

What Is Semantic Analysis?

- Semantic analysis checks that the **usage** (i.e. the meaning) of language components is correct and consistent in terms of their context.
  - Are names declared?
  - Are types used consistently?
  - Are operations consistent with the types of their operands?
  - … … …

Language Component Context.

- The **context** of a language component is determined by its position in the parse;
  - defined by a parse tree, or
  - defined by the calling sequence of recogniser methods.
- The meaning depends on where the component is within the program, and what has gone on before.

For Example ...

```
integer a
```

- We are concerned with the meaning of "a" in the context of a declaration.
- Has "a" been declared before?
  - if so then we have a semantic error,
  - if not then we must *remember* that the variable "a" has been declared and that it has a type of "integer".

… And Then We Have …

```
let a := b + 1.23
```

- We are now in the context of an assignment statement.
- Have "a" and "b" been declared?
- Do "a", "b" and "1.23" have consistent types?
- We will need to access the information we have remembered from the program's declarations.

An Implementation Strategy.

- Write **helper methods** for semantic analysis, corresponding to the syntax.
- **Call analysis methods** at appropriate points in the recogniser methods.
  - That is, the semantic checks are performed *while doing syntactic analysis*...
- Maintain a **symbol table** with type information about all user-defined names.
- Have recogniser methods **return type information** for the blocks they've matched.

```
private void recDecl () {
  int varType;
  if (have("integer")) {
    mustBe("integer");
    varType = LanguageType.Integer;
  } … …
  …DeclareId (scanner.CurrentToken,
               varType);
  mustBe (Token.IdentifierToken);
}
```

```
private void recLet () {
  mustBe("let");
  int varType =
    …CheckId (scanner.CurrentToken);
  mustBe (Token.IdentifierToken);
  mustBe (":=");
  int exprType = recExpr();
  …CheckTypesSame(varType, exprType);
}
```

Please note...

- “Have recogniser methods **return type information** for the blocks they've matched.”

- This *isn't* what Ardkit's `Semantics` does (or its examples, e.g. Block1Semantics.cs)
- Instead, it maintains a single `currentType` variable – “the type I'm expecting to see next”
- That approach only works well for *really really simple* languages – please don't use it for the coursework!

Other Strategies...

- The approach we're using in this module is to do semantic analysis while parsing – what a single-pass compiler would do
- You can also do semantic analysis as a separate pass – i.e. you parse as we saw last time, building an AST, then walk over the tree filling in the types.
- For some kinds of languages (e.g. those with type inference) this makes life much easier!

The *LanguageType* Class.

- Ardkit's representation of a **type expression** within the programming language (represented as `int`).
- Encapsulates type constants and string conversion methods for a range of primitive types:

  integer, real, boolean and string.
- All members of this class are static.

Semantic Analysis Classes.

- Encapsulate functionality within a dedicated semantic analysis class.
  - Parser requires only a reference to an object of this class.
  - Semantic analyser requires a reference back to the parser to allow access to its state
    - e.g. the *IsRecovering* property.

The ISemantics Interface.

```
public interface ISemantics {
      int CurrentType { get; set; }
      void ResetCurrentType ();
} // end ISemantics interface.
```

- The same interface-abstract-custom structure as for lexing/parsing.
- The only thing here is the CurrentType property – which I've said not to use!
- You'll extend this with what you need for your own language implementation.

Architecture Outline.

```
class MyParser : RecoveringRdParser {

  private MySemantics semantics;

  public MyParser ()
  : base (new MyScanner())
  {
    semantics = new MySemantics (this);
  }
  … recogniser
    methods …
}
```

```
class MySemantics : Semantics {
  … … …
  public MySemantics (IParser p)
  : base (p)
  { … … … }

  … language semantic analysis methods …
}
```

The *Semantics* Class.

- Implements the *ISemantics* interface; refer to the Ardkit documentation.
- You'll extend it with protected attributes for the semantics information you need.
- *semanticError(…)* reports an error; for *RecoveringRdParser*, it checks whether we're in recovery mode and ignores errors.

The Symbol Table.

- Overview Of Role.
(Simple for now – more complex next.)

- Outline Responsibilities.

- Architecture.

- *Symbol* Class.

Overview Of Role.

- The **symbol table** is the main data structure used in semantic analysis and artifact generation.
- It's a record of user-defined names in a program together with their attributes (e.g. the type of each variable).
- This table is maintained during semantic analysis, and subsequently used during artifact generation.

Outline Responsibilities.

- Stores all user-defined names and attributes.
- Attributes reflect language requirements; e.g.
  - user-defined name.
  - associated language type.
  - … other depending on the meaning of the identifier (e.g. in a language where you can define your own types, they'd be here too)
- Adds a name and its attributes.
- Checks if a specified name is defined.
- Gets the attributes for a specified name.

Outline Class Design.

- A *Symbol* class to represent a single name with its attributes.
- A *SymbolTable* class with a dynamic collection of *Symbol* objects.
- A *Scope* class to encapsulate symbol tables within a scoped name context (we'll come back to this next lecture).

Symbol Table Interfaces.

```
public interface ISymbol {
      String Name { get; }
      int Type { get; }
      IToken Source { get; }
}

public interface ISymbolTable {
      bool IsDefined (String name);
      bool Add (ISymbol symbol);
      ISymbol Get (String name);
}
```

The Symbols Framework.

- See the *Ardkit* symbols framework.
- `ISymbol`
  - `Symbol :` *Name, Type, Source*
    - » `VarSymbol`
    - » `ConstSymbol`
    - » `ArraySymbol`
    - » `FunctionSymbol`

Scope.

- What Is Scope?
- Representing Scope.
- A *Scope* Class Implementation.
- Using The *Scope* Class.

Scope Is …

… the area of program text in which a name has a specific valid meaning.

… in most languages, determined at compile time, rather than runtime.

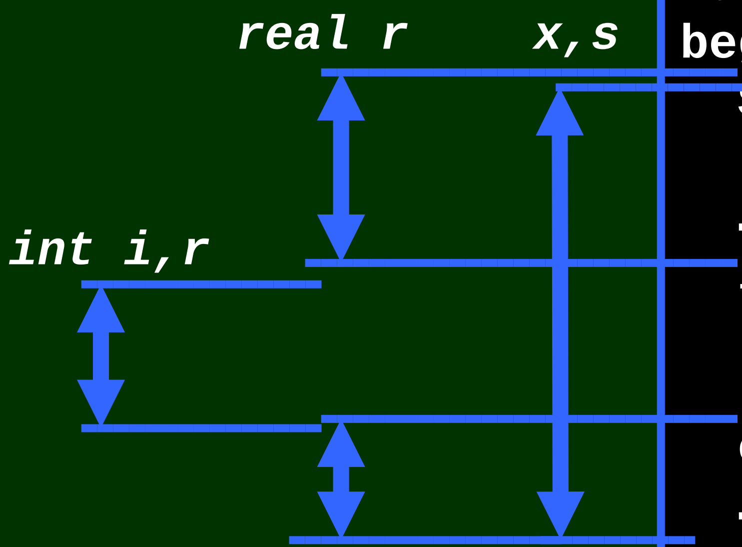
… dependant on the semantics of each different language.

… a property that must be reflected in the semantic analysis of a program.

Scope constructions.

- A scope is a syntactic entity that defines the lifetime of names declared within it.
- In various languages this can be
  - a universe (program + libraries + OS...),
  - a program,
  - a class,
  - a subroutine, or
  - a block. { ... }

A Scope Example.




```
function int myFunc (int x)
begin
   String s;
   real r;
   … … …
   if ( … ) begin
      int i, r;
      … … …
   end
   … … …
end
```

This declaration **shadows** the existing r

- A similar layout would be an error in Java:
  – the scope is the same, but a specific name can only be declared once in a method.

Scope Semantics.

- Most languages that include scoped declarations have the following “lexical scoping” semantics – visibility is decided at compile time.
- An identifier is only accessible
  - within the scope construction in which it is declared,
  - including any nested scope constructions.

Handling Scope.

- Each scope construction can have its own symbol table for the names declared within it.
- It must also have access to the “parent” symbol table (to handle nesting);
  - i.e. the enclosing scope.
- Searching for a name always begins with the inner (closest) scope and works out through the generations of enclosing scopes.

Implementing Scope.

- Modelled by a *Scope* class that encapsulates the functionality and symbol table for a single scope construction. (This replaces our previous single symbol table.)
- A **stack** of *Scope* objects represents the nesting of scope constructions.

Static *Scope* API.

- *OpenScope()* will instantiate a *Scope* object and push it on to the scope stack.
  - Used to start a scope construction.
  - The constructor will be private.
- *CloseScope()* will pop the stack.
  - used to close a scope construction.
- *CurrentScope* property retrieves the current *Scope* object.

Why Use These Static Methods?

- Guarantees that only one scoping mechanism applies to a compilation.
- Hides the stack functionality.
- Hides and hence protects the instantiation of *Scope* objects.

The *Scope* API.

- Exposes the *ISymbolTable* interface.
  - *Add*() wraps the symbol table *Add()*.
  - IsDefined() *and* Get() are overridden to reflect searching of nested scopes.
- Depth() returns level of nesting of a name.

Ardkit Support.

- The *Ardkit* toolkit exposes a *Scope* class.
  - See *Scope* class reference page.
- This implements the *ISymbolTable* interface.
- Exposes and implements the API as described in this presentation.

Using Scope.

- Consider the EBNF extract

`<Block> ::= begin <Decls> <Stats> end ;`

```
private void recBlock ()
{
  Scope.openScope ();
  mustBe ("begin");
  recDecls ();
  recStats ();
  mustBe ("end");
  Scope.closeScope ();
}
```

To add new names use
`Scope.CurrentScope.Add (…);`

To retrieve symbols use
`Symbol s = Scope.CurrentScope.Get("a");`

A Declaration Example.

- Introduction.
- Declaring Variables.
- The Semantic Analysis Methods.
- Handling Errors.
- The Revised Recogniser.

(possibly break here?)

Introduction.

- We will look at some excerpts from a language EBNF and show how semantic analysis would be performed.
- This should enable you to apply the techniques used to any EBNF as a whole.
- For the excerpt the RD recogniser will be amended to include calls to appropriate semantic analysis methods.

Declaring Variables.

- Consider the following EBNF

```
<Decl> ::= (int | real) Ident (, Ident)* ;
```

```
private void recDecl () {
  if (have ("int"))
        mustBe ("int");
  else mustBe ("real");
  mustBe (Token.IdentifierToken);
  while (have (",")) {
    mustBe (",");
    mustBe (Token.IdentifierToken);
  }
}
```

And Now With Semantics...

```
private void recDecl () {
  int varType;
  if (have ("int")) {
    varType = LanguageType.Integer;
    mustBe ("int");
  } else {
    varType = LanguageType.Real;
    mustBe ("real");
  }
  semantics.DeclareId (scanner.CurrentToken, varType);
  mustBe (Token.IdentifierToken);
  while (have (",")) {
    mustBe (",");
    semantics.DeclareId (scanner.CurrentToken, varType);
    mustBe (Token.IdentifierToken);
  }
}
```

The *DeclareId()* Method.

```
public void DeclareId (IToken id, int varType) {
  if (!id.Is (Token.IdentifierToken)) return;
  Scope symbols = Scope.CurrentScope;
  if (symbols.IsDefined (id.TokenValue)) {
    semanticError (new AlreadyDeclaredError (
                id, symbols.Get(id.TokenValue)));
  } else {
    symbols.Add (new VarSymbol (id, varType));
  }
} // end DeclareId method.
```

Note …

… only process if an identifier token.

… access the symbol table for the current scope using the *Scope.CurrentScope* property.

… the *AlreadyDeclaredError* class is defined in Ardkit to represent the semantic error.

… instantiates and adds the appropriate symbol object denoting the identifier used as a variable.

Representing Semantic Errors.

- What semantic errors may be reported depends on the specific language being processed.
- You must define an error class for each possible semantic error that may occur.
- The *Ardkit* toolkit provides;
  - the *NotDeclaredError*, *AlreadyDeclaredError* and *TypeConflictError* classes for typical errors;
  - the *CompilerError* class to act as a base for your own semantic error classes.
  - Refer to the *Ardkit* class reference pages, and the “Using Errors” page.

Alternative Specifications.

- The form of the BNF specification can affect the implementation of the semantic analysis.
- Consider the following EBNF

```
<Decl> ::= <Type> Ident <Ident-List> ;
<Type> ::= int | real ;
<Ident-List> ::= , Ident <Ident-List> | <> ;
```

```
private void recDecl () {
  int varType = recType();
  semantics.DeclareId (scanner.CurrentToken,
                       varType);
  mustBe (Token.IdentifierToken);
  recIdentList(varType);
}
```

```
private int recType () {
  int varType;
  if (have ("int")) {
    varType = LanguageType.Integer;
    mustBe ("int");
  } else if (have ("real")) {
    varType = LanguageType.Real;
    mustBe ("real");
  } else … report error …
  return varType;
}
```

This recogniser method now returns some semantic information – the type expression that it recognised.

```
private void recIdentList (int varType) {
  if (have (",")) {
    mustBe (",");
    semantics.DeclareId (scanner.CurrentToken,
                         varType);
    mustBe (Token.IdentifierToken);
    recIdentList ();
  }
}
```

Declaration Styles.

- For a single-pass compiler, the language syntax affects the order in which information is available for semantic analysis.

```
<Decl> ::= decl Ident <Ident-List> as <Type>;
```

- We don't know the type of the variables being declared until the end of the declaration.
  - Store the identifiers in a List<Token>...
  - … and when the type is found,
    add all the identifiers to the symbol table.
  - (For multipass, this isn't a problem...)

Other Kinds of Declarations.

- So far we have looked at declarations of identifiers used as simple variables.
- In a programming language we might have to handle the semantics of components such as
  - constants
  - procedures / functions with typed signatures
  - typed formal parameters
  - user-defined types
  - classes with internal fields and methods
- Symbol subclasses must reflect the semantic requirements of these components.

You Can Now …

… describe the role of semantic analysis.

… describe how semantic analysis can be incorporated into a RD compiler.

… explain the role of the symbol table and its required functionality.

… explain the role that scope plays when type-checking a program.

… implement semantic analysis for simple variable declarations.