Languages and Compilers
Semantic Analysis: Expressions and More

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Today's Plan...

- Analysing Expressions.
- Other Types of Identifiers.

Analysing Expressions.

- Handling Variables and Literal Constants.
- Type Checking.
- Statements & Expressions.
- Assignment Example.
- Expression Example.

Analysing Variables and Literals.

- If the RD parser drives the semantic analysis:
  - (as we saw last week – not the only way)
- To semantically analyse the use of a variable:
  - ✓ check the name has been declared, and
  - ✓ check for appropriate type compatibility.
- To semantically analyse the use of a **literal:** (constant values, e.g. 42, 3.141, “hello world”)
  - ✓ check for appropriate type compatibility.

General Strategy.

- Recognisers return type information.
- Let's have some Semantics methods:
  - *CheckId(token)* – recognise a variable identifier, and return its type (if your BNF contains Variable, you might not even need the helper!)
  - *CheckTypesSame(type1, type2)* – check whether two types are compatible (in a simple language, whether they're the same type or not)

```
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);
}
```

The *CheckId* Method ...

- Supply with the required token.

```
public int CheckId (IToken id) {
```

  - (remember **`int`** is how Ardkit represents type expressions)

- If not an identifier then return a dummy type.

```
  if (!id.Is (Token.IdentifierToken))
    return LanguageType.Undefined;
```

… *CheckId* continued.

- If the variable name is not defined in the current scope (see last week!) then report a semantic error, and return a dummy type …

```
  if (!Scope.CurrentScope.IsDefined (id.TokenValue)) {
    semanticError (new NotDeclaredError (id));
    return LanguageType.Undefined;
  }
```

- … else look up and return the type

```
  return Scope.CurrentScope.Get (id.TokenValue).Type;
} // end CheckId method.
```

The *CheckLiteral* method.

- We could have a very similar helper method that takes a token representing a literal, and returns its type...

```
int CheckLiteral (IToken lit) {
  if (lit.Is (Token.IntegerToken))
    return LanguageType.Integer;
  else if (lit.Is (Token.RealToken))
    return LanguageType.Real;
  else
    return LanguageType.Undefined;
}
```

- Or you could do this in recLiteral, if your BNF required you to write that

CheckTypesSame...

- The arguments are the two types:

```
public void CheckTypesSame (?????,
                            int oldType, int newType) {
```

- Complain if they don't match:

```
  if (oldType != newType) {
    semanticError (new TypeConflictError
                       (?????, newType, oldType));
  }
} // end checkType method.
```

CheckTypesSame completed

- The arguments are the two types:

```
public void CheckTypesSame (IToken token,
                            int oldType, int newType) {
```

- Complain if they don't match:

```
  if (oldType != newType) {
    semanticError (new TypeConflictError
                        (token, newType, oldType));
  }
} // end checkType method.
```

- We also need the token to give a source location in the error message!

CheckTypesSame in practice

- This is a very simple kind of type checking! No implicit type conversion...
- Many real languages would need more complex logic in typechecking methods
  - identifying when one type can be automatically converted to another
  - handling user-defined types
  - etc. etc.

Dealing with Expressions.

- Consider the EBNF extracts

```
<Expr> ::= <Factor> ( (+|-) <Factor>)* ;
<Factor> ::= Ident | Integer | "(" <Expr> ")" ;
```

```
private void recExpr () {
  recFactor ();
  while (have(“+”) || have("-")) {
    if (have("+")) mustBe ("+");
    else           mustBe ("-");
    recFactor ();
  }
}
```

- How do we add semantics to this?
- Make each production return the type!

Flashback to the Type Inference lecture...

```
(height / num) * (i + 1)
 float    int     int int
    float            int
          float
```

- Start with the simplest parts of the expression (the **terminal symbols**), and work outwards
- `1` is a constant, of type `int`
- `...`
- So the type of this expression is `float`

Dealing with Expressions.

```
private int recFactor () {
  int ty;
  if (have (Token.IdentifierToken)) {
    ty = semantics.CheckId (scanner.CurrentToken);
    mustBe (Token.IdentifierToken);
  }
  else if (have (Token.IntegerToken)) {
    ty = LanguageType.Integer;
    mustBe (Token.IntegerToken);
  }
  else {
    mustBe ("(");
    ty = recExpr();
    mustBe (")");
  }
  return ty;
}
```

Dealing with Expressions.

```
private int recExpr () {
  int lTy = recFactor ();
  while (have("+") || have("-")) {
    if (have("+")) mustBe ("+");
    else           mustBe ("-");
    int rTy = recFactor ();

    // a+b and a-b require a and b to
    // have the same type.
    semantics.CheckTypesSame(scanner.CurrentToken,
                             lTy, rTy);
  }
  return lTy;
}
```

(What would we need to do to handle a/b ?)

Assignment Example.

- Consider the EBNF extracts

```
<Statement> ::= <Assign> | … ;
<Assign> ::= Ident = <Expr> ;
```

```
private void recAssign () {
  int varTy = semantics.CheckId(scanner.CurrentToken);
  mustBe (Token.IdentifierToken);
  mustBe ("=");
  IToken expToken = scanner.CurrentToken;
  int expTy = recExpr();
  checkTypesSame(expToken, varTy, expTy);
}
```

Assignment Example.

- Cons

Still void – there's no type information to return from an assignment (in this language!)

```
<Statement    <Assign> | … ;
<Assign>      Ident = <Expr
```

```
private void recAssign () {
  int varTy = semantics.Che
  mustBe (Token.Identifi    Token);
  mustBe ("=");
  IToken expToken = scanner.CurrentToken;
  int expTy = recExpr();
  checkTypesSame(expToken, varTy, expTy);
}
```

Save the token at this point, so we report the error at the right place in the source code...

Wider Use Of Expressions.

- Expressions are also used in other kinds of statements – e.g. `if`/`while` conditions.

- The semantic methods defined here can also be used in those contexts.
  - e.g. for a `while` loop's condition, we could parse the expression, and check that its type matches `LanguageType.Boolean`

Other Uses Of Identifiers.

- User-Defined Names.
- Constants, arrays and Subroutines.
- Symbol Class Hierarchy.
- User-Defined Types.

User-Defined Names.

- One of the main roles of semantic analysis is checking the use of user-defined names (or identifiers).
- To date we have looked at simple variables but there may be other uses of names
  - constants,
  - arrays,
  - functions,
  - user-defined types
  - etc. etc.

Extending The *Symbol* class.

- The *Symbol* class must be extended to handle the different uses of identifiers.
  - Responsibility of this class is to store an identifier name with its attributes.
  - These different uses of names require modified attributes.
  - See the *Ardkit* toolkit provision of the subclasses to represent variables, constants, arrays and functions.

Arrays.

- A language may allow an identifier to denote an array.
- An array is a fixed-size collection of locations, (usually) all of the same type.
- Arrays supported by the Ardkit *ArraySymbol* class may have multiple dimensions with specified bounds but must be regular.
- Languages might not need to treat arrays specially – functional languages generally don't

Functions.

- A language might have functions, procedures or methods as components; we will use the generic term “function” for these.
- Functions are (usually) named blocks of code, with optional parameters, that may return a value.
- Must distinguish between requirements for
  - the function declaration (formal parameters)
  - the function call (actual parameters)

Function Call.

- Assume the following function declaration in some language:

```
int sum (int x, int y)
begin … … … end
```

- What attributes are required to analyse the call?

```
i = sum (a, 5)
```

- Looking up **sum** in the symbol table, the following attributes will allow the call to be analysed.
  - type returned by the function,
  - number of parameters, and
  - type of each parameter
- Note that the **names** of the formal parameters are not required for the analysis of the call.

Function Declaration.

- Consider the following function declaration in some language:

```
int sum (int x, int y)
begin … … … end
```

- What attributes are required to analyse the function body?

```
begin … … … end
```

- Within the function body the code will be analysed in the context of the function name and the collection of formal parameters.
- We create a new scope for the function body, and put the formal parameters into that scope's symbol table.

Ardkit Support for Functions.

- Refer to the Ardkit reference page for the *FunctionSymbol* class.

- Review the source code for the symbol classes' implementations.
- Note that the API is targetted at processing
  - the declaration of a function signature (setting the symbol's attributes),
  - the declaration of the function body (getting the formal parameter attributes), and
  - the calling of a function (getting the attributes).

User-Defined Types.

- A language may allow user-defined types to be declared.
- Type-checking code needs to take these new types into account.
- In Ardkit, we'd need to create new symbol classes for the kinds of types that users can declare
  - e.g. a product/struct type would need to list the fields it contains with types and names...

Summary.

We Have Seen How …

… variables and literal constants need to be semantically analysed,

… expressions can be type-checked by passing type information between recogniser methods,

… identifiers can be used with different meanings with different semantic analysis requirements, and

… a symbol class hierarchy can represent these different meanings.