COMP 520 - Compilers

Lecture 10 (Tue Feb 14)

Contextual Analysis: Identification

• Please pick up at the back of the room
  – WA2 graded papers
  – (A-J) (K-R) (S-Z)

• Reading for Tue 2/21
  – Chapter 5: Contextual Analysis - section 5.1 (pp 136 - 150)
Topics

- **WA2 review**
  - first problem

- **Bottom-up parsing**
  - try the example

- **Identifiers**
  - identifiers and what they denote
  - scopes

- **Identification**
  - Implementation strategies
Identifiers

• An identifier has a
  – name - a string
  – denotation – what it represents in the context in which it is used

• Examples of identifiers in Java

```
Token id = new Token(TokenKind.IDENTIFIER, "x");
```
### Identifier denotations

- Identifiers have many denotations in modern programming languages

<table>
<thead>
<tr>
<th>Category</th>
<th>Denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>memory address(es)</td>
</tr>
<tr>
<td>Method</td>
<td>executable code address</td>
</tr>
<tr>
<td>Type</td>
<td>interpretation of values and operations, e.g. a class name or a basetype like “int”</td>
</tr>
<tr>
<td>Classname</td>
<td>provides access to members of a class</td>
</tr>
<tr>
<td>Member</td>
<td>members of a class (or components in a record)</td>
</tr>
<tr>
<td>Namespace</td>
<td>provides access to a collection of externally defined identifiers, e.g. package name in import</td>
</tr>
<tr>
<td>Literal Value</td>
<td>e.g. true, false</td>
</tr>
</tbody>
</table>
Contextual analysis: Identification

• Identifiers are
  – defined (introduced)
    • typically through declarations
    • sometimes “pre-defined” (e.g. true, false in Triangle)
  – referenced (used)
    • occurrences other than in a declaration
    • we generally call these “references”
    • our book calls these “applied occurrences”

• Identification
  – record definitions of identifiers and their attributes when declared
    • attributes describe the category and specific details of a declaration
  – relate each reference to the appropriate attributes
    • our book calls this “identification”
    • in modern languages this is non-trivial
Identification in the AST - Triangle

- Traverse AST
  - Record definitions in Declaration nodes
  - Link references to defining declaration

```plaintext
let
  var n: Integer;
  var c: Char
in
  begin
    c := '&';
    n := n + 1
  end
```
Scope of a declaration

• Monolithic block structure
  – All declarations are in a single global scope
    • No identifier can be declared more than once
    • .. so each reference has at most one controlling declaration

• Flat block structure (two-level scope)
  – Global scope and local scope
    • Single global scope and multiple disjoint local scopes
    • Each identifier declared at most once in global scope, and at most once
      in a given local scope

• Nested block structure
  – Arbitrary nesting of blocks
    • Declarations in a more deeply nested block hide those in enclosing
      blocks
More complex notions of scope

• An identifier may have multiple definitions
  – imports from other packages
  – class name, constructor
  – overloading
  – Inheritance
  – qualified reference
  – visibility (public / private)
  – accessibility (static / instance)

• Examples (Java)
  ```java
  int Foo = 3;

  Foo id = new Foo();

  Foo.method(Foo);
  ```
Java Identification

Token id = new Token(TokenKind.ID, “x”)

• How to determine the definition that applies to a reference?
  – context
    • Java class names can only appear in some places (where?)
    • variable, function and procedure names can appear in other places
  – qualified access
    • prefix determines applicable definitions
      – e.g. System.out.println(…)
  – visibility rules
    • a subset of definitions is visible at a given program point
      – scope rules: local variables, parameters, members, classnames
      – inheritance of class or interface(s)
      – qualified references
      – accessibility: public / private / protected
  – type rules
    • overloading
      – foo(5), foo(“string”)
Scopes: Nested block structure in Triangle

let
  var a: Integer;
  var b: Boolean
in
  begin
    ... a, b ...
  let
    var b: Integer;
    var c: Boolean
  in
    begin
      ... a, b, c ...
    let var d: Integer
    in
      ... a, b, c, d ...;
    ... a, b, c ...
  end;
  ... a, b ...
end

• The Triangle block command
  – let Declaration in SingleCommand

  – the scope of the declaration is limited to the SingleCommand

  – types, functions, procedures, variables can be declared

  – a declaration hides the definition of the same name in a surrounding scope

  – a use (an applied occurrence) refers to the nearest surrounding declaration
Subtleties in nested block structure

let
  const a ~ 3;
  const b ~ 4
in
  begin
    ... a, b ...
  let
    var b ~ a + 5;
    var c ~ b + 6
  in
    ... a, b, c ...
end

• Initializers in declarations
  – a variable can be given an initial value through evaluation of an expression
  – what definitions apply when the initializing expressions are evaluated?
Identification: implementation

- Identification table (a.k.a. symbol table)
  - maps identifier names to attributes
    - attributes vary greatly depending on the category of identifier
      - strategy: the attributes of an identifier are in the AST node where it is declared
      - all declaration nodes in miniJava AST are subtype of Declaration (Decl)

- implementation
  - (auto-expanding) hashtable
    - O(n) amortized access cost for O(n) insertions and lookups

  - Java: class HashMap<String,Decl>
    - clear()
    - boolean containsKey( String id )
    - Decl put( String id , Decl decl ) // associate id with decl
    - Decl get( String id ) // decl or null, if id not in hashmap
    - void remove( String id ) // remove current association of id, if any
### Scoped Identification table

- **Extends hashtable with two operations**
  - openScope()
  - closeScope()
  - Get(id.spelling()) returns innermost declaration

- **Implementation challenges**
  - remove mappings when leaving scope
  - handling multiple declarations
Identification in Java

- **parameters to the identification process**
  - current package
    - access to all top-level classes
  - scoped identification table
    - enclosing variable declarations
    - enclosing parameter declarations
  - identification table for current class
    - this is scoped for nested classes
    - may be scoped to reflect inheritance
  - identification tables for other classes
    - explicit imports
    - implicit imports, e.g. same package

- in full Java, identification process returns *list* of possible definitions or error
  - type checking provides final disambiguation
Identification in miniJava

- **Parameters to the identification process**
  - Class declarations
    - to identify uses of class names e.g.
      ```java
      Foo x = ... 
      new Foo()
      ```
  - Member declarations in current class
    - to identify uses of fields or methods
  - Local declarations in current method
    - to identify uses of parameters or local variables
  - Member declarations in other classes
    - to identify qualified references, e.g.
      ```java
      Foo.field
      x.y.z
      ```

- **Each Identifier occurrence in a miniJava AST has a unique declaration**
  - almost always
class Foo {
    int x;
    int p(int y) {
        if (p(x)) {
            int x = 10;
            x = y;
        }
        return x;
    }
}
class Foo {
    int x;
    int p(int y) {
        if (p(x)) {
            int x = 10;
            x = y;
        }
        return x;
    }
}
Logical order of Contextual analysis

1. Identification
   - check validity of declarations
     • is this declaration allowed in the current context?
   - link references to corresponding declarations
   - AST traversal order
     • top down, declarations before references

2. Type checking
   - assign types to expressions
   - check type agreement
     • operators and operands
     • assignment statements
   - AST traversal order
     • bottom up (assuming no overloading)
Contextual analysis in a single traversal

• For each node
  – *inherit* some information from parent
    • e.g. Identification table
  – traverse subtree rooted at node
  – *synthesize* some information to return to parent
    • e.g. type of expression computed by node
    • e.g. updated identification table

• Traversing the subtree rooted at a node
  – for each child in turn
    • apply contextual analysis on child
    • providing inherited data
    • receiving synthesized data
Example contextual analyses in Triangle

- **Contextual analysis of Let command**
  - start a new scope in identification table
  - contextual analysis of Declaration
    - updates identification table
  - contextual analysis of Command
  - remove scope in identification table

- **Contextual analysis of BinaryExpression**
  - contextual analysis of left expression
    - save returned type
  - contextual analysis of right expression
    - save returned type
  - look up operator argument types and result type
    - check agreement with operator