Imperial College London
Department of Computing

Compilers (221)

Exercises – Semantic Analysis

Check your answers with the tutorial helpers during tutorials and with each other on Piazza.

Let me know if you find a mistake or have a better answer. The check methods below are not exhaustive. Nor are they specific to a particular programming language. They illustrate some of the checks that often required. Each programming languages will have its own set of semantic rules that the compiler and/or runtime system needs to check. Take a look at the language specification document for your favourite programming language to see how it defines semantic rules for particular statements/constructs.

1.	Consider the following errors that might be present in a Java-like program: (i) undefined variable (ii) access to a non-existing element of an array (iii) integer overflow (iv) incompatible type (v) unmatched ‘)’ (vi) non-terminating loop For each error, indicate when the error can be detected and briefly describe what the detection involves. If specific cases of an error are detected at different times describe the specific cases. ++ (i) This can be done during semantic analysis, by checking whether the symbol table has a valid variable definition for the identifier. (ii) This is normally done at runtime, by checking the value of the subscript expression against the lower and upper bounds. Can also be done at compile-time if the expression can be evaluated or the value inferred, e.g. a constant expression. (iii) Similar to (ii), but we may also able to let the CPU detect overflow at runtime and catch the exception that arises. Note some languages do not consider overflows as errors. (iv) Normally done at compile-time by checking the types held in symbol tables. May involve type casting/inferences. Certain checks may be required at runtime, e.g. checking whether an object is of a particular class. (v) Syntax error dedicated by the parser. (vi) This is not normally considered an error since non-terminating loops are acceptable in many programs, particularly concurrent programs.	L2
2.	For each of the following statements: a) `if expr then statement [else statement] fi` b) `for id = expr (to | downto) expr do statement od` c) `switch expr { case (int | default) : StatementSeq } endswitch` Write the ASTnode class for these statements including syntactic and semantic attributes as well a semantic check method. Your method for `for` should work integer, character and boolean expressions. You can assume that there is a function `scalartype` that tests if a given type is of a particular standard type, e.g. `int`, `char`, `boolean`. ++	L5

```
def scalartype(type, name):   # checks if a type is a standard type
   return type == TOP_ST.lookup(name):

class IfAST extends AST:
   ExpressionAST E          # Syntactic value
   StatementAST  S1         # Syntactic value
   StatementAST  S2         # Syntactic value

   def check ():
      E.check()             # note: E.check() also records the type of E
      if  ! scalartype(E.type, ''boolean''):
         error(''if conditional expression not a boolean'')
      S1.check()
      if S2 != None: S2.check()

class ForAST extends AST:
   String forloopvarname    # Syntactic value
   ExpressionAST E1         # Syntactic value
   ExpressionAST E1         # Syntactic value
   StatementAST  S          # Syntactic value
   Boolean ascending        # Syntactic value to=true, downto=false
   VARIABLE forvar          # Semantic value

   def check():
      E1.check()
      E2.check()

      F = ST.lookup(forloopvarname)

      if    F == None: error(''for variable not declared locally'')
      elif  ! F instanceof VARIABLE:  error(''for variable not a variable'')
      elif  ! scalartype(F.type, ''integer'') and
            ! scalartype(F.type, ''boolean'') and
            ! scalartype(F.type, ''char'') and ....
               error(''For variable not of a supported type'')
      elif  ! assignCompat(F.type, E1.type):
               error(''From expression type not compatible with for variable'')
      elif  ! assignCompat(F.type, E2.type):
               error(''To expression type not compatible with for variable'')
      else  S.check()

class SwitchAST extends AST:
   ExpressionAST E          # Syntactic values
   CaseDefaultASTList  L
   CaseLabels labels        # Semantic value – holds list of all int labels
                            # defaults treated as labels.

   def check ():
      E.check ()
      if  !scalartype(E.type, "int"):
           error ("Switch expression not of int type")
      else:
         labels = [ ]

         for C in L:  # check each case-default clause
            C.check ()   # C.check includes overflow check

            if  C.label in labels:
               error ("Case/Default label already declared")
            else labels.append (C.label)

            if L.len=0: error ("Need at least 1 case/default")
```

3. Give an AST class for a **return** statement e.g. L3

```
ReturnStatement → return | return Expression
```

Your class should include syntactic and any semantic attributes as well a *check* method that reports as many semantic errors as possible. You can assume that there is a function *Get_Enclosing_Method* that returns a reference to the method object for the method that encloses the return statement, or **null** if there is no enclosing method.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
class ReturnAST extends AST:
   ExpressionAST  E   # syntactic value for Return expr or none

   def Check ( ):
      M = Get_Enclosing_Method ( )   # returns ref to current method ref or None

      if E:  E.check ( )

      if M == None:
         error (‘‘return not in a  method’’)
      elif M.isconstructor and E != None:
         error (‘‘constructors cannot return a value’’)
      elif M.ReturnType == VOID and E != None:
         error (‘‘method is void - cannot return a value’’)
      elif M.ReturnType != VOID and E == None:
         error (‘‘return value required’’)
      elif not Compat (E.Type, M.ReturnType):
         error (‘‘expr not type compatible with method return type’’)

      M.ReturnsInMethod++   # Increment number of returns in method.
                            # Use elsewhere that we have at least one return
```

4. Develop an AST class for a class declaration, e.g. L5

```
ClassDeclaration → class classid [extends superclassid] { Block }
```

Your class should include syntactic and any semantic attributes as well a check method that reports as many semantic errors as possible. You may wish to use the following type for the semantic attribute:

```
class CLASS extends TYPE {CLASS superclass, SymbolTable locals}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
class ClassDeclAST extends AST:
   String classname      # Syntactic value
   String superclassname # Syntactic value
   BlockAST block        # Syntactic value
   CLASS type            # Semantic value

   def check():
      C = ST.lookup(classname)
      if  C != None:
         error(‘‘class already defined’’)
      else:
         type = new CLASS()
         ST.add(classname, type)

         if superclassname != "":      # check superclass
            S = ST.lookupAll(superclassname)
```

```
            if S == None:
               error(''superclass name not declared'')
            elif ! S instanceof CLASS:
               error(''superclass is not a class'')
            elif S.isFinal():
               error(''superclass is final'')  # in Java
            elif assignCompat(C,S):
               error(''superclass is also a subclass!'')
            else
               type.superclass = S         # link to superclass object
               ST = new SymbolTable(ST)    # open a new scope
               type.symtab = ST            # allows us to check X.Y style names
               block.check()               # check and build block
               ST = ST.encSymTable         # return to enclosing scope
```

5. Write an AST class for a **new** expression, e.g. L4

```
NewExpr → new classid '(' expressionlist ')'
```

Your class should include syntactic and semantic attributes as well as a check method that reports as many semantic errors as possible. Assume that classes have at most one constructor i.e. ignore constructor overloading.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
class CLASS extends TYPE {
   CLASS superclass;
   SymbolTable locals;
   boolean isabstract;
}

class NewExprAST extends AST:
   String classname            # syntactic value
   ExpressionlistAST params    # syntactic value
   CLASS inst                  # semantic value

   def Check ( ):
      C = ST.lookupAll(classname)

      if     C == None: error (''classname is unknown'')
      elif  ! C instanceof CLASS:  error (''classname is not a class'')
      elif  ! C.isabstract :  error (''class is an abstract class'')
      else:   # locate constructor if any

      Cons = C.locals.lookup (classname)
      if  Cons == None && params.len > 0: error (''too many params'')
      elif Cons.formals.len > params.len: error (''too many params'')
      elif Cons.formals.len < params.len: error (''too few params'')
      else:  # check parameter types
         for k in params.len:   // params is indexable
            params [k].Check ( )     # check parameter
            if assignCompat (Cons.formals[k].type, params [k].type)
               error (''type of param k incompatible'')
         inst = C
```

6.	Most object-oriented languages allow method overloading, for example:
	
	`void Apply (int)`
	`void Apply (boolean)`
	`void Apply ()`
	`void Apply (int, int)`
	`int  Apply (String)`
	
	What compilation issues does this feature present?
	
	++
	
	If method overloading is permitted, there can be different methods of the same name but different arguments types/return types, the compiler must choose between methods based on the *types* and *number* of the actual and formal *parameters* and expected *return* type. Constructor overloading is also supported by some languages.
	
	When processing the declaration of an overloaded method, the new declaration must not hide the old definitions, e.g. the symbol table must allow multiple identical names in the scope, or the same name to map to multiple implementations.
	
	Depending on the language semantics it may be necessary to produce an error if overloaded methods have parameter types that are too "similar", or methods with the same parameter types but different return types.
	
	Need to consider scope of overloaded methods as well as inheritance and interface types.
	
	Need to have a naming scheme for overloaded methods to use at link-load-run times and for profilers and debuggers.
	
	Some languages even allow different implementations of a method to be chosen based on the runtime type of an actual parameter.
	
	Dynamic class loading may require special treatment
	
	Produce good error messages requires some care.
		L3

7.	Some languages allow the programmer to omit variable declarations entirely. Other languages require the programmer to declare all variables, but not to declare their types. Still other languages require the programmer to declare both variables and their types. Give a short argument in favour of each of three approaches. Which argument do you find most convincing? Why? ++ The 1st approach leads to shorter programs that are arguably easier to write and read. It provides polymorphism "for free," and relieves the programmer of the need to prove to the compiler that the program is type safe. The 2nd approach is naturally polymorphic, but avoids errors due to misspellings and to accidental use of the same variable name for multiple purposes. The 3rd approach is self-documenting, and arguably forces the programmer to think through things more carefully, reducing the incidence of bugs. Bugs that do occur are likely to be found at compile time. Given detailed knowledge of types, the compiler can produce more efficient code than it can for the first two approaches. Which approach is better is largely a matter of personal preference. Experience suggests, however, that very large systems are easier to maintain (and run much faster) when written in statically typed languages.	L3