Semantic Analysis

Is the Program “Legal” ?

Naranker Dulay
n.dulay@imperial.ac.uk

https://www.doc.ic.ac.uk/~nd/compilers

Java compiler

If you're interested you can download the source code of the Java compiler (see links to zip, bz2 or gz file on the LHS of the webpage) from:

`http://hg.openjdk.java.net/jdk9/jdk9/langtools`

The compiler is very complex reflecting the large feature set and complex semantics of Java. Nevertheless, it's instructive to take a peek at some of it. It's nicely written.

The compiler is hidden away in

`src/jdk.compiler/share/classes/com/sun/tools/javac/`

Parts of interest:

`parser/Tokens.java` and `Scanner.Java` – scanner

`parser/JavacParser.java` - recursive descent parser

`comp/Attr.java` – semantic analysis

`comp/Check.java` – type checking

`jvm/Code.java` – code generation

Semantic Analysis

Context free grammars cannot check all the necessary properties of a legal program, for example, declaration related properties like scoping or type correctness.

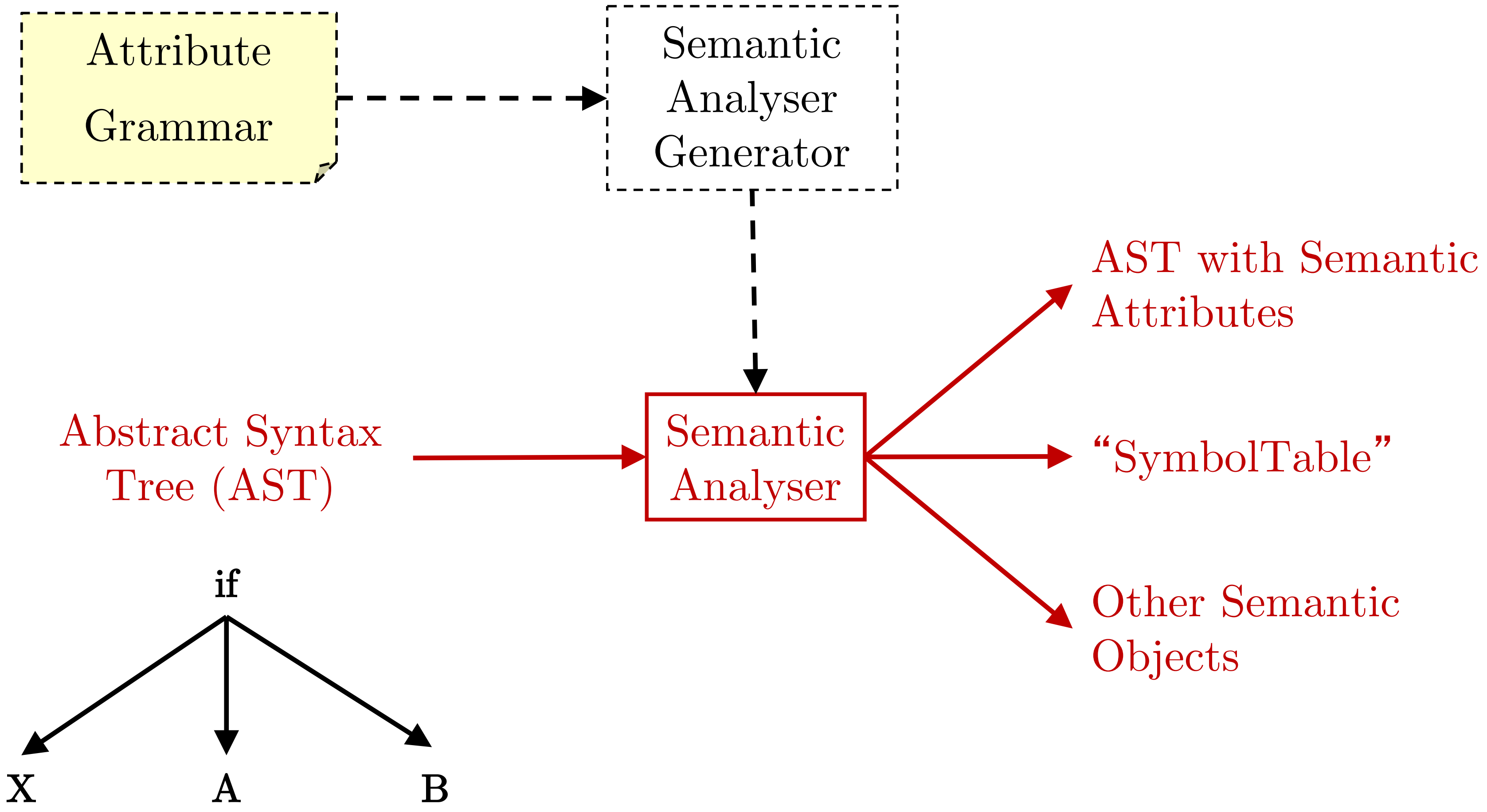
The semantic analysis phase of a compiler attempts to statically check whether a given program conforms to the semantic rules of the language, as well as provide additional information for the code generator and runtime system.

The degree of compile-time checking possible varies from language to language. With languages like Python little compile-time checking can be done. Others, like Java, afford a wide range of compile-time checking.

If semantic checks cannot be done at compile time, then it is incumbent on the language's runtime system to perform the checks.

Semantic Analysis is also known as **Context Analysis**.

Approach


Attribute
Grammar
Semantic
Analyser
Generator
Abstract Syntax
Tree (AST)
Semantic
Analyser
AST with Semantic
Attributes
"SymbolTable"
Other Semantic
Objects
if
X
A
B

Semantic Checks

- `AgeT Age`
- `Age =` *Expr*
- `int Score[Teams]`
- `Score[Team] =` *Expr*
- `double Calc(int Age, string Name, Address Addr) {...}`
- `Alpha = Calc(`*X*`,` *Y*`,` *Z*`)`

Variable Declaration

`AgeT Age`

- Check that the identifier **`AgeT`** is in *scope.* Does it mater if it is the current scope or an enclosing scope?
- Check that the identifier **`AgeT`** is defined as a type.
- Is it is permitted to declare variables of type **`AgeT?`** What if AgetT was void or an abstract class?
- Check that the identifier **`Age`** has not already been declared in the current scope. Is it an error if it has been?
- If **`AgeT`** is defined in an enclosing scope should **`int AgeT`** be allowed in an inner scope? What about **`AgeT AgeT`**?

Assignment

`Age = `*`Expr`*

- Check that identifier **`Age`** in scope and is declared as a variable. Could **Age** be declared as a parameter?

- Check that it is permitted to assign values of type ***`Expr`*`.type`** to variables of type **`Age.type`** (assume **`type`** is a semantic attribute of **`Age`** and of ***`Expr`***).

- What are the assignment-compatibility rules in different programming languages, e.g. in Java, C++?

- If ***`Expr`*** is a constant numeric expression, does its value lies within the range of **`Age.type`**?

Array Declaration

`int Score[Teams]`

- Check that the identifier `int` is in scope and defined as a type?
- Check that it is permitted to declare arrays whose elements are of `int` type?
- Check that `Teams` is an `int` constant (`int` expressions in some languages). Check that the value of `Teams` is greater than zero.
- Check that the identifier `Score` has not already been declared in the current scope?

Another check?

- If `Teams * sizeof(int)` is a very large number, should we give a compile-time warning about the size of the array?

Array Element Assignment

`Score[Team] = `*`Expr`*

- Check that the identifier **`Score`** is in scope and declared as a variable.
- Check that **`Score.type`** is an array type. Check that it is a one-dimensional array type. Check that **`Team.type`** is an integer type.
- Check that it is permitted to assign values of ***`Expr`*`.type`** to variables of **`Score.type.arrayElementType`**?
- If **`Team`** is a constant expression and the bounds of **`Score`** are known, check that **`Team`** lies within those bounds.
- For multi-dimensional arrays is it permitted to slice arrays? For example if we have **int Score[10][10], with** Score[3] refering to a row. Is it permitted to index on other types, e.g. booleans, strings?

Function Declaration

```
double Calc(int Age, string Name, Address Addr) {...}
```

- Check identifiers **`double`**, **`int`**, **`string`**, **`Address`** are in scope and declared as types.
- Check that it is permitted to declare functions of type **`double`** and parameters of type **`int`**, **`string`**, **`Address`**.
- Check that **`Calc`** is only declared once in this scope. Is overloading of functions permitted?
- Check that **`Age`**, **`Name`** and **`Addr`** are only declared once in the scope of function **`Calc`**. Is there any need for 2 scopes, one for parameters and one for the function body?
- Check that **`Age`**, **`Name`** and **`Addr`** are only accessible inside the body of **`Calc`**. The semantic rules for classes/packages are more complex/interesting.

Function Call

```
double Calc(int Age, string Name, Address Addr)
Alpha = Calc(X, Y, Z)
```

- Check that the type of expression ***X*** **is** parameter-compatible with **Calc.Age.type** (i.e. with **int**), ***Y*** with **Calc.Name.type** (i.e. with **string**), ***Z*** with **Calc.Addr.type** (i.e. with **Address**)?

- Check that the type of **Calc.returntype** is returntype-compatible with **Alpha.type**?

- Are the number of parameters in the function call equal to the number of parameters in the function declaration?

Type Checking in Languages

The semantic rules for type checking vary greatly in languages, for example:

- Some languages don't require a variable to be **explicitly typed**. In others **types are inferred**.
- Some languages allow the type of a variable to be **dynamically typed** and allow the type of a variable to change, just like its value.
- Operators and built-in functions are often **overloaded**. Many functional languages support **polymorphic typing**.
- The rules for **assignment-compatibility** and **type equivalence** vary between languages. Some languages also restrict how some of their types can be used.
- Many languages support some level of implicit **coercion** (**casting**), e.g. allow assigning an integer value to a floating point variable.
- Some languages allow an identifier to be used to **name several kinds of entities** (e.g. a variable, a method, a label) provided that we can tell from the context which one to use
- Some languages support **multiple sizes** for integers and floats. Some languages allow integers to be arbitrarily large.

Semantic Analysis

Ideally a programming language should have a precise and complete static semantics description that can be used as input for a semantic-analyser generator.

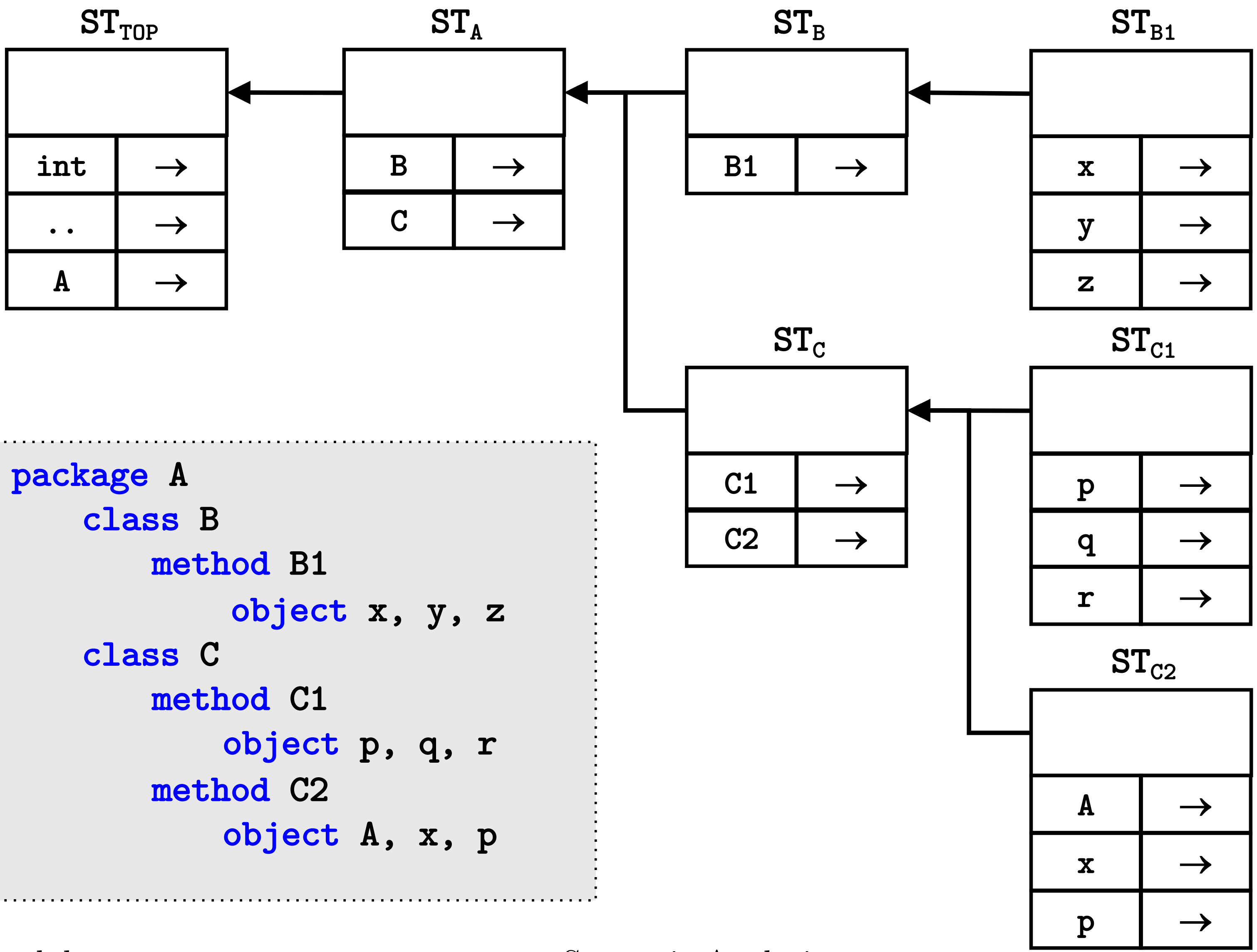
The best known automated approach to semantic-analyser generation uses a technique known as **attribute grammars** to decorate the AST nodes of a program with attributes. Attribute grammars are rarely used in practice and most semantic analysers are hand-crafted.

We'll overview the organisation and operation of a hand-crafted semantic analyser ~~and then take a brief look at attribute grammars~~.

Symbol Tables

- A primary goal of semantic analysers is to establish and check the **"long range" relationships between AST nodes**, particularly the defining and applied occurrences of identifiers.
- We could store the semantic information that the semantic analyser gathers in AST nodes, but such an approach would be slow when looking up identifiers since the AST is not organised for fast identifier lookup and programs typically have hundreds if not thousands of identifiers.
- Most semantic analysers use a lookup **dictionary** of identifiers that maps identifiers to either AST nodes or other objects holding identifier information.
- Confusingly the semantic analyser's dictionary is termed the **Symbol Table**. Note: the symbol table is not indexed on the symbols (tokens) of a program, rather it's indexed on the string name of identifier symbols (tokens). A better term would be to call it the *Identifier Table* or *Identifier Dictionary*.
- In order to support scoping we'll construct a **tree hierarchy of Symbol Tables** rather than a single Symbol Table. For an alternative approach see Appel.
- We'll also assume that identifiers are **declared before use** - this will allow us to construct the symbol table as we proceed otherwise we would need a separate pass to build the symbol table before performing identifier checks or a technique of postponing such checks.

Symbol Table Hierarchy




```
package A
    class B
        method B1
            object x, y, z
    class C
        method C1
            object p, q, r
        method C2
            object A, x, p
```

Symbol Table Implementation 1

```
class SymbolTable:

    SymbolTable encSymTable             # Ref to enclosing symbol table
    Dictionary  dict                    # Maps names to objects

    def SymbolTable(SymbolTable st):    # Create new symbol table
        dict = Dictionary();            # Initialise dictionary
        encSymTable = st;               # Reference enclosing symbol table

    def add(name, obj):
        return dict.add(name, obj)      # add name & object to dictionary

    def lookup(name):                   # return object else None if name not
        return dict.get(name)           # in dict


    # continued on next slide
```

Symbol Table Implementation 2

```
class SymbolTable:

    SymbolTable encSymTable             # Ref to enclosing symbol table
    Dictionary  dict                    # Maps names to objects
    # continued from previous slide


    def lookupAll(name):                # Lookup name in current and enclosing
        S = self                        # symbol tables
        while S != None:
            obj = S.lookup(name)
            if obj != None: return obj  # name found, return obj
            S = S.encSymTable           # name not found, move to enclosing ST
        return None                     # name not found, return None
```

Identifier Objects 1

What should an identifier entry in the symbol table refer to? One approach is to refer to AST Nodes. Another is to refer to separate **Identifier Objects** that directly hold semantic information for the identifier. We'll adopt this approach and allow both symbol table entries and AST nodes to reference identifier objects as needed. For identifier objects we'll use the following class hierarchy:

```
class IDENTIFIER { }
        # Could hold a reference to the ASTnode that declares IDENTIFIER

class TYPE     extends IDENTIFIER { }
        # Could hold attributes common to all types

class VARIABLE extends IDENTIFIER { TYPE type; }

class PARAM    extends IDENTIFIER { TYPE type; }

class SCALAR   extends TYPE { const int min;  const int max; }
```

Identifier Objects 2

Continued

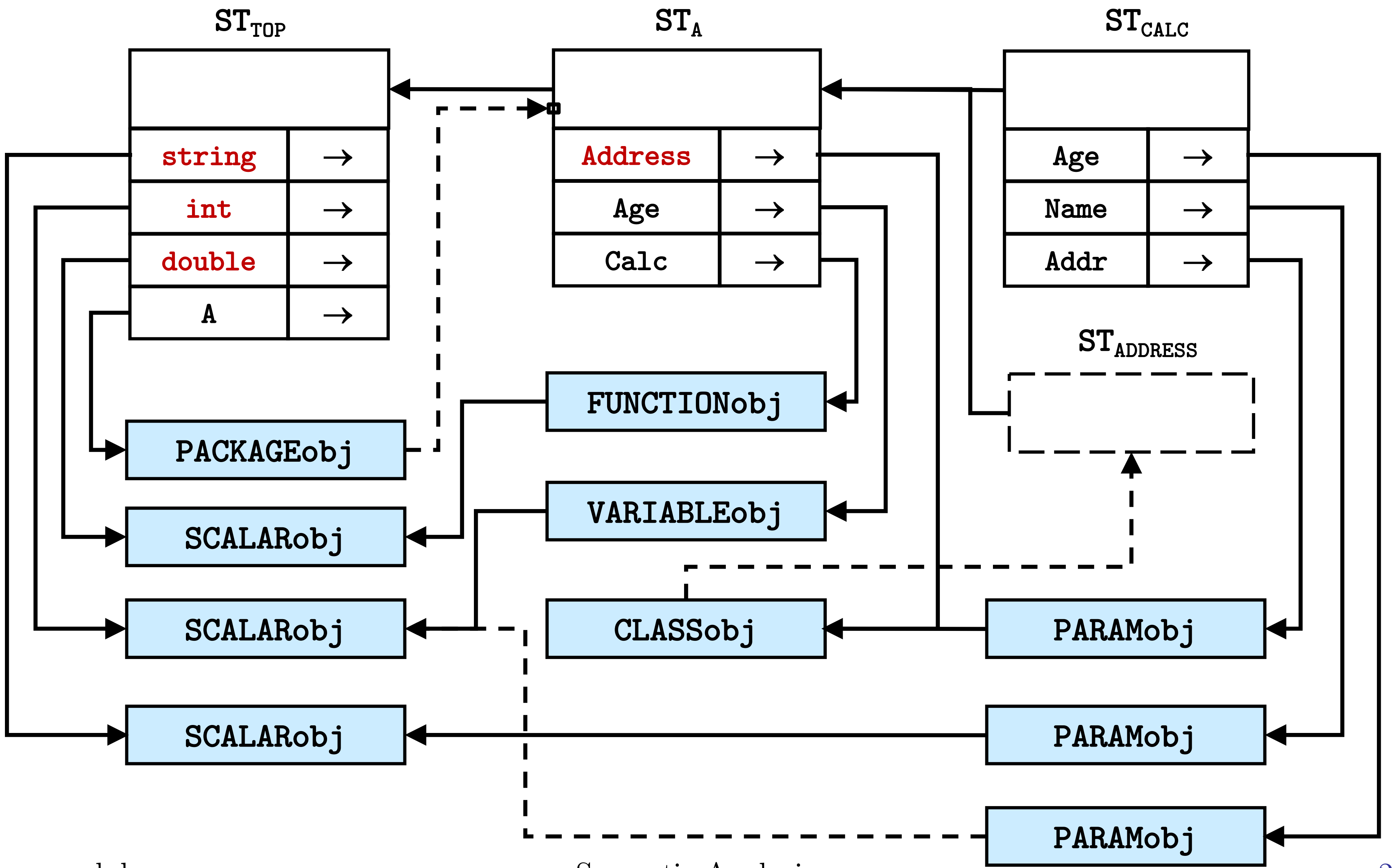
```
class ARRAY    extends ______________

                  { ________________________________ }

class CLASS    extends _______________

                  { _________________________________
                  }

class FUNCTION extends _______________

                  { _________________________________
                           }

class PACKAGE  extends ____________

                  { _________________________________
                           }
```

Identifier Objects 2

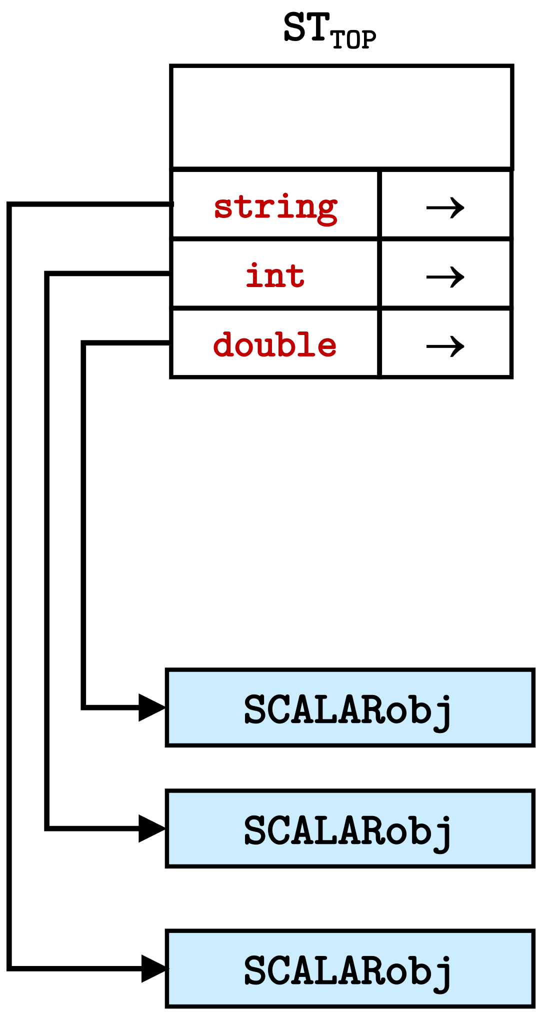
Continued

```
class ARRAY    extends TYPE
                    { TYPE elementtype; int elements; }

class CLASS    extends TYPE
                    { CLASS superclass;
                      SymbolTable symtab; }  // id's local to class

class FUNCTION extends IDENTIFIER
                    { TYPE returntype;
                      PARAM formals[];
                      SymbolTable symtab; } // id's local to function

class PACKAGE  extends IDENTIFIER
                    { SymbolTable symtab; }  // id's local to package
```

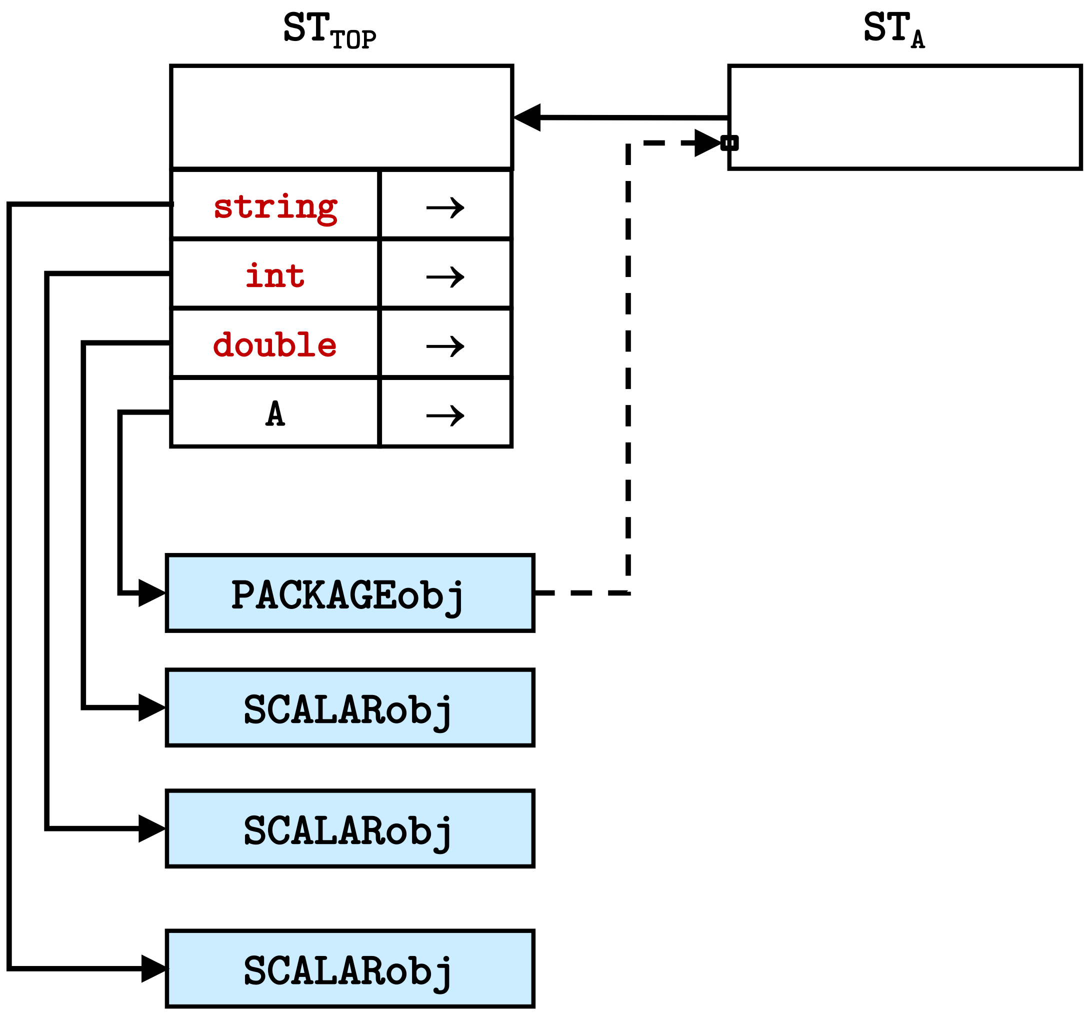
```
package A
    class Address { ... }
    int Age;
    double Calc(int Age, string Name, Address Addr) {...}
```



```
package
```

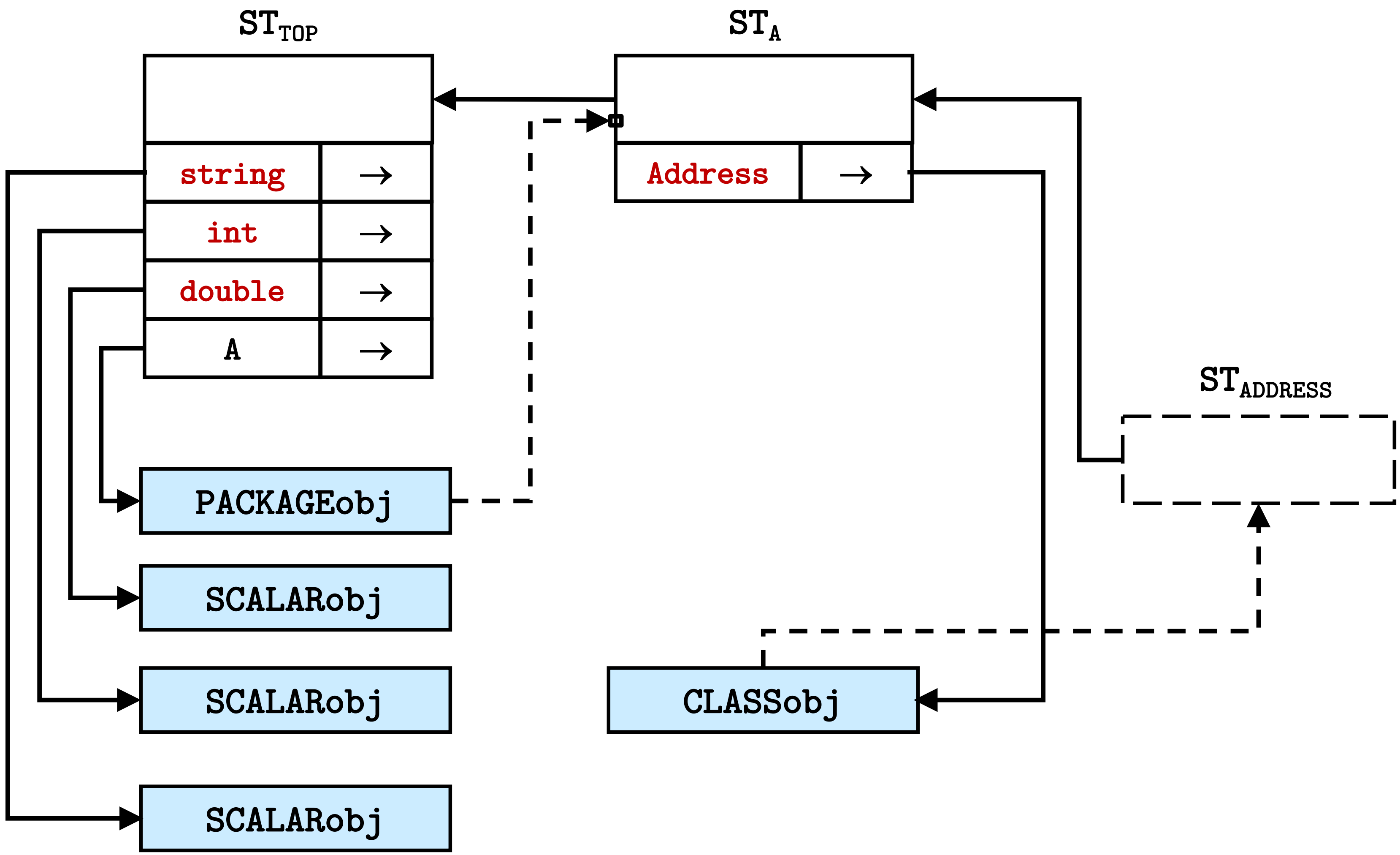


```
package A
```

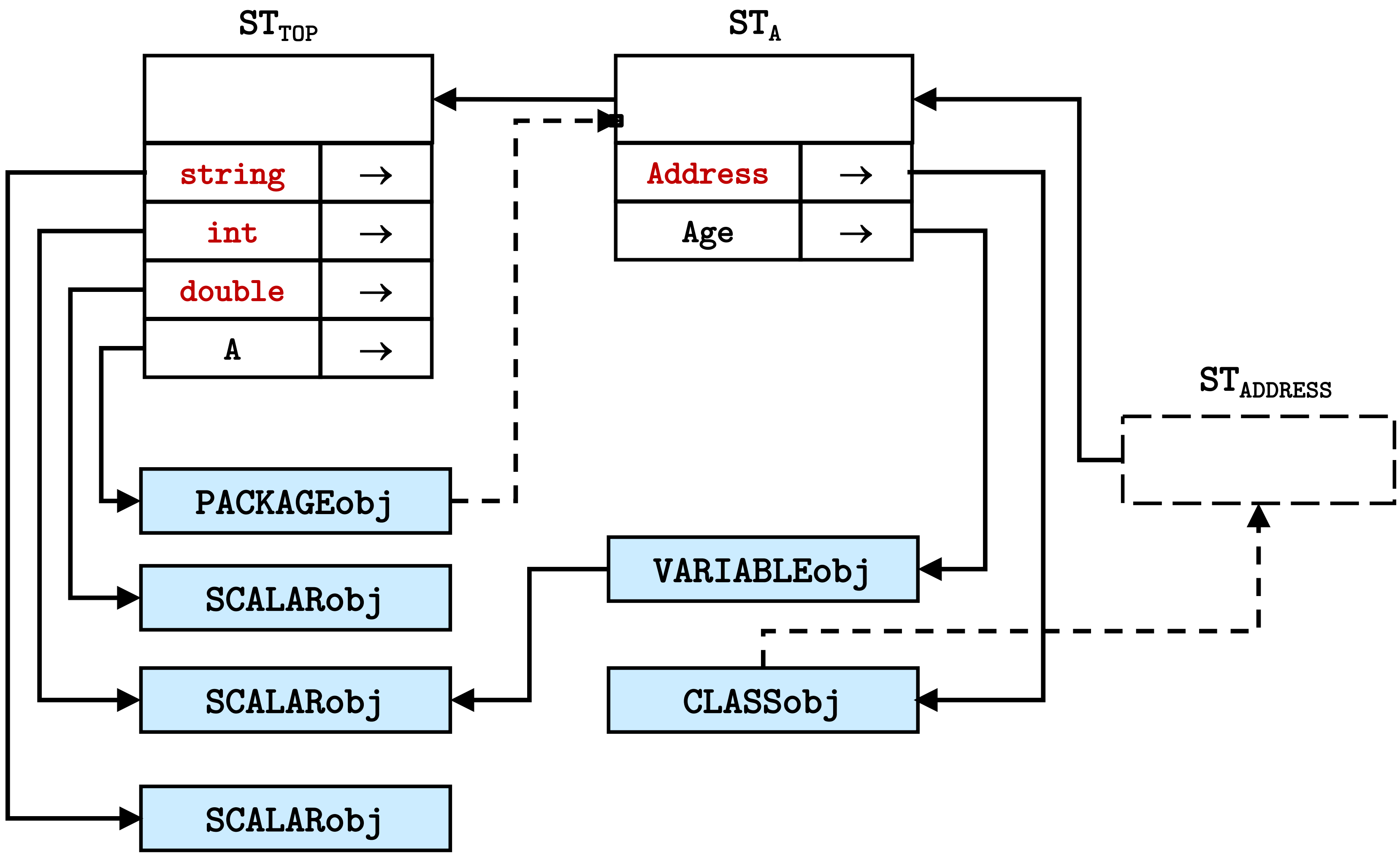


ST_{TOP}
ST_A
string →
int →
double →
A →
PACKAGEobj
SCALARobj
SCALARobj
SCALARobj

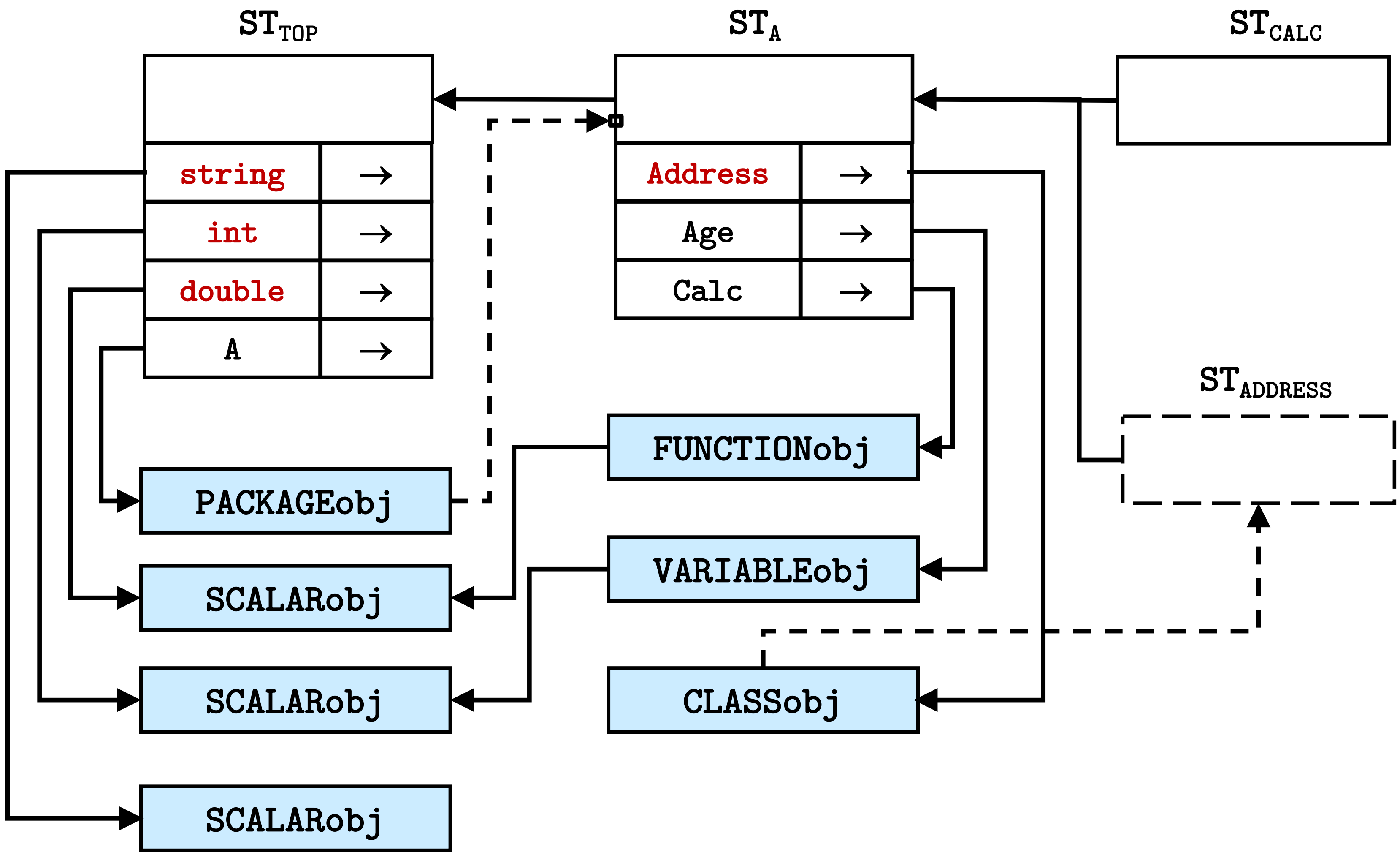
```
package A
    class Address { ... }
```



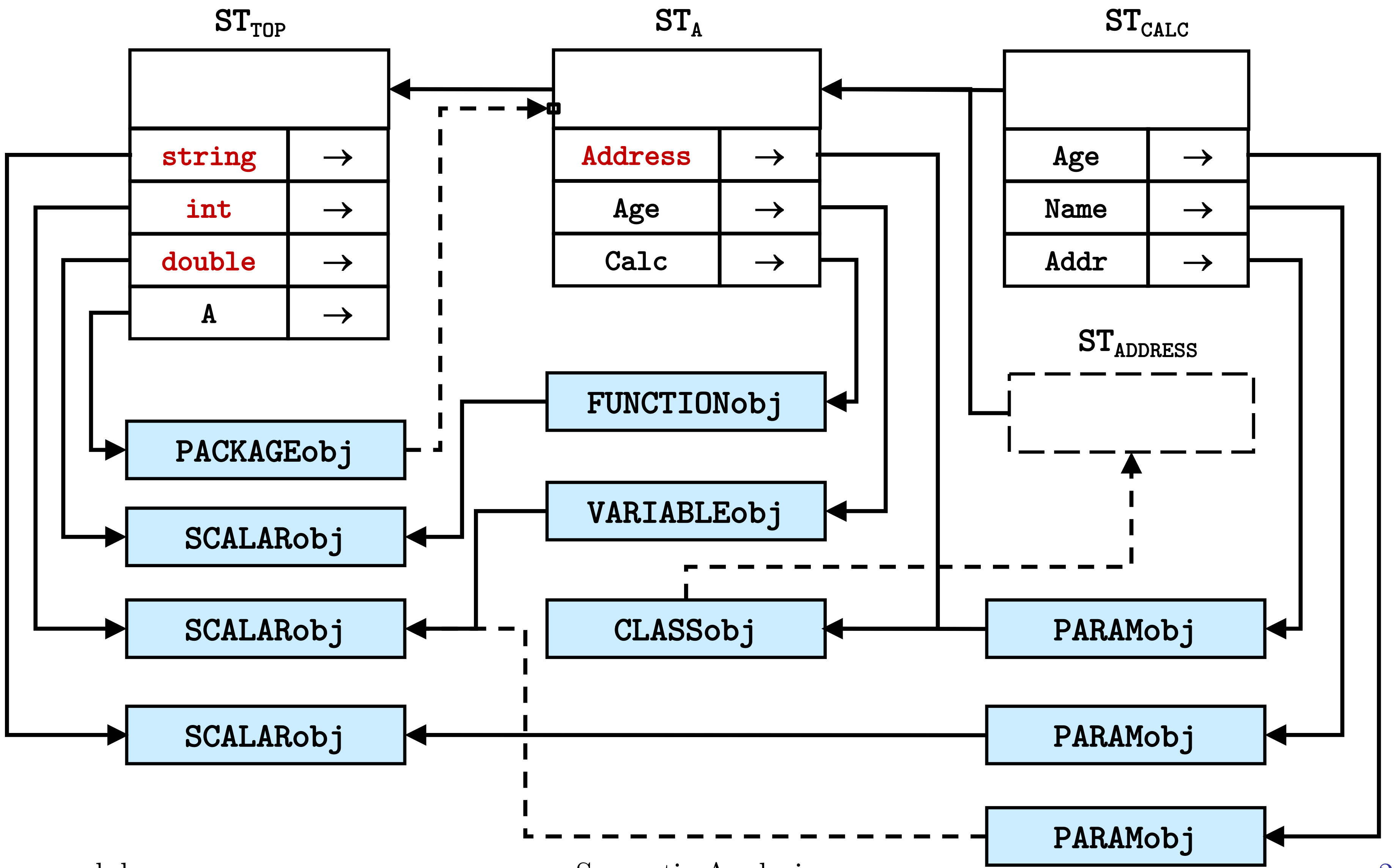
```
package A
    class Address { ... }
    int Age;
```



```
package A
    class Address { ... }
    int Age;
    double Calc(
```



```
package A
    class Address { ... }
    int Age;
    double Calc(int Age, string Name, Address Addr) {...}
```



Top-Level Symbol Table

Our initial (top-level) symbol table will be pre-loaded with all the identifier entries for globally visible identifiers, e.g. standard types, constants, functions, etc.

```
TOP_ST = SymbolTable(None)                    # Top-Level Symbol Table
ST = TOP_ST                                   # Current Symbol Table

ST.add("int",  new SCALAR(min=-2147483648, max=+2147483647))
ST.add("char", new SCALAR(min=0, max=255))
ST.add("bool", new SCALAR(min=0, max=1))
...
doubleT = ST.lookup("double")
ST.add("sin", new FUNCTION(returntype=doubleT,
                           formals=new PARAM(doubleT))
...
```

Variable Declaration

int Age

```
class VariableDeclAST:
    String typename                          # Syntactic attribute
    String varname                           # Syntactic attribute
    ---------------------------------------------------------------
    VARIABLE varObj                          # Semantic attribute

    def Check():                             # ST is the current Symbol Table
      T = ST.lookupAll(typename)
      V = ST.lookup(varname)
      if       T == None:           error("unknown type %s" % typename)
      elif ! T instanceof TYPE: error("%s is not a type" % typename)
      elif ! T.isDeclarable():  error("cannot declare %s objects" % typename)
      elif ! V == None:          error("%s is already declared" % varname)
      else varObj = new VARIABLE(T)      # varObj now holds a reference to T
           ST.add(varname, varObj)       # add to symbol table
```

Assignment

```
                                        Age = E

class AssignmentAST:
    String varname                      # Syntactic attribute
    ExpressionAST expr                  # Syntactic attribute
    ------------------------------------------------------------
    VARIABLE varObj                     # Semantic attribute

    def Check():
      V = ST.lookupAll(varname)
      expr.check()         # assume 'check' also records the type of expr
      if    V == None:     error("unknown variable %s" % varname)
      elif  ! V instanceof VARIABLE:
                           error("%s is not a variable" % varname)
      elif  ! assignCompat(V.type, expr.type):
                           error("lhs and rhs not type compatible")
      else  varobj = V
```

Function Declaration 1

double Calc(int Age, string Name, Address Addr)

```
class FunctionDeclarationAST:
    String returntypename;  String funcname;  ParameterASTlist parameters
    ----------------------------------------------------------------------
    FUNCTION funcObj            # Semantic attribute

    def CheckFunctionNameAndReturnType():    # Similar to variable decl. check
        T = ST.lookupAll(returntypename)
        F = ST.lookup(funcname)
        if      T == None:    error (“unknown type %s” % returntypename)
        elif ! T instanceof TYPE: error (“%s is not a type” % returntypename)
        elif ! T.isReturnable():
                              error (“cannot return %s objects” % returntypename)
        elif ! F == None:     error (“%s is already declared” % funcname)
        else
                funcObj = new FUNCTION(T)       # link to T and parameter list
                ST.add(funcname, funcObj)     # add F to symbol table
```

Function Declaration 2

double Calc(int Age, string Name, Address Addr)

```
class FunctionDeclarationAST:
    String returntypename;  String funcname;  ParameterASTlist parameters
    ---------------------------------------------------------------------
    FUNCTION funcObj                 # Semantic attribute

    def Check():
      CheckFunctionNameAndReturnType():     # defined on previous slide

      ST = new SymbolTable(ST)       # create and link new symbol table
      funcObj.symtab = ST

      for P in parameters:           # check parameter declarations
          P.check()
          funcObj.formals.append(P.paramObj)

      ST = ST.encSymTable            # return to enclosing symbol table
```

Function Call 1

```
                              Calc(X, Y, Z)

class CallAST:
    String funcname                                # Syntactic value
    ExpressionASTlist actuals            # Syntactic value
    ------------------------------------------------------------------
    FUNCTION funcObj                            # Semantic value

    def Check():
        F = ST.lookupAll(funcname)

        if   F == None:   error ("unknown function %s" % funcname)
        elif ! F instanceof FUNCTION:
                          error ("%s is not a function" % funcname)
        elif ! F.formals.len == actuals.len:
                          error ("wrong no. of params")

        # continued on next slide
```

Function Call 2

```
                              Calc(X, Y, Z)

class CallAST:
    String funcname                                   # Syntactic value
    ExpressionASTlist actuals          # Syntactic value
    ------------------------------------------------------------------------
    FUNCTION funcObj                              # Semantic value

    def Check():  # continued from previous slide
          else   # check parameters and set semantic value
             for K in actuals.len:
                 actuals[K].check()
                 if ! assignCompat(F.formals[K].type, actuals[K].type) :
                      error("type of func param %d incompatible with
                                     declaration" % K)
             funcObj = F
```

Note: checking of a function's returntype is handled by the enclosing expression

Type Checking

In principle the type checking in our small case study is simple, we just need to check that types `T1` and `T2` refer to the same type object, i.e. `T1` == `T2`.

```
def assignCompat(TYPE lhs, TYPE rhs):  return lhs == rhs
```

However in practice type checking is often more ad-hoc, for example, many languages allow an integer to be assigned to a double:

```
INTtype = TOP_ST.lookup("int")
DOUBLEtype = TOP_ST.lookup("double")

def assignCompat(TYPE lhs, TYPE rhs):
                 return (lhs == rhs)
                     or (lhs == DOUBLEtype and rhs == INTtype)
```

For this the code-generator will need to ensure that the **`rhs`** is properly converted from **`int`** to **`double`**. Similar checks and coercions occur in arithmetic expressions with overloaded operators.

Single Inheritance

For singly-inherited classes we can extend the type-checking function to check whether any superclass of the rhs value is assignment compatible, with the lhs for example:

```
class CLASS extends TYPE { CLASS superclass, ... }

def assignCompat(TYPE lhs, TYPE rhs):
   if   lhs == rhs:    return true
   elif rhs instanceof CLASS:
                       return assignCompat(lhs, rhs.superclass)
   else return false
```

Note: `assignCompat` can also be used in a Class declaration check to ensure that the superclass of a class is *not* also a subclass.

This kind of type checking is called name equivalence. Some languages use structural equivalence where types are considered to be the same if they have the same underlying structure, e.g. if their ASTs are identical in structure

ANTLR4 - Putting it together 1

In order to execute the semantic analysis phase we need to initialise our symbol table and any other data structures and the call the **check()** method for the root of the AST i.e. the rule corresponding to the start symbol (program).

The **check()** method is really just a visitor method so we can use any available visitor framework. However, a simple recursive descent of the tree is probably more flexible.

In ANTLR4 the parse tree classes are automatically generated. So how do we add the semantic attributes to ANTLR's classes?

ANTLR4 - Putting it together 2

For ANLTR4 there are a number of places in the ANTLR grammar that we can embed Java/Python code into the Grammar (see book for further details):

```
grammar Hello;

// add JavaDeclarations before HelloParser class
@parser::header  { JavaDeclarations }

// add JavaDeclarations to HelloParser class
@parser::members { JavaDeclarations }

// add JavaVariables to progContext class
prog locals [ JavaVariables ]
        : 'hello' ID;
...
```

Examine the ANTLR generated .java / .py files to check that the placement of your declarations is as expected.

ANTLR4 - Putting it together 3

Warning: Frowned upon programming technique.

In Python we could dynamically add attributes to ANTLR4 classes (or objects) by assignment or by using the **setattr** function e.g.

```
class ANTclass:
      x = 1

ANTclass.y = 2              # adds y=2 to ANTclass
setattr(ANTclass, z, 3)     # adds z=3 to ANTclass
print(ANTclass.x, ANTclass.y, ANTclass.z)

ANTobj = ANTclass()
ANTobj.w = 4                # adds w to object ANTobj
print(ANTobj.x, ANTobj.y, ANTobj.z, ANTobj.w)
```

Summary

Unlike lexical analysis and syntax analysis where there is more widespread agreement on the use of automated techniques, semantic analysers tend to be hand-crafted. This is partly due to the complexity and wide variation of the semantic rules for programming languages, but also because the area is somewhat neglected.

The most common automated technique is to use attribute grammars. These are "relatively" easy to write, but the underlying evaluation mechanism (dataflow execution) is not so well-known. Since the semantic rules also tend to mirror those that a hand-crafted semantic analyser would use anyway, most compiler-writers prefer to stick with the greater flexibility of a hand-crafted analyser vs. the more declarative approach of attribute grammars.

For more details on semantic analysis see [Cooper - Chapter 4, Appel - Chapters 4 & 5, Aho et al – Chapters 5 & 6]