Part VI : Type Analysis

- Type Rules
- Attributed Grammars
- Attributes
- The full specification of the context-dependent syntax of JO
- How to get from a specification to a compiler

Type Rules

- Identifier declaration is not the only thing to be checked in a compiler.
- In J0, as in most programming languages, expressions have types.
- We have to check that the types make sense.
- Examples:
  - The operands of `+` need to be integers.
  - The operands of `==` need to be of the same type. (`int, int` is OK, so is `string[], string[]` but `int, string` is not.)
  - The number of arguments passed to a function must match the number of parameters of this function.
  - Indexing `x[n]` is only legal for variables `x` of array type.
  - etc.
- How are type rules specified?

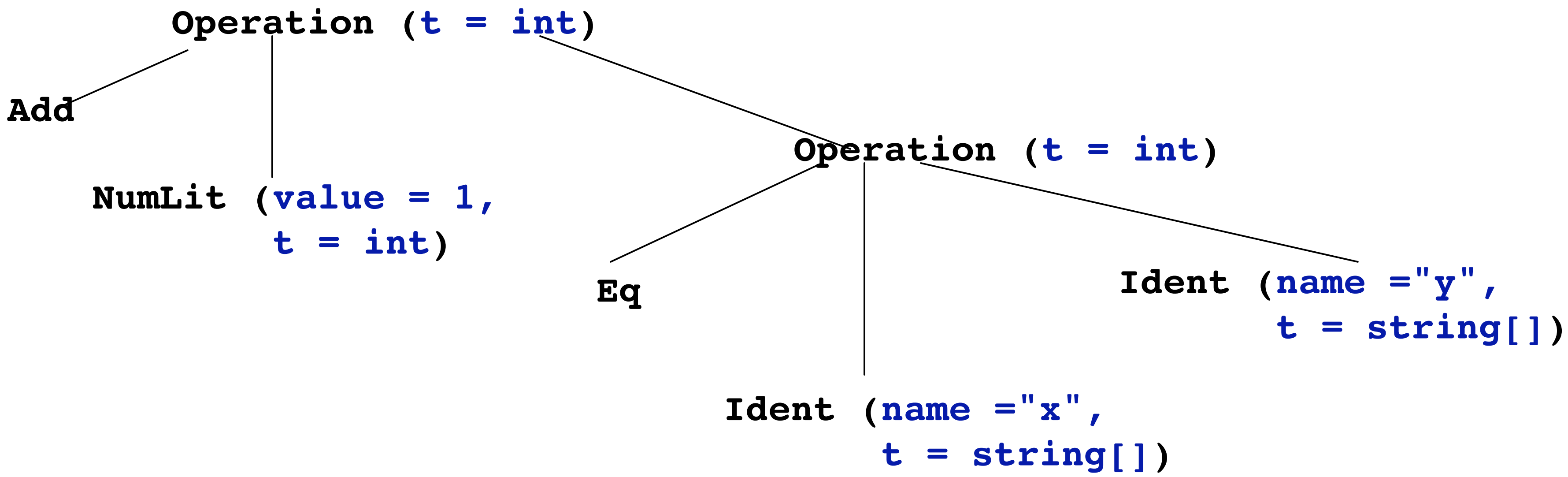
Attributed Grammars

- We augment our context free grammars with attributes and attribution rules.
- Every symbol can now be associated with attributes.
- Every production can now be associated with attribution rules which relate the attributes of the symbols in the production.
- Example:

```
E(t) = Operation Add E(t1) E(t2)      t1 = t2 = int, t = int
     | Operation Eq E(t1) E(t2)       t1 = t2, t = int
     | Ident (name)                   sym = lookup(name), t = sym.type
     | NumLit (value)                 t = int
```

Attributed Grammars (2)

- An *attribution* is an assignment of all attributes in a syntax tree that satisfy all attribution rules.
- Example : `1 + (x == y)`



Attributed Grammars (3)

- A program is legal, if
  - it is a sentence in the language given by the context-free grammar, and
  - there is an attribution for its structure tree.
- A language's full characterisation is hence given by its *context-free* syntax and its *context-dependent* syntax.
- Nothing is yet said about the *meaning* of a program, though (this is also called its *semantics*).

Attributes

- Typical attributes are:
  - The *type* of an expression
  - The *symbol* produced by a declaration
  - The *symbol table* (or: *scope*) produced by a set of declarations
- Symbol tables are often represented as a global variable rather than a set of attributes.
- This is the difference between concept and pragmatics.
- It's important to make sure that pragmatics don't destroy concepts - symbol tables can be regarded as an attribute, we just choose a more efficient centralised representation.

The Full Specification of the Context-Dependent Syntax of JO

`P`	`= ModDecl ident {D}`	"create a new outermost scope "
`D`	`= VD | FD`	
`VD`	`= VarDecl T(t) name`	"create a new symbol in current scope with given **name** and type **t**.
`FD`	`= FunDecl RT(t) name {VD} S`	"process parameters **{VD}** in a nested scope, create a new symbol in current scope with given **name** and a function type which refers to parameters and resulttype **t**.
`T(t)`	`= int`	
	`| string`	
	`|` $T(t_1)$ `[]`	
`RT(t)`	`=` $T(t_1)$	
	`| void`	

The Full Specification of the Context-Dependent Syntax of JO (2)

```
S    = VD
     | FunCall Ident(name) Es(formal)
     | Assignment E(t1) E(t2)
     | Block {S}
     | IfStmt E(t) S [S]
     | WhileStmt E(t) S
     | ReturnStmt E(t)

E(t) = Ident(name)

     | FunCall Ident(name) Es(formal)


     | Subscript E(t1) E(t2)

     | NumLit int
     | StrLit String
     | Operation UnOp E(t1)
     | Operation IntOp E(t1) E(t2)
     | Operation EqOp E(t1) E(t2)
     | NewArray T(t1) E(t2)
```

The Full Specification of the Context-Dependent Syntax of JO (3)

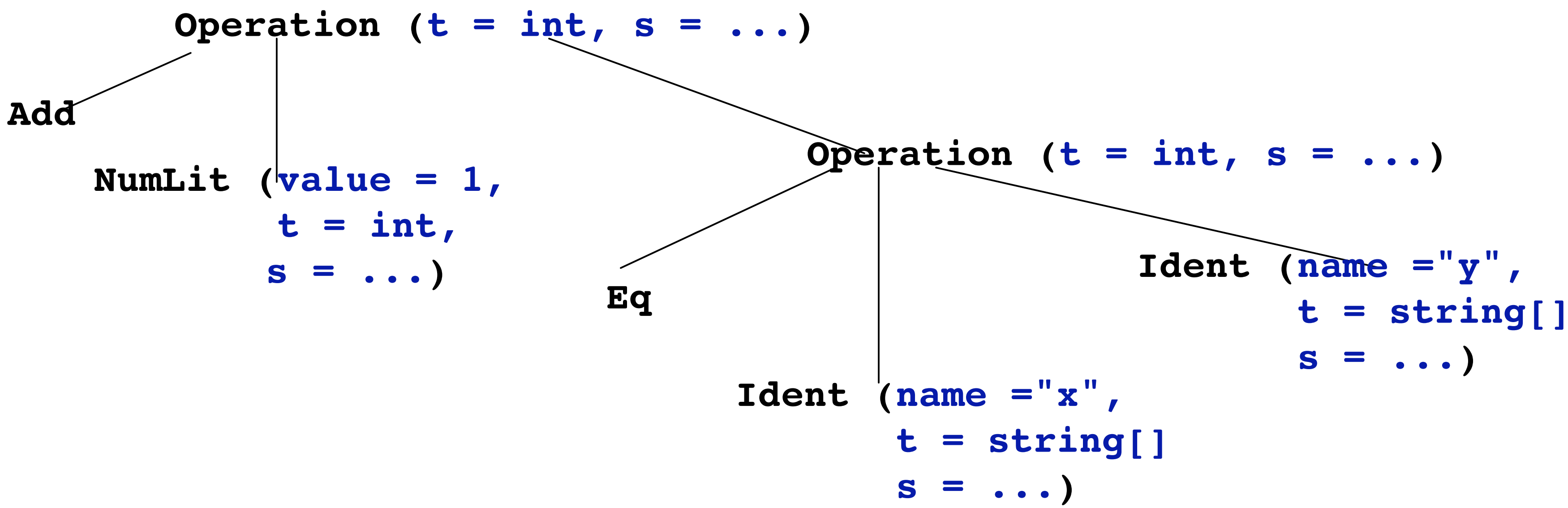
```
Es(formal) = ∈
       | E(t), Es(formal1)

UnOp   = Neg | Not
EqOp   = Eq | Ne
IntOp = And | Or | Add | Sub | Mul |
        Div | Mod | Lt | Gt | Le | Ge
```

How to get from a Specification to a Compiler

- Instead of *guessing* attributes and *checking* that they satisfy the attribution rules, we have to *compute* them.
- Attributes are usually computed from the values of other attributes.
- Important: Attributes should be assigned only once!
- Attributes can be distinguished by how they ``flow'' through the structure tree.
  - Some flow up the tree - these are *synthesised* attributes.
  - Some flow down the tree - these are *inherited* attributes.
- In a compiler, synthesised attributes are represented as return types (or, alternatively: output parameters in C/C++) of the tree visitors.
- Inherited attributes are represented as (input) parameters to the tree visitors.

Example



- s is inherited
- t is synthesized
- name, value are intrinsic

Representing Attributes

- Some attributes are required to be present in later phases of thecompiler -- they are *persistent.*
- Other attributes are required only for type checking -- they are *transient*.
- Persistent attributes can be stored as additional fields in the abstract syntax tree.
- Transient attributes are parameters and results of visitor methods.
  - Synthesized, transient = visitor results
  - Inherited, transient = visitor parameters.
- Some attributes can alternatively be represented as global variables. This is simpler if these attributes change rarely.

Example

```
E(t) = Operation IntOp E(t1) E(t2)
     | Operation EqOp E(t1) E(t2)
     | Operation UnOp E(t1)
     | Ident(name)
```

- There are the following attributes:
  - **type** : synthesized, persistent
  - **sym** : intrinsic, persistent
  - **scope**: inherited, transient
- Hence we need an analyzer visitor which is structured as follows.

```
public class Analyzer implements Tree.Visitor {

    private Scope scope;

    Analyzer(Scope scope) {
      this.scope = scope;
  }

    public Type analyze(Tree tree, Scope scope) {
      tree.apply(new Analyzer(scope));
      return tree.type;
    }

    public void caseOperation(Operation tree){
      if (tree.operator == Not ||
          tree.operator == Neg) {
        Type t = analyze(tree.left, scope);
        if (!t.sameType(Type.intType)) {
          error(tree.pos, "integer expected, but "+
                Tree.type + " found");
          tree.type = Type.BAD; // error type
        } else {
          tree.type = Type.intType;
        }
      } else { …
      }
    }

    public void caseIdent(Ident tree) {
      tree.sym = scope.lookup(tree.name)
      if (tree.sym == null) {
        error(tree.pos, tree.name + "
              undefined")
        tree.type = Type.BAD;                    }
      else {                              tree.type =
sym.type;
      }
    }

    ...

}
```

Optimizations :

1. Creating a new visitor every time a node is visited takes time. Efficiency improvement if current visitor is reused.

```
public Type analyze(Tree tree, Scope scope) {
   Scope scope1 = this.scope;
   this.scope = scope;
   tree.apply(this);
   this.scope = scope1;
   return tree.type;
}
```

2. The `scope` attribute varies only infrequently.
   ⇒ can use a more efficient analyze method which does not save/change/restore the scope.

```
public Type analyze(Tree tree) {
   tree.apply(this);
}
```

Optimizations (2)

3. Checking types and reporting errors if mismatch occurs often and is fairly tedious.
   ⇒ Create a helper function `checkType`

```
public Type checkType(int pos, Type found, Type required) {
   if (found.sameType(required)) return found;
   error(pos, "type error: " + required + " required" +
              " but " + found + " found");
   return Type.BAD;
}
```

Then `caseOperation` could be written as follows.

```
public void caseOperation(Operation tree) {
   if (tree.operator == Not || tree.operator == Neg) {
     tree.type = checkType(analyze(tree.left),Tree.intType);
   } else ...
}
```

Summary

- Name analysis and type checking are some of the most complicated tasks of a compiler.
- Tasks: (1) Check that given AST is legal according to the rules of the language (context-dependent syntax).
  (2) Determine certain attributes of trees (such as: **`type, sym`**), which are needed in later phases.
- Context-dependent syntax can be specified with an attribute grammar.
- Attributes are fields of tree nodes, computed by attribution rules.
  - three subclasses: *inherited, synthesized, intrinsic*.
- Implementation: Visitor methods over trees. Attributes are represented as
  - fields of the tree, or
  - parameters or results of the visitor method, or
  - fields of the visitor class.

What 's to come after the break :

- Code generation.
  - What are JVM bytecodes?
  - How are J0 constructs mapped into bytecodes?
  - How to structure a code generator.
- Run-time organization
  - Garbage collectors

The Full Specification of the Context-Dependent Syntax of JO

`P`	`= ModDecl ident {D}`	"create a new outermost scope "
`D`	`= VD	FD`	
`VD`	`= VarDecl T(t) name`	"create a new symbol in current scope with given `name` and type `t`.
`FD`	`= FunDecl RT(t) name {VD} S`	"process parameters `{VD}` in a nested scope, create a new symbol named `currentFun` in current scope with given `name` and a function type which refers to parameters and resulttype `t`.
`T(t)`	`= int`	`t = Type.intType`
	`| string`	`t = Type.stringType`
	`| T(t1) []`	`t = new Type.ArrayType`
`RT(t)`	`= T(t1)`	`t = t1`
	`| void`	`t = Type.voidType`

The Full Specification of the Context-Dependent Syntax of JO (2)

```
S    = VD
     | FunCall Ident(name) Es(formal)    "see under E », with t = Type.voidType
     | Assignment E(t1) E(t2)            t1 = t2
     | Block {S}                         "open new local scope"
     | IfStmt E(t) S [S]                 t = Type.intType
     | WhileStmt E(t) S                  t = Type.intType
     | ReturnStmt E(t)                   t = currentFun.type.resType

E(t) = Ident(name, sym)                  sym = lookup(name)
                                         t = sym.type
     | FunCall Ident(name) Es(formal)    sym = lookup(name)
                                         sym.type instanceof FunType
                                         formal = sym.type.params
                                         t = sym.type.resType
     | Subscript E(t1) E(t2)             t1 instanceof ArrayType
                                         t2 = int, t = t1.elemType
     | NumLit int                        t = Type.intType
     | StrLit String                     t = Type.stringType
     | Operation UnOp E(t1)              t = t1 = Type.intType
     | Operation IntOp E(t1) E(t2)       t = t1 = t2 = Type.intType
     | Operation EqOp E(t1) E(t2)        t1 = t2, t= Type.intType
     | NewArray T(t1) E(t2)              t2 = int, « t = t1[] »
```

The Full Specification of the Context-Dependent Syntax of JO (3)

```
Es(formal) = ∈                          formal = null
       | E(t), Es(formal1)              formal.type = t, formal1 = formal.next

UnOp   = Neg | Not
EqOp   = Eq | Ne
IntOp  = And | Or | Add | Sub | Mul |
         Div | Mod | Lt | Gt | Le | Ge
```