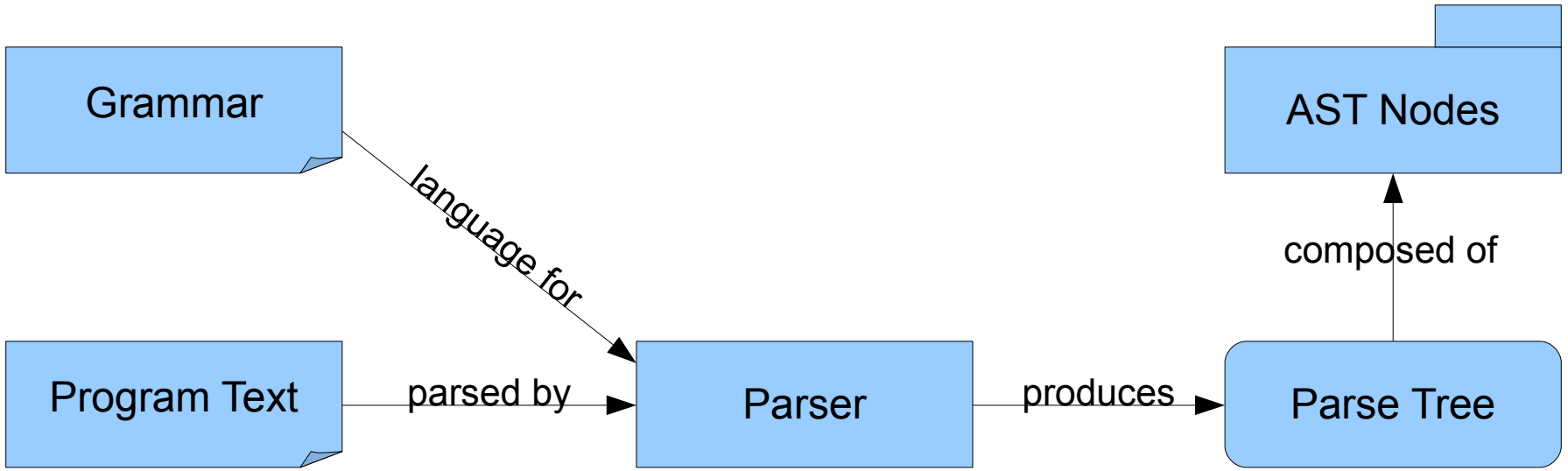
Generating Programs with Grammars and ASTGen

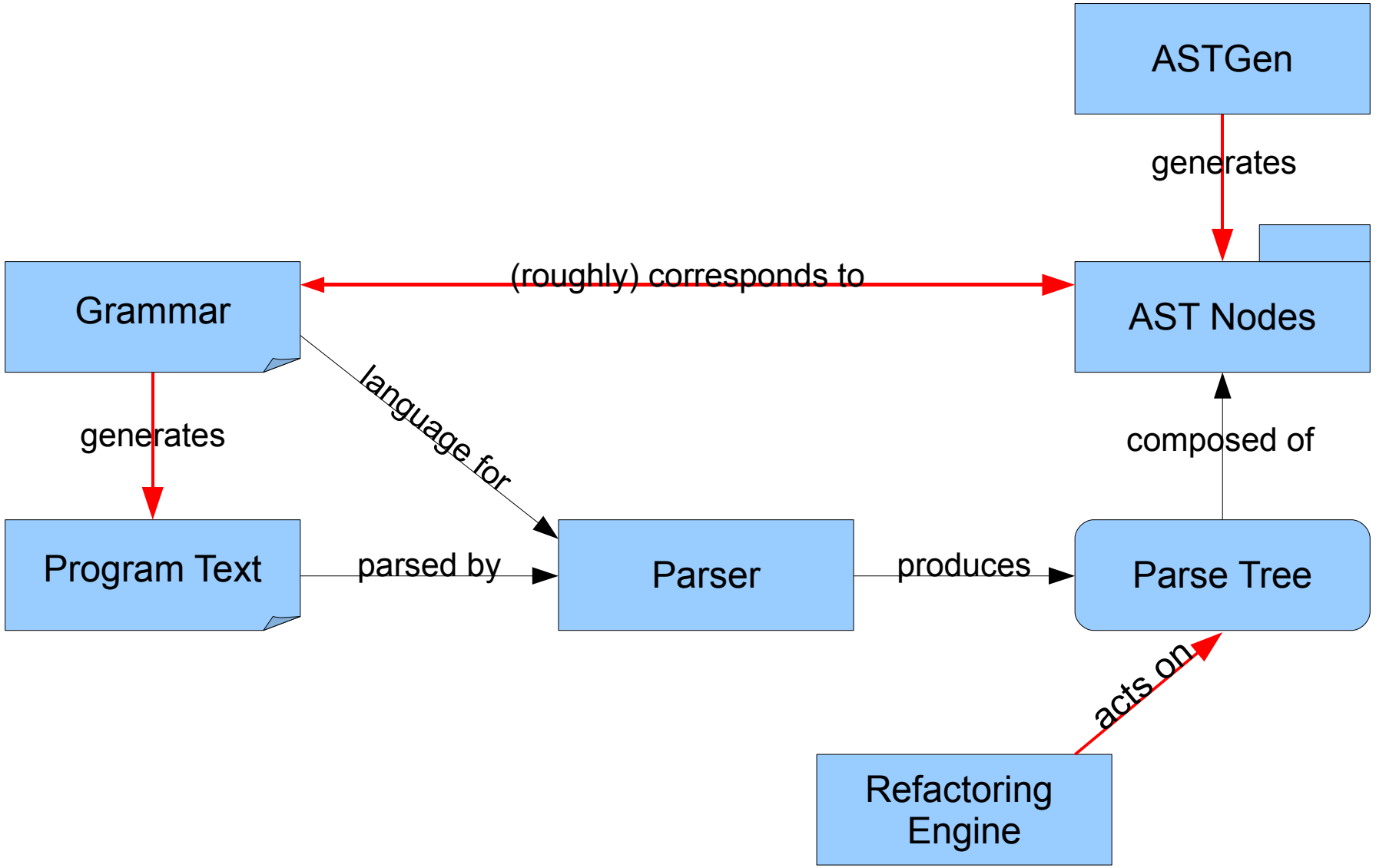
598DM Guest Lecture
Brett Daniel

Tuesday, March 11, 2008

Overview

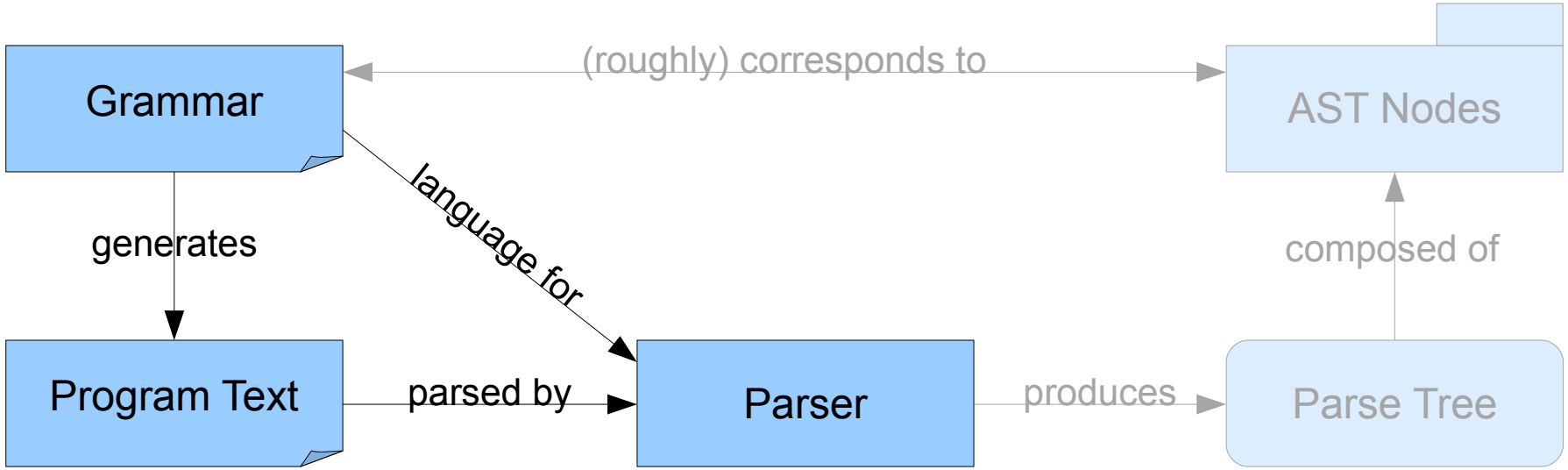

Grammar
language for
AST Nodes
composed of
Program Text
parsed by
Parser
produces
Parse Tree

Overview

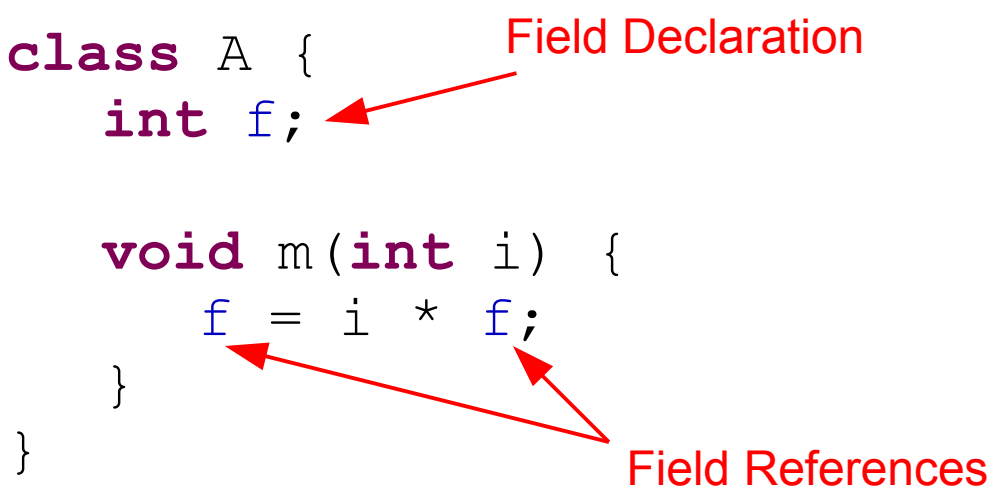

ASTGen
generates
Grammar
(roughly) corresponds to
AST Nodes
generates
language for
composed of
Program Text
parsed by
Parser
produces
Parse Tree
acts on
Refactoring Engine

Grammar-Based Testing

- Produce many programs from grammar
- Used to test parser/compiler



Example: Fields



Field Declaration Grammar

```
FieldDeclaration ::= Modifier Type Identifier “;”

Modifer ::= “public”
          | “protected”
          | “private”
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= “int”
            | “boolean”

Identifier ::= [a-zA-Z_]+
```

Terminal Symbol Coverage

- The set of test inputs contains each terminal symbol

```
FieldDeclaration ::= Modifier Type Identifier “;”

Modifer ::= “public”
          | “protected”
          | “private”
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= “int”
            | “boolean”

Identifier ::= [a-zA-Z_]+
```

Terminal Symbol Coverage

- The set of test inputs contains each terminal symbol

```
FieldDeclaration ::= Modifier Type Identifier ";"

Modifer ::= "public"
          | "protected"
          | "private"
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= "int"
            | "boolean"

Identifier ::= [a-zA-Z_]+
```

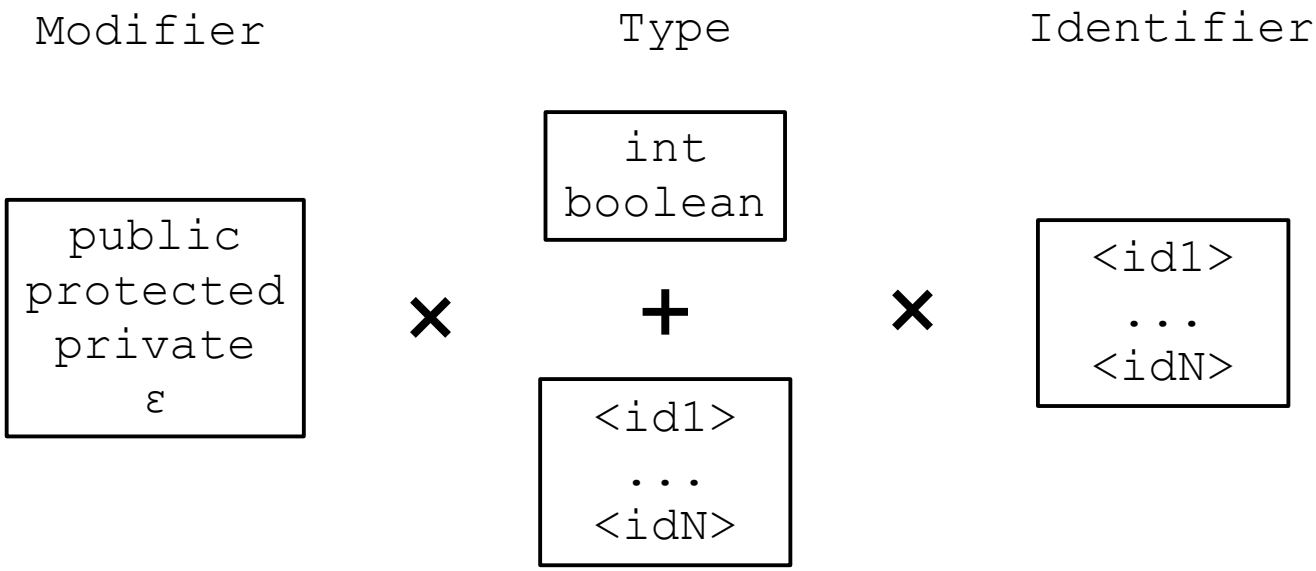
Production Coverage

- The set of test inputs contains each production

```
FieldDeclaration ::= Modifier Type Identifier ";"

Modifer ::= "public"
          | "protected"
          | "private"
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= "int"
            | "boolean"

Identifier ::= [a-zA-Z_]+
```

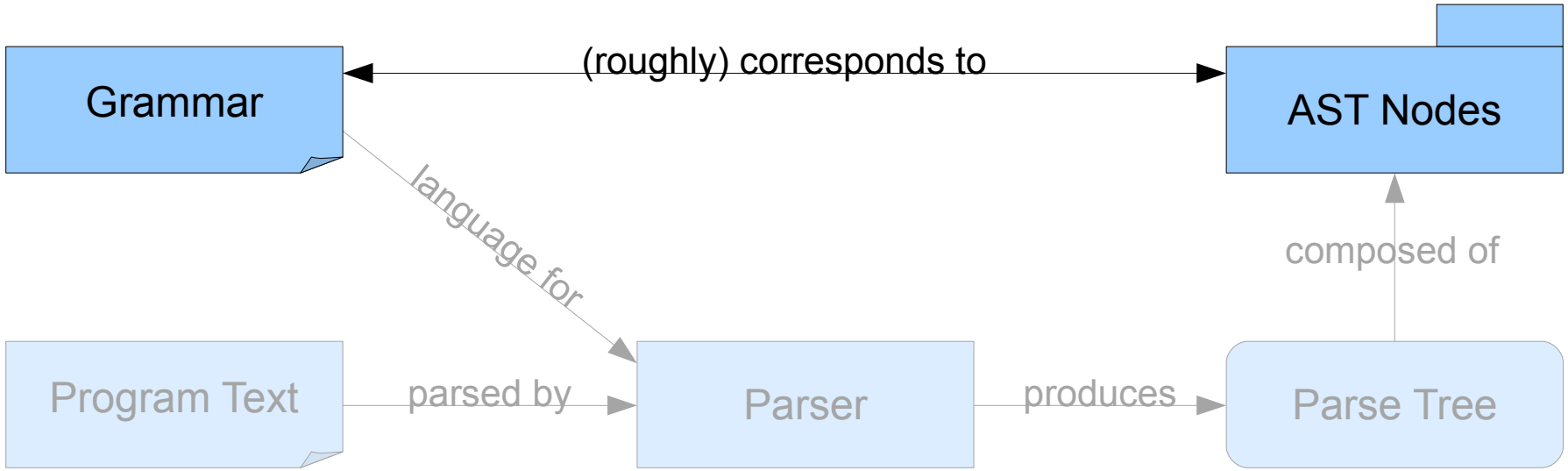
Derivation Coverage

- The set of test inputs contains every possible string

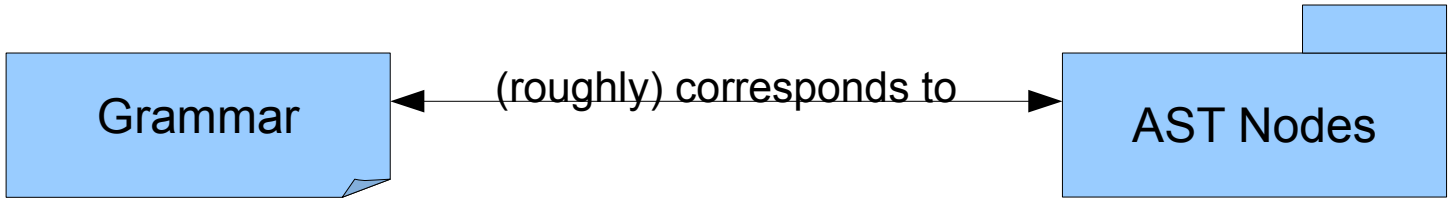


Grammars and ASTs

- Grammar roughly corresponds to AST class structure



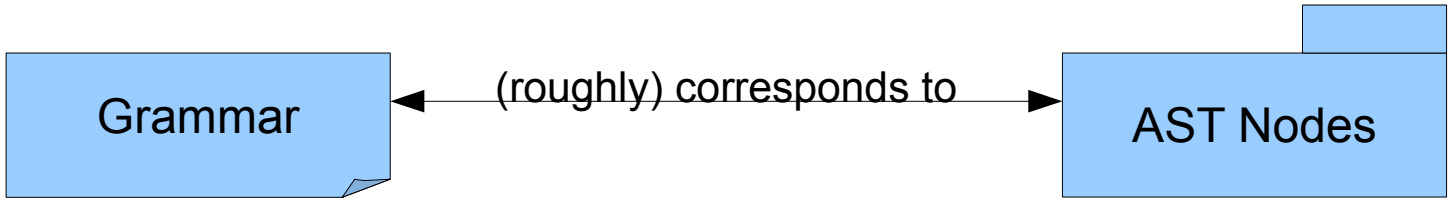
Grammars and ASTs



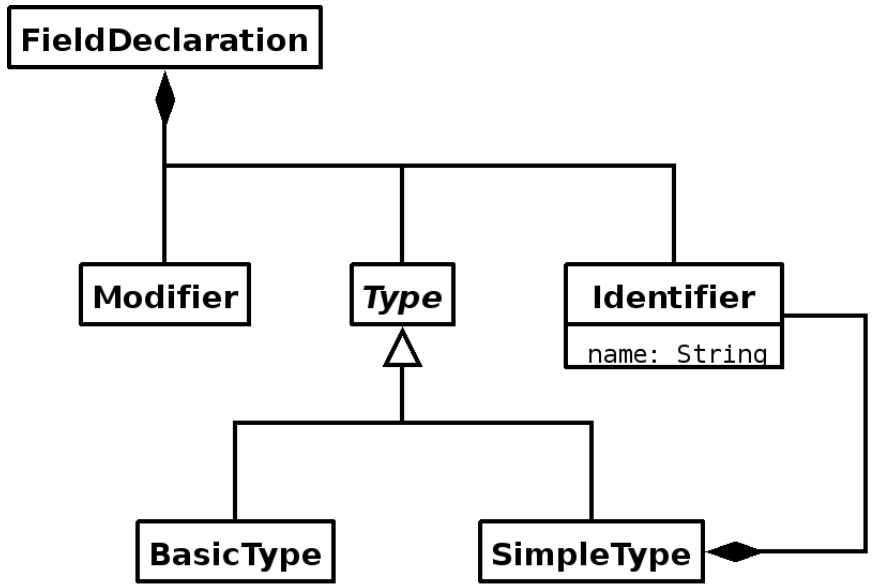

```
FieldDeclaration ::=
  Modifier Type Identifier ";"

Modifer ::= "public"
          | "protected"
          | "private"
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= "int"
            | "boolean"

Identifier ::= [a-zA-Z_]+
```

?

Grammars and ASTs



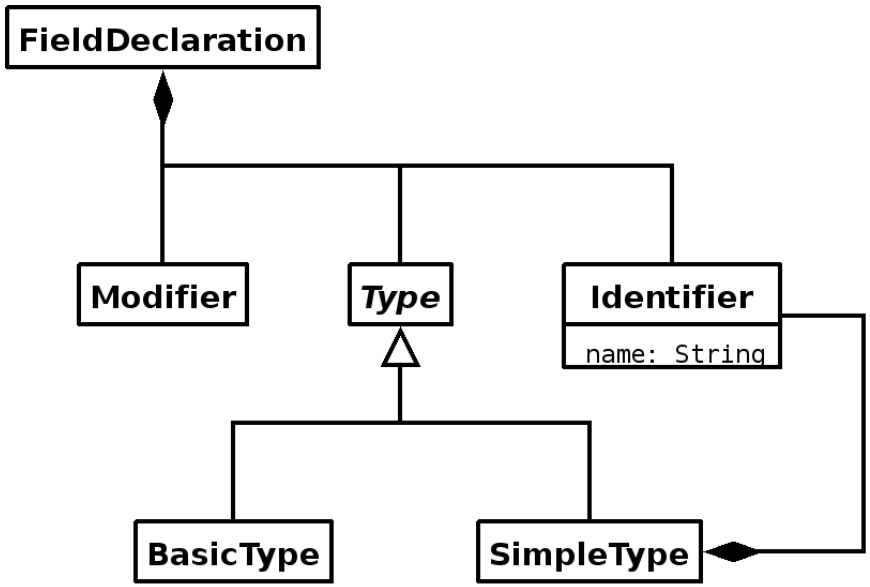

```
FieldDeclaration ::=
  Modifier Type Identifier ";"

Modifer ::= "public"
          | "protected"
          | "private"
          | ε

Type ::= BasicType
       | Identifier

BasicType ::= "int"
            | "boolean"

Identifier ::= [a-zA-Z_]+
```



Abstract Grammars

- Grammar used to generate AST class and composition structure

```
FieldDeclaration ::=
  Modifier Type Identifier;

Modifer;

abstract Type;
BasicType : Type;
SimpleType : Type ::= Identifier;

Identifier ::= <Name:String>;
```



What About Field References?

- Difficult to define grammar for all types of field references

```
class A {
    int f;

    void m(int i) {
        f = i * f;
    }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

What About Field References?

- Difficult to define grammar for all types of field references
  - Want both read and write references
  - Expressions depend on field name
  - Productions spread across language grammar
  - Multiple classes, inheritance relationships, containment, etc.
- Alternative: “Extended Grammars”

Generating ASTs

- ASTGen generators produce many AST Nodes
- Combining generators produces many parse trees

Generators

- Normal Java Classes

```
ModifierGenerator modGen =
  new ModifierGenerator(
    Modifier.PUBLIC,
    Modifier.PRIVATE);
```

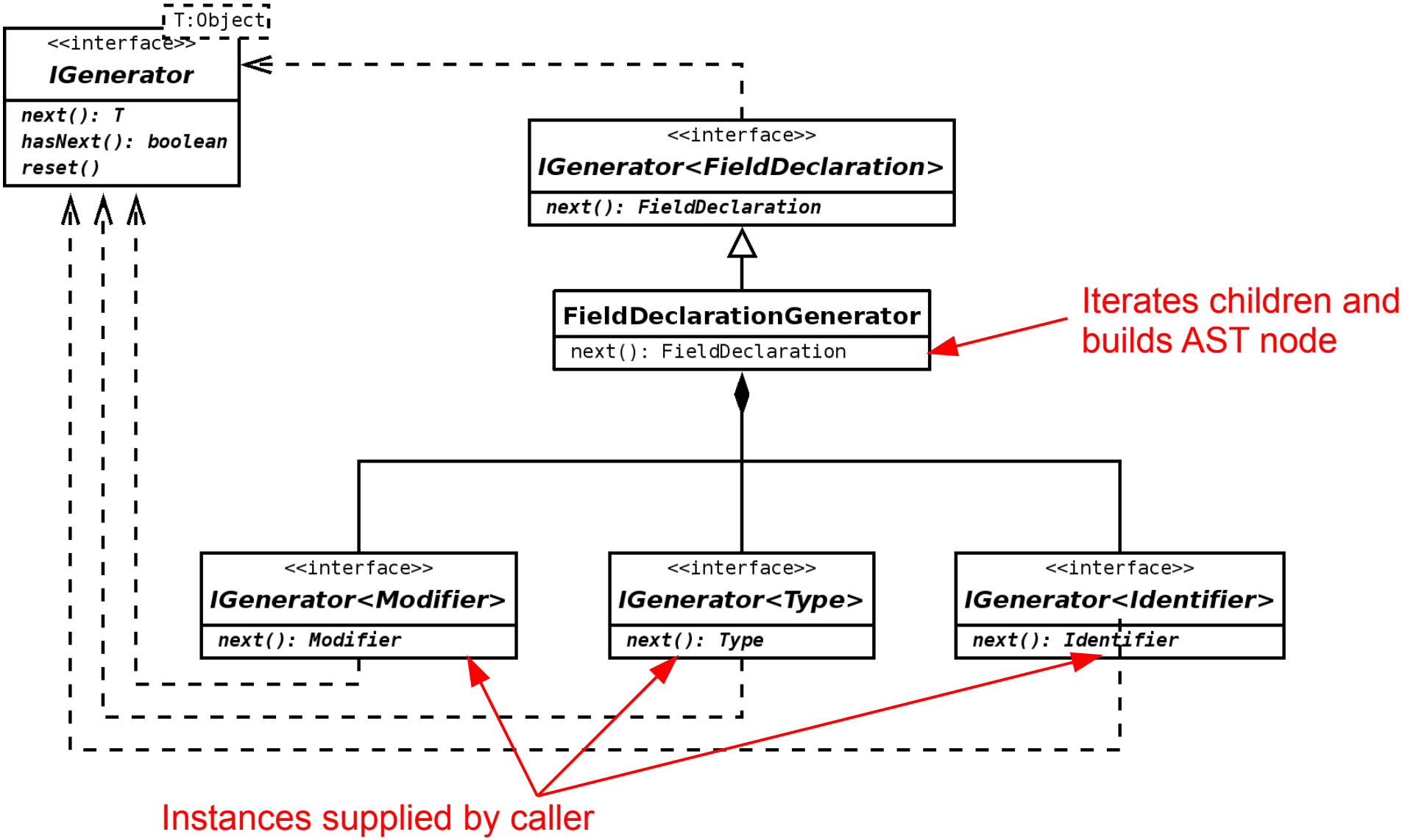
- Produce values when prompted

```
Modifier = modGen.next();
```

Generator Characteristics

- Iterative
  - Act like collection of AST fragments
  - Internally, produce fragments lazily
- Composable
  - Tester creates complex generators by reusing simpler parts
  - Generators contain code to combine AST fragments
- Bounded-Exhaustive
  - Achieves derivation coverage within bounds

Field Declaration Generator



Field Declaration Generator

```
// define child generators
ModifierGenerator modGen =
  new ModifierGenerator(Modifier.PUBLIC, Modifier.PRIVATE);

IGenerator<Type> typeGen =
  new Chain<Type>(BasicType.INT, new SimpleType("Object"));

IdentifierGenerator idGen =
  new IdentifierGenerator("f", "g");

// compose generator
FieldDeclarationGenerator fieldDeclGen =
  new FieldDeclarationGenerator();
fieldDeclGen.setModifierGenerator(modGen);
fieldDeclGen.setTypeGenerator(typeGen);
fieldDeclGen.setIdentifierGenerator(idGen);

// loop over values
for(FieldDeclaration fieldDecl : fieldDeclGen) {
  System.out.println(fieldDecl);
}
```

Complex Generator

- Double-class field reference generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

…

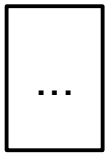
Complex Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```



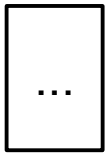
Complex Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```



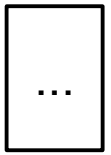
Complex Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```



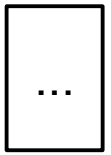
Complex Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

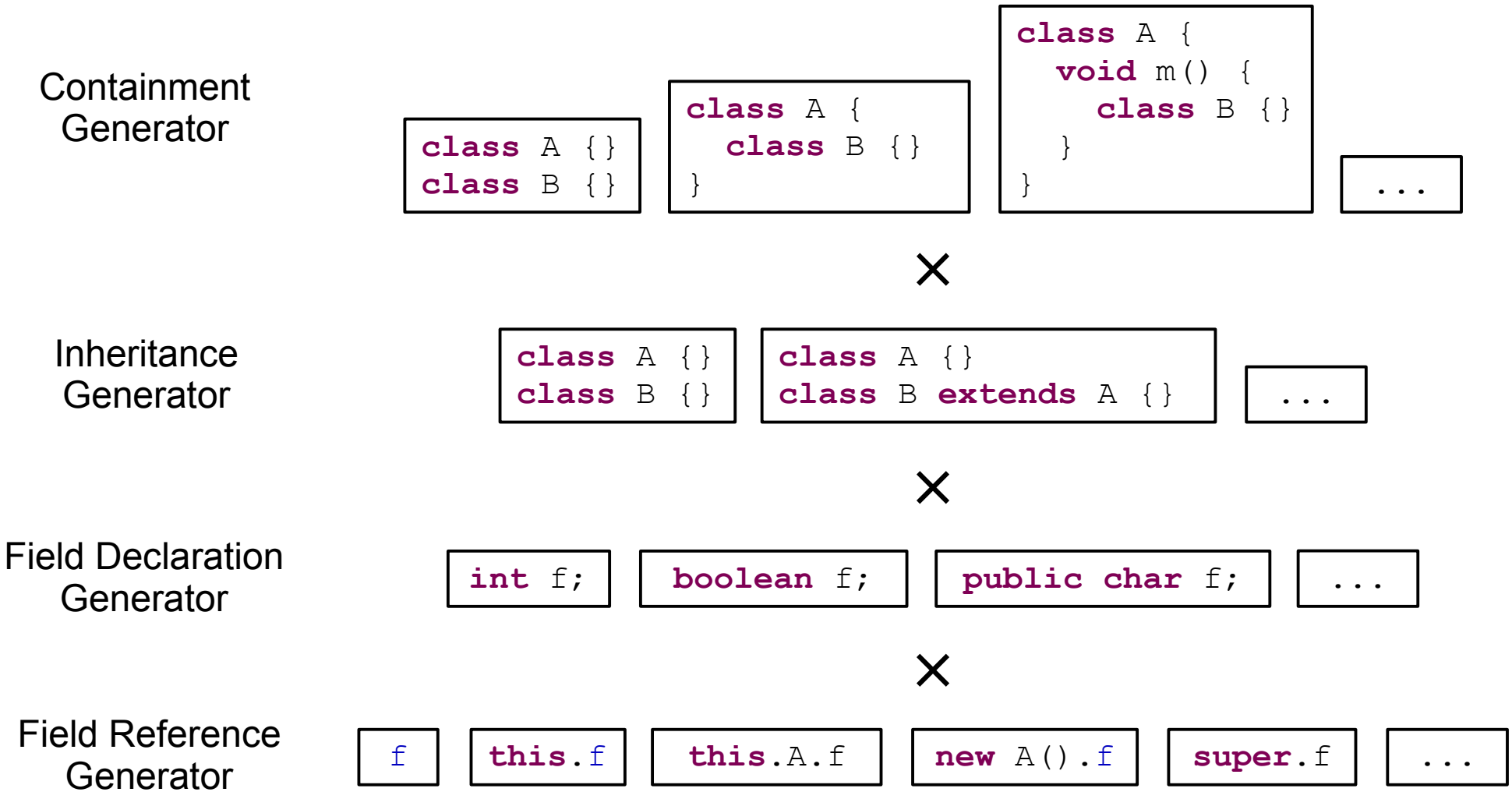
```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

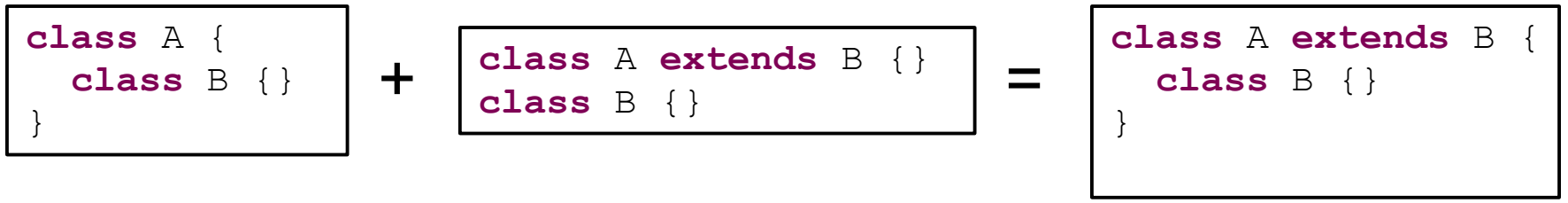


Composing Generators



Invalid Combinations

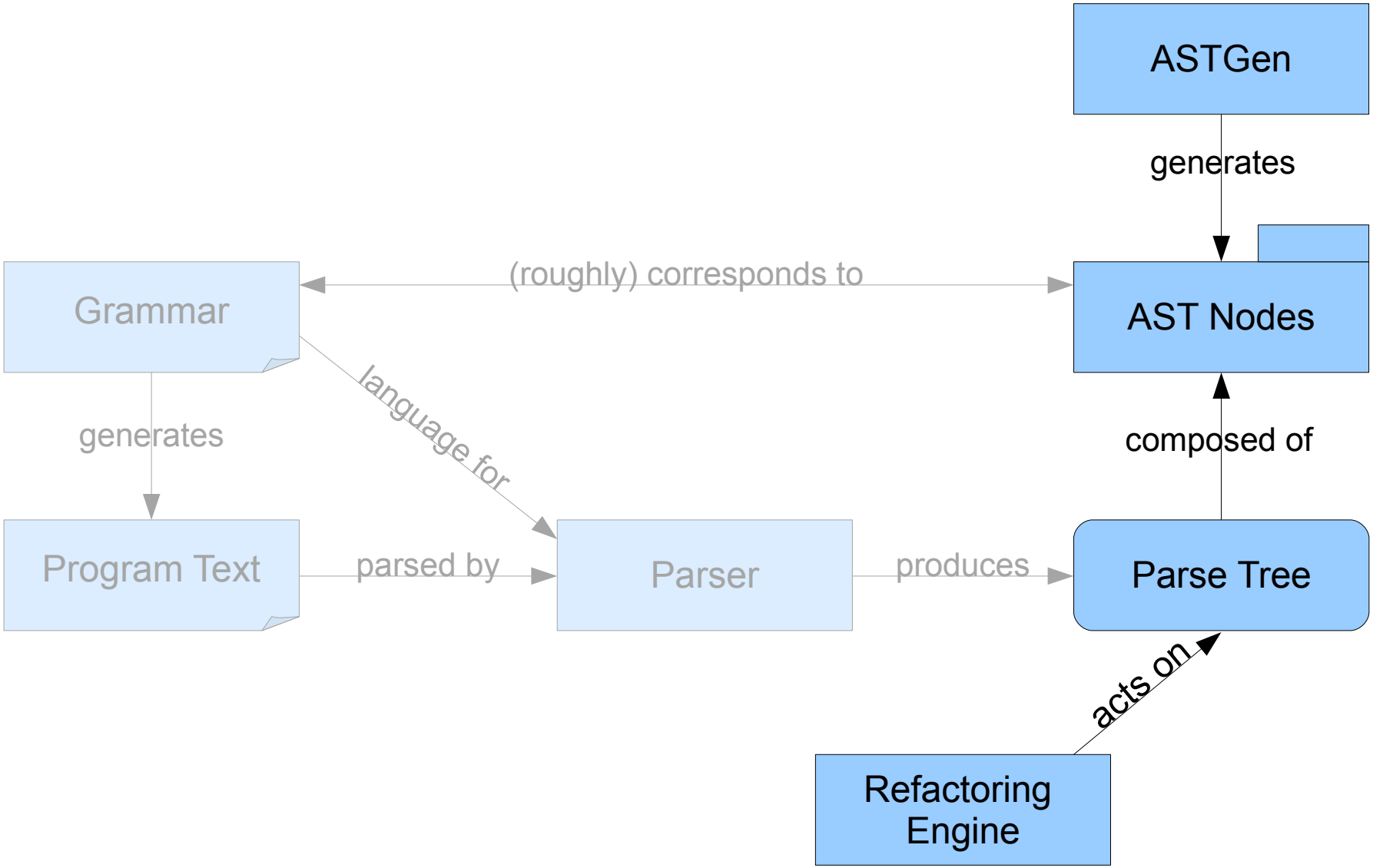
- Composition may be invalid



- Three solutions
  - Tester writes filter that verifies values
  - Dependent generators

  - Delegate to compiler

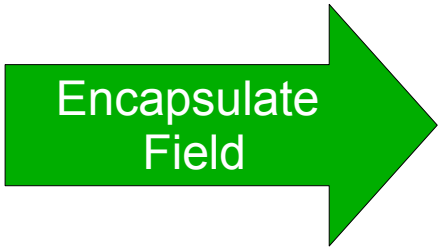
Testing Refactoring Engines



Example: Encapsulate Field

Replaces all field reads and writes with accesses through getter and setter methods

```
class A {
    int f;

    void m(int i) {
        f = i * f;
    }
}
```




```
class A {
  5 private int f;

    void m(int i) {
        setF(i * getF());
    }   4        3

  2 void setF(int f) {
        this.f = f;
    }

  1 int getF() {
        return f;
    }
}
```

Testing Encapsulate Field

```
String fieldName = "f";
IGenerator<Program> testGen =
  new DoubleClassFieldReferenceGenerator(fieldName);

for (Program in : testGen) {
  Refactoring r = new EncapsulateFieldRefactoring();
  r.setTargetField(fieldName);

  Program out = r.performRefactoring(in);
  checkOracles(out);
}
```

Oracles

- Check that the program was refactored correctly
- Challenges
  - Don't know expected output
  - Semantic equivalence is undecidable
  - Need to verify that correct structural changes were made

Oracles

Oracles

- **DoesCrash**: Engine throws exception
- **DoesNotCompile**: Refactored program does not compile
- **WarningStatus**: Engine can(not) perform refactoring
- **Inverse**: A refactoring is not undone by its inverse
- **Custom Structural**: Check for desired structural changes
- **Differential**: Perform refactoring in both Eclipse and NetBeans and compare result

Case Study

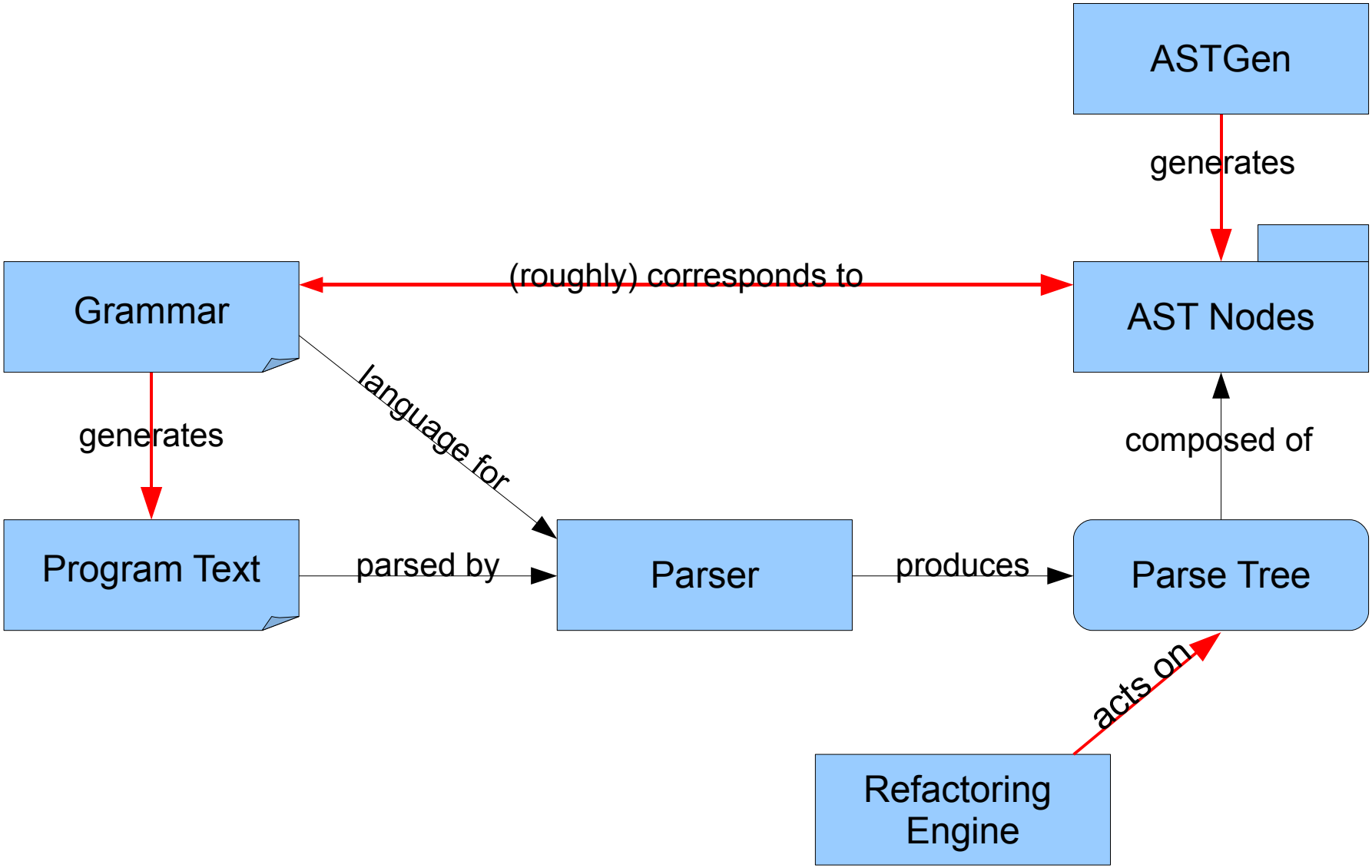
- Tested Eclipse and NetBeans
- Eight refactorings
  - Target field, method, or class
- Wrote about 50 generators
- Reported 47 new bugs
- Compared effectiveness of oracles

Results

- 47 new bugs reported
  - 21 in Eclipse: 20 confirmed by developers
  - 26 in NetBeans: 17 confirmed, 3 fixed, 5 duplicates, 1 won't fix
  - Found others, but did not report duplicate or fixed

http://mir.cs.uiuc.edu/astgen/

Recap


ASTGen
generates
Grammar
(roughly) corresponds to
AST Nodes
generates
language for
composed of
Program Text
parsed by
Parser
produces
Parse Tree
acts on
Refactoring Engine