Extensible Compilers

Diplomarbeit

Matthias Zenger

Supervisor: Martin Odersky

Motivation

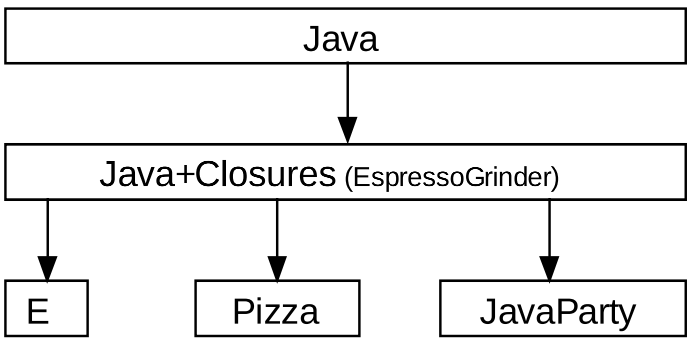
Assumption in traditional compiler construction:
Compilers are build for a concrete, fixed source language

Consequence
The notion of reusability and extensibility has no supporting mechanism in compiler construction

But
New programming languages evolve quickly, depending on the experience and requirements of their users

$\Longrightarrow$ Family of related programming languages

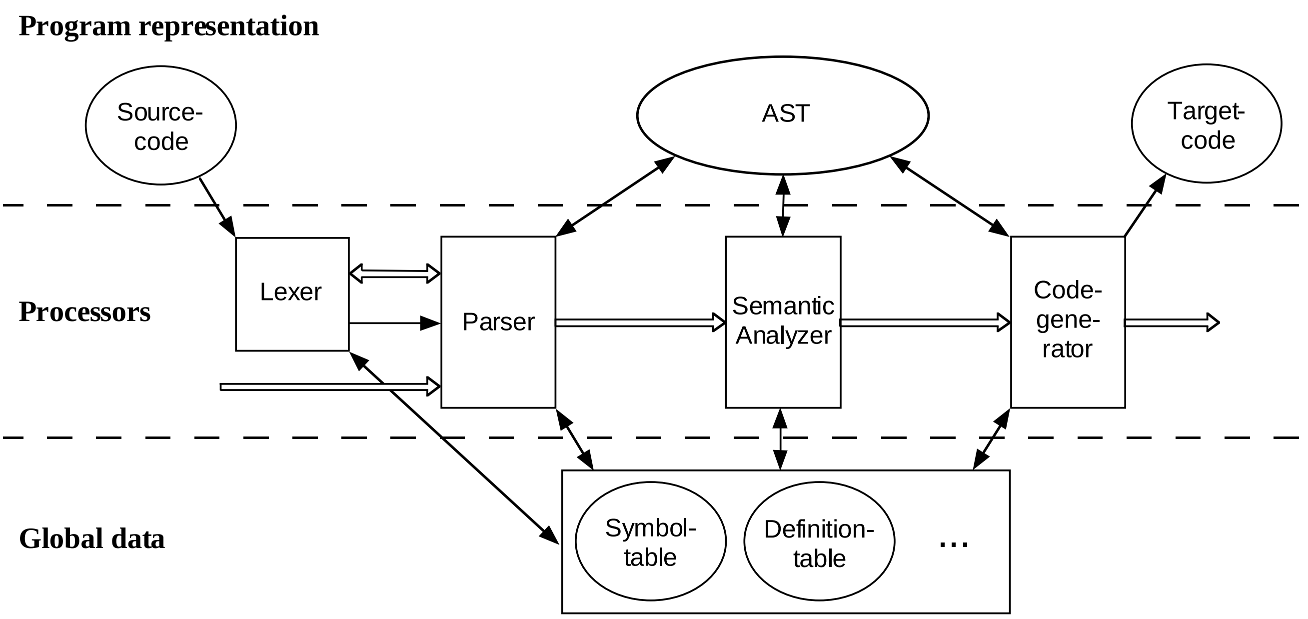
Example



Traditional Compiler Architecture (1)

Basic concepts of a modern compiler:

- organization as a repository
- functional decomposition into a number of subsequent passes



Traditional Compiler Architecture (2)

Problem

neither data nor functions are usually organized in a way that supports extensibility and reuse

Consequence

- *building* a compiler for an extended language often means starting from scratch
- *maintaining* compilers for related languages is usually difficult

Requirements for Extensibility

A compiler should be built in a way that

- reduces complexity by a functional and structural decomposition of the system

and that allows

- the extension of data types (e.g. program representation)
- modifications to existing components/processors
- the inclusion of new components/processors

with

- no modifications to existing code, and
- reuse of existing code whenever possible.

Overview

Part 1

...discusses different approaches to implement abstract syntax trees and processors in an extensible fashion.

Extensible algebraic types are proposed as ideal data types for the internal program representation.

Part 2

...proposes a software architecture for building extensible compilers.

An extensible Java compiler is presented, that supports all defined requirements.

Problem: Extensible Interpreters

The abstract syntax is represented by a recursive data structure:

```
datatype Tree = Number Int
              | Variable String
              | Plus Tree Tree
              | Times Tree Tree
```

Processors are recursive functions that operate on these data structures:

```
typeCheck :: Tree -> Env -> Type
...
transform :: Tree -> Tree
...
generateCode :: Tree -> Code
...
```

A compiler should be implemented in a way that makes it easy to

1. extend the types and adjust the existing processors accordingly, and
2. extend the set of processors

Design Pattern: Interpreter

```
abstract class Tree {
  abstract Type typeCheck(Env env);
           Tree transform() { return this; }
  abstract Code generateCode();
}
class Number extends Tree {
   int value;
   Number(int value) { this.value = value; }
   Type typeCheck(Env env) {...}
   Tree transform() {...}
   Code generateCode() {...}
}
class Variable extends Tree {
   String name;
   Variable(String name) { this.name = name; }
   Type typeCheck(Env env) {...}
   Code generateCode() {...}
}
...
```

Consequences

- makes it easy to add new `Tree` variants, but complicate to extend the set of operations (e.g. processors)
- operations are mixed and distributed all over the code
  ⇒ code is hard to understand, to maintain and to change

Design Pattern: Visitor (1)

```
abstract class Tree {
  abstract void accept(Visitor v);

   class Number extends Tree {
      int value;
      Number(int value) { this.value = value; }
      void accept(Visitor v) { v.visit(this); }
   }
   class Variable extends Tree {
      String name;
      Variable(int name) { this.name = name; }
      void accept(Visitor v) { v.visit(this); }
   }
   ...
}
interface Visitor {
  void visit(Tree.Number tree);
  void visit(Tree.Variable tree);
   ...
}
class TypeCheck implements Visitor {
   void visit(Tree.Number tree) { res = ... }
   void visit(Tree.Variable tree) { res = ... }
   Type res;
   Env env;
   Type typeCheck(Tree tree, Env env) {
      Env oldEnv = this.env;
      this.env = env;
      tree.accept(this);
      this.env = oldEnv;
      return res;
   }
}
class Transform implements Visitor { ... }
class GenerateCode implements Visitor { ... }
```

Extending Visitors

```
abstract class ExtendedTree extends Tree {
   class Lambda extends ExtendedTree {
       Variable v;
       Tree body;
      Lambda(Variable v, Tree b) { this.v = v; this.body = b; }
     void accept(Visitor v) { ((ExtendedVisitor)v).visit(this); }
    }
   class Apply extends ExtendedTree {
       Tree fun;
       Tree arg;
      Apply(Tree fun, Tree arg) { this.fun = fun; this.arg = arg;}
     void accept(Visitor v) { ((ExtendedVisitor)v).visit(this); }
    }
}
interface ExtendedVisitor extends Visitor {
  void visit(ExtendedTree.Lambda tree);
  void visit(ExtendedTree.Apply tree);
}
class ExtendedTypeCheck extends TypeChecker
                       implements ExtendedVisitor {
   void visit(Tree.Number tree) { res = ... }
   void visit(Tree.Variable tree) { res = ... }
}
class ExtendedTransform extends Transform
                       implements ExtendedVisitor { ... }
class ExtendedGenerateCode extends GenerateCode
                          implements ExtendedVisitor { ... }
```

Design Pattern: Visitor (2)

Consequences

- makes it easy to write new operations, but complicates the extension of data types and reuse of existing operations
- operations are localized in a concrete visitor class;
  no code distribution (contiguity)
- passing arguments and returning results is difficult
- double dispatch limits performance

Algebraic Types

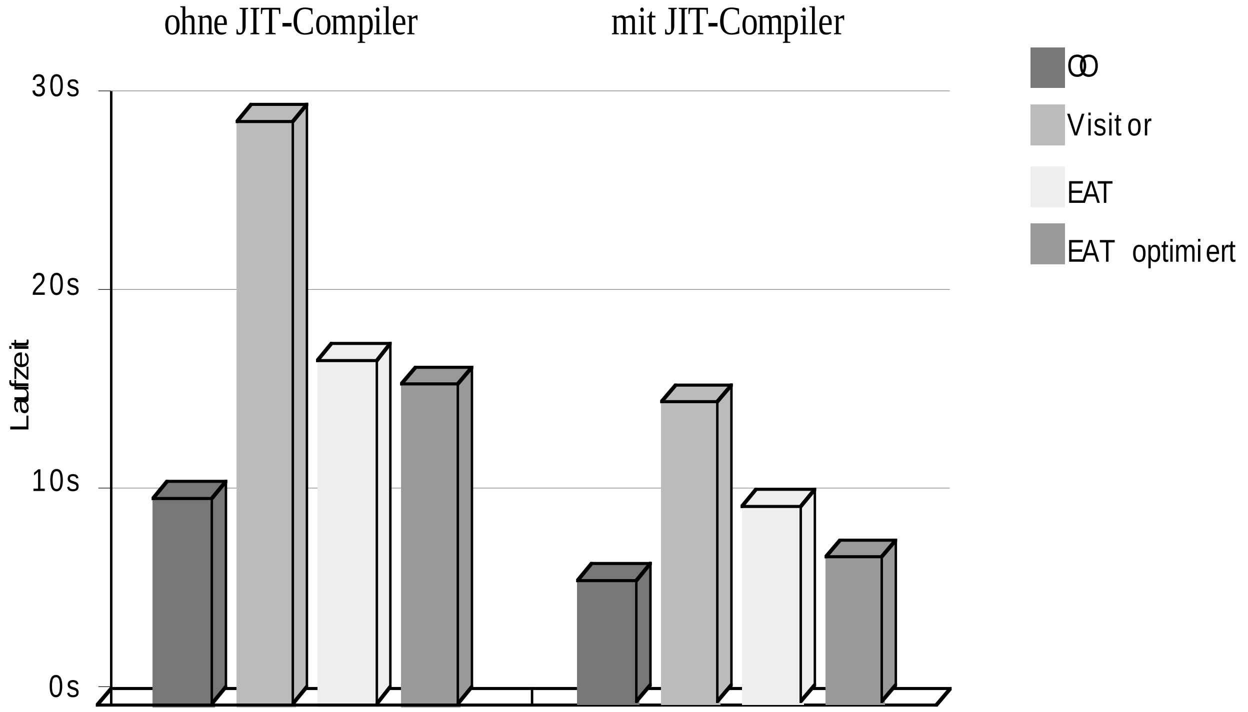
```
class Tree {
   case Number(int value);
   case Variable(String name);
   case Plus(Tree left, Tree right);
    ...
}
class Operations {
   Type typeCheck(Tree tree, Env env) {
      switch (tree) {
         case Number(int x): ...
         case Variable("null"): ...
         case Variable(String name): ...
          ...
         default: ...
      }
   }
   Tree transform(Tree tree) { ... }
   Code generateCode(Tree tree) { ... }
}


class ExtendedTree extends Tree {
   case Lambda(Variable v, Tree body);
   case Apply(Tree fun, Tree arg);
}
class ExtendedOperations extends Operations {
   Type typeCheck(Tree tree, Env env) {
      switch ((ExtendedTree)tree) {
         case Lambda(Variable v, Tree body): ...
         case Apply(Tree fun, Tree arg): ...
        default: return super.typeCheck(tree, env);
      }
   }
   Code generateCode(Tree tree) { ... }
}
```

Extensible Algebraic Types

Consequences

- synthesizes the best of the two previous approaches; it easily allows:
  - to add new `Tree` variants
  - to extend (the set of) operations
  - to reuse existing operations without any modifications
- EATs provide contiguity
- EATs and pattern matching are flexible to use; no structural patterns have to be observed while writing an operation
- EATs can be translated into efficient code

Benchmarks



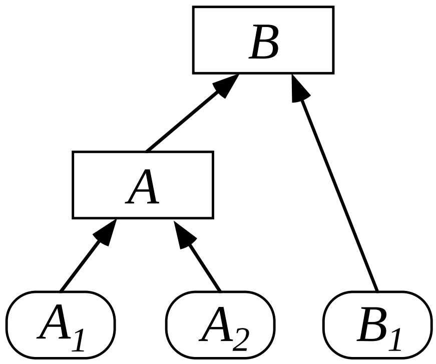
Gemessen mit JDK 1.2beta3 auf einer UltraSparc 1.

Extensible Union Types

"Traditional" way for extending algebraic types
(set theoretic):

```
class A {                  class B extends A {
   case A1(f1);               case B1(f3);
   case A2(f2);            }
}
```

$$A = A_1 + A_2 \qquad \leq \qquad B = A + B_1 = A_1 + A_2 + B_1$$



Consequences

- closes recursion in data types
- existing operations for a data type cannot be applied to the extended type
- does not fit into Java:
  - multiple immediate supertypes
  - new supertypes for existing classes
  - confuses programmer

Extensible Algebraic Types

EATs are structurally defined

Aim

make B a subtype of A

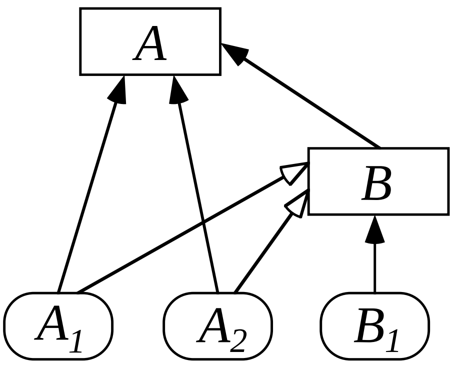
Idea

the cases of an EAT constitute a minimal set of constructors that subtypes of that type have to support

⇨ EATs are modelled by open type sums:

$$Y = inherited_Y + cases_Y + default_Y$$

$$\text{where} \quad \begin{aligned} cases_Y &= \sum_i Y_i \\ inherited_Y &= \sum_{Y \ll X} cases_X \\ default_Y &= \sum_{Z \ll Y} cases_Z \end{aligned}$$



$$A = A_1 + A_2 + default_A \quad \geq \quad B = A_1 + A_2 + B_1 + default_B$$

Architecture for an Extensible Compiler

Aim

an extendable, flexible and easy to understand compiler architecture, in which code reuse is supported as much as possible

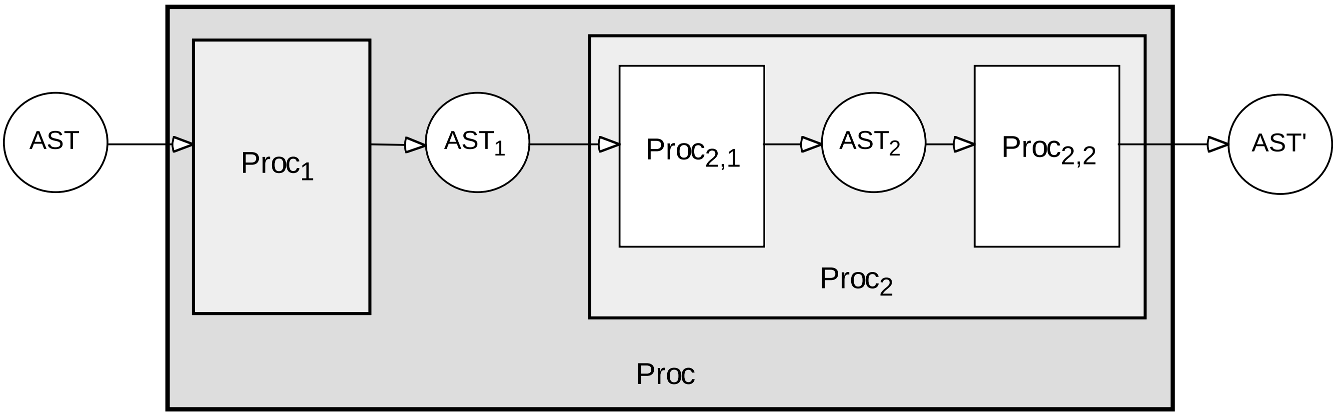
Influences on the design

- the architectural style: *Batch-Sequential Repository*
- a general architectural pattern: *Context-Component* (for the system decomposition)
- *extensible algebraic types* and pattern matching (to represent data structures and to implement processors)
- *Java* (use of late binding)
- *no tools* (simple and easy to understand concepts)

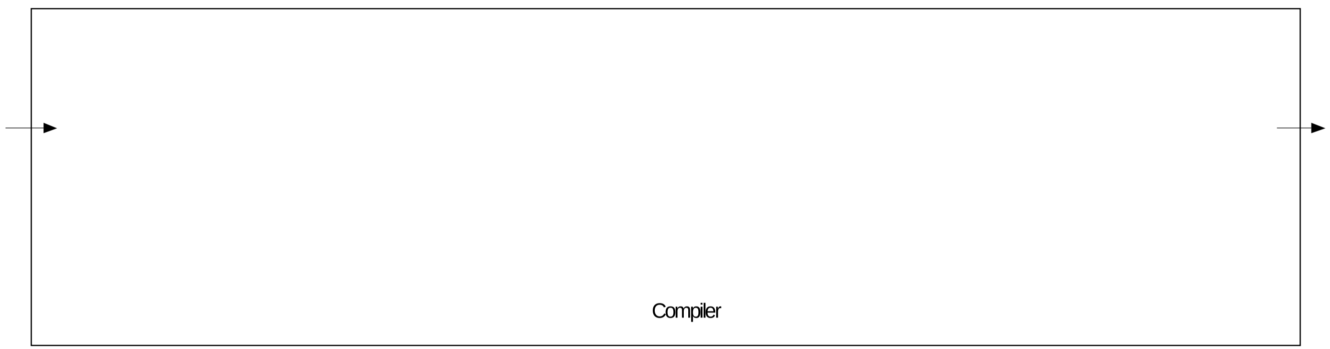
Design Idea

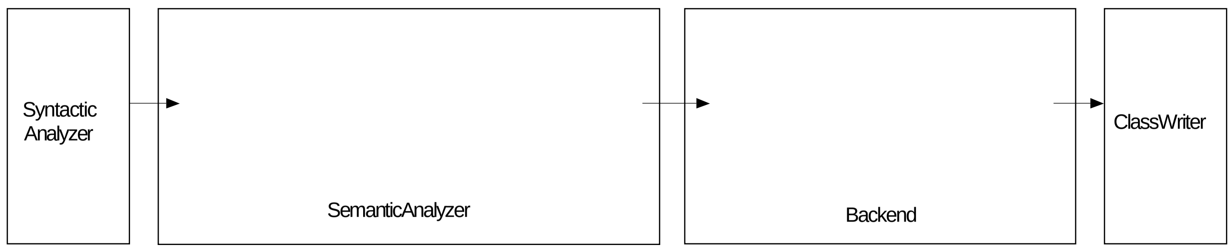
Compiler Model

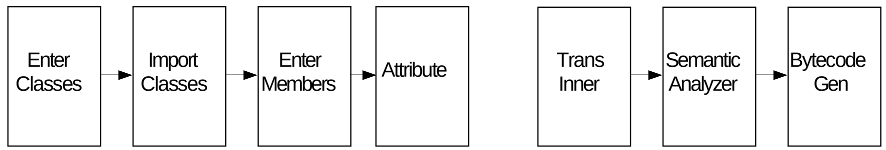
- a compiler is a processor
- a processor is either *primitive* or *composite*
- a *primitive processor* operates on the abstract syntax tree and performs side-effects
- a *composite processor* is defined by a sequence of other processors



Example



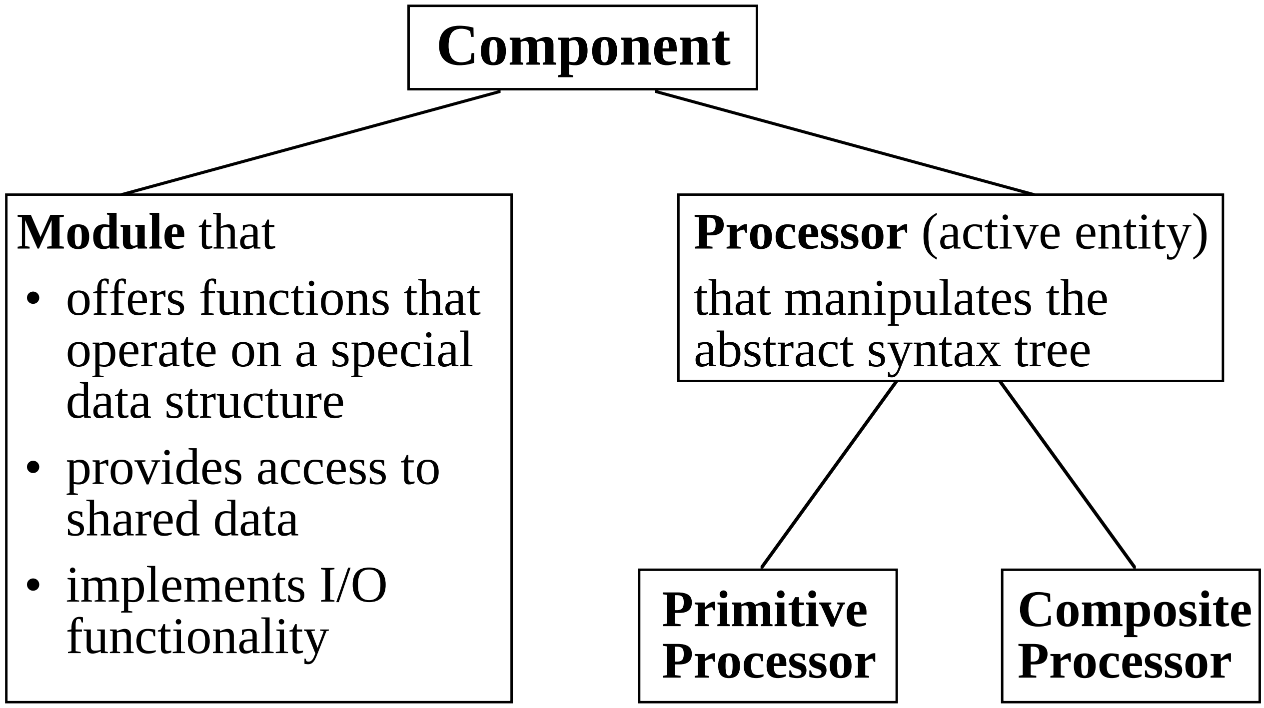

Syntactic
Analyzer
SemanticAnalyzer
Backend
ClassWriter



Composite Processors

- introduce new abstraction levels
- easily allow to reuse or to modify existing parts of the system
- make it easier to understand the system (reduce cxomplexity)

Components



Principle

The composition of a system (e.g. composite processor) is separated from the implementation of it's components.

⇨ the system decomposition is encapsulated in a configuration object, called **Context**.

Contexts (1)

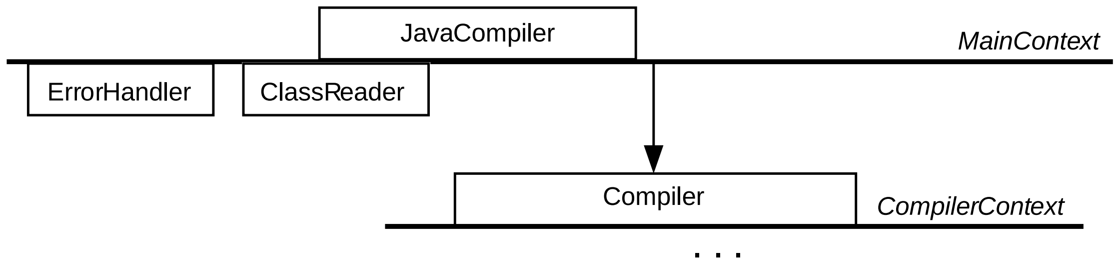
Component objects	implement a special functionality.
Context objects	represent (sub)systems. They define all subcomponents of the corresponding component object.

A **Context** is a mixture of an *AbstractFactory* and an *ObjectServer*, it defines

- the enclosing context
- factory methods for components and nested contexts
- protocols for creating the different components

Implementation of Contexts

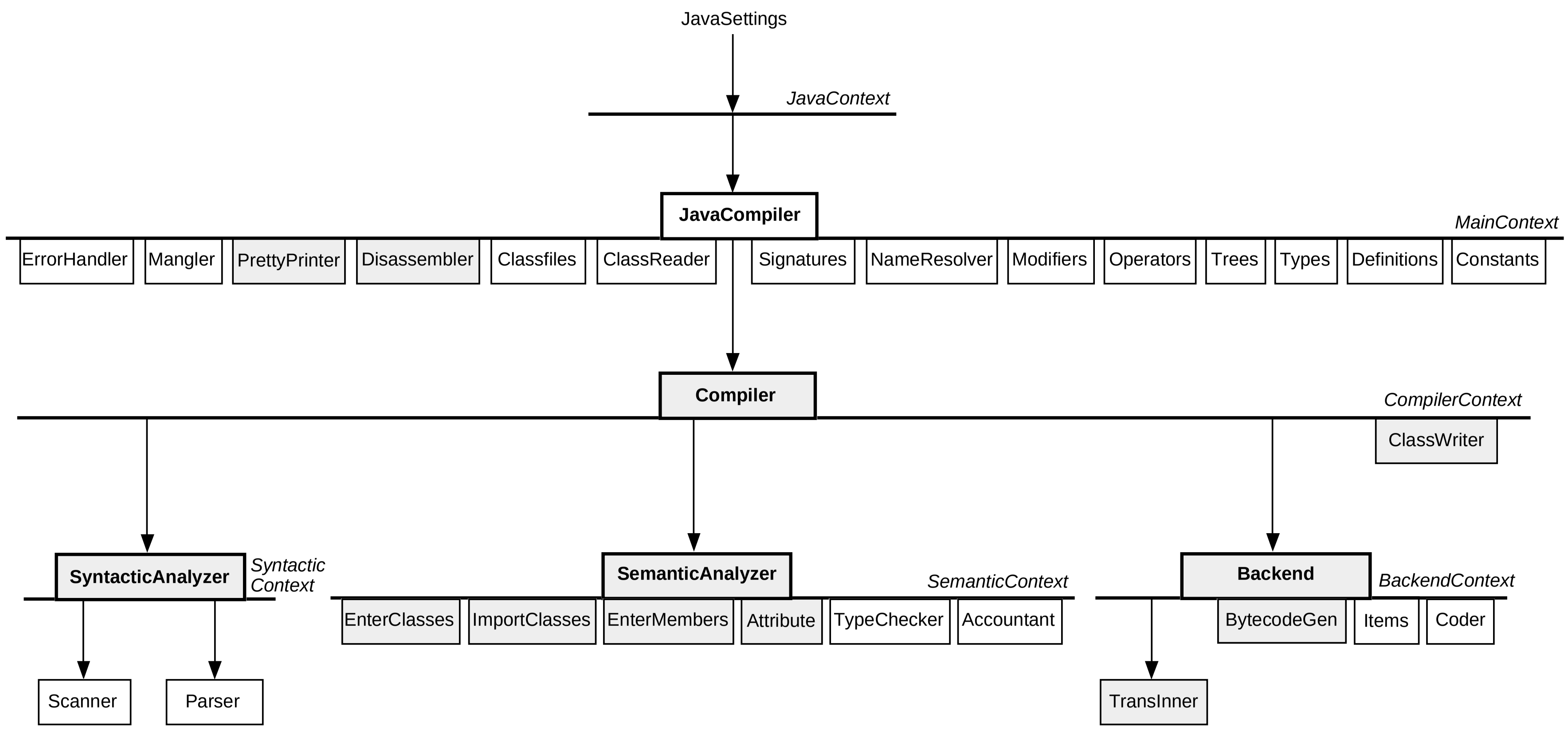
```
public class MainContext extends Context {
   public    JavaContext  enclContext;    // outer context
   protected ErrorHandler report;         // singleton components
   protected ClassReader  reader;
   ...
   public ErrorHandler ErrorHandler() {   // factory methods for
      if (report == null) {               // singleton components
         report = new ErrorHandler();
         report.init(this);
      }
      return report;
   }
   public ClassReader ClassReader() {
      ...
   }
   ...
   public Compiler Compiler() {           // factory method for
      Compiler compiler = new Compiler(); // a composite processor
      compiler.init(CompilerContext());
      return compiler;
   }                                      // factory method for the
   protected CompilerContext CompilerContext() { // nested context
      return new CompilerContext();
   }
}
```

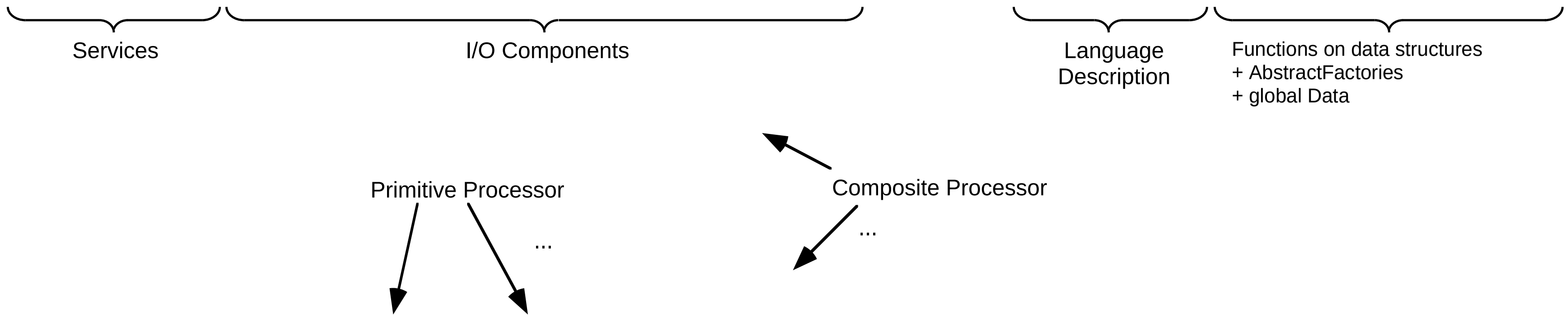


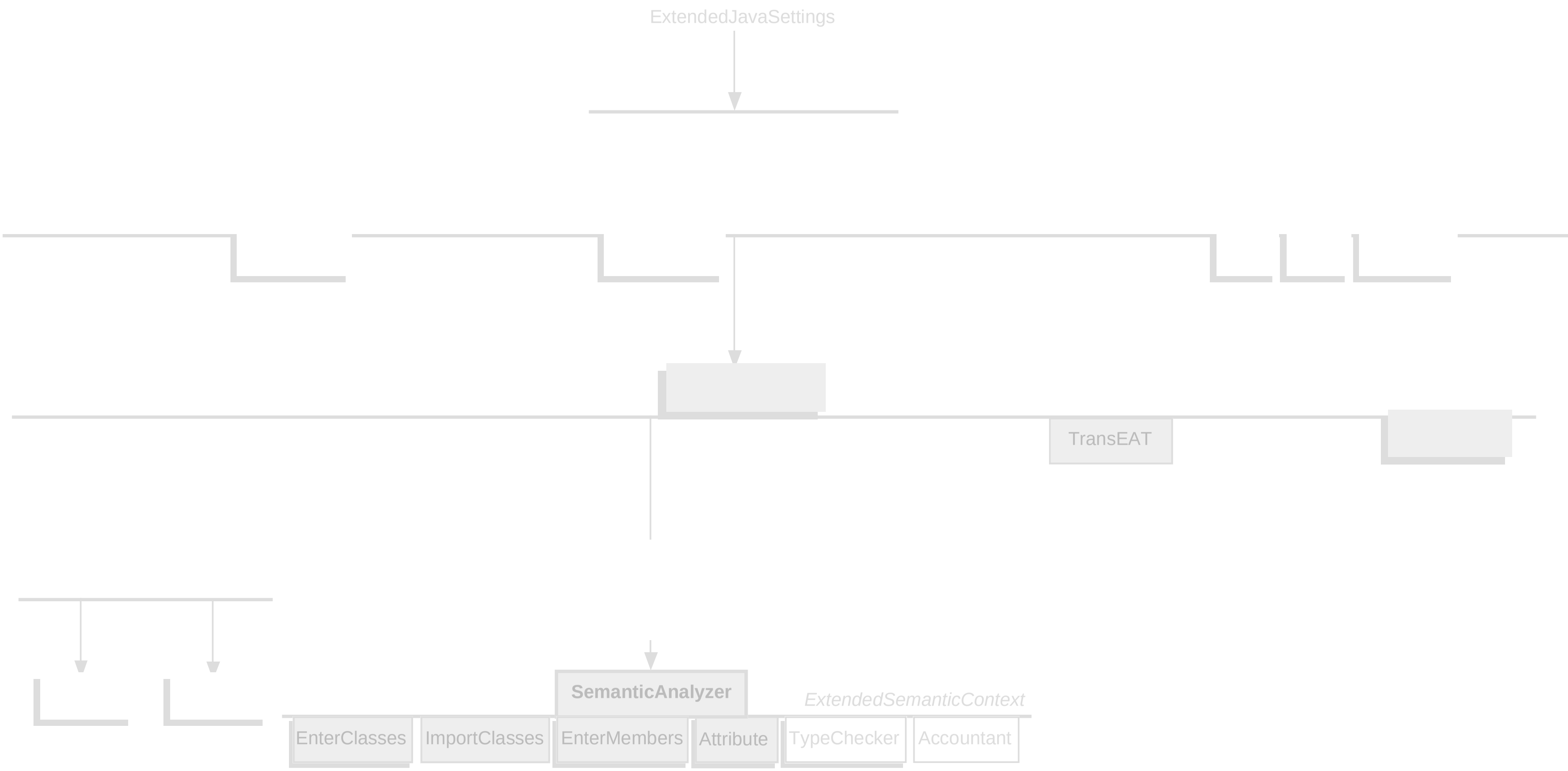
Implementation of a Component

```
public class Compiler extends CompositeProcessor {
  protected CompilerContext  context;
   ...

  public void init(CompilerContext context) {
     super.init(this);
     this.context = context;
   }
   ...

  public TreeList process(TreeList trees) throws AbortCompilation {
    return trees.apply(context.SyntacticAnalyzer())
                .apply(context.SemanticAnalyzer())
                .apply(context.Backend())
                .apply(context.ClassWriter());
   }
}
```

Structure of an Extensible Compiler




Services
I/O Components
Language Description
Functions on data structures
+ AbstractFactories
+ global Data
Primitive Processor
...
Composite Processor
...



ExtendedJavaSettings
TransEAT
SemanticAnalyzer
ExtendedSemanticContext
EnterClasses
ImportClasses
EnterMembers
Attribute
TypeChecker
Accountant

Contexts (2)

- Contexts allow a hierarchical organization of complex systems
- Contexts offer a uniform way for configuring systems:
  - centralized, and
  - decoupled from the implementation of all components
- Contexts document the structural decomposition of the compiler
- A system that is structured by contexts is flexible and easy to extend / reuse

Building an Extended Compiler

1. Extension of algebraic types and corresponding abstract factories
2. Extension of existing components by subclassing
   (a) changing behaviour by overriding existing methods
   (b) new methods to offer new functionality
3. Implementation of new components
4. Creation of an extended context hierarchy to configure the new compiler

⇨ *new compilers evolve incrementally out of existing compilers*

new compilers reuse "old" components/contexts

no code modifications are necessary

⇨ *the "old" compiler still exists*

bug fixes for the "old" compiler apply automatically for the new compiler

Representation of Data (1)

Examples

Tree, Definition, Type, Constant

Principle

every data type is implemented by

- a type definition in form of an extensible algebraic data type,
- abstract factories to create instances of the type, and
- a component that offers functions on the type

⇨ data and functions are separately defined

the type and functions on that type can be extended separately and independently

Representation of Data (2)

An OO approach is not flexible enough:

```
class ClassType extends Type {
   boolean subtype(Type t) {
      <Java subtype relation>
   }
   ...
}

class ExtendedClassType extends ClassType {
   boolean subtype(Type t) {
      <Ext.Java subtype relation>
   }
   ...
}
```

For translating EATs, in some contexts you need the new subtype method, in others you have to access the old subtype method.