

Adventures in Extensibility: Of Languages and Compilers

Robert Grimm, New York University

Joint Work with Laune Harris, Martin Hirzel, and Anh Le



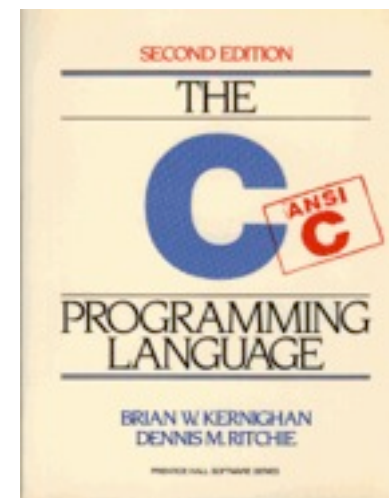
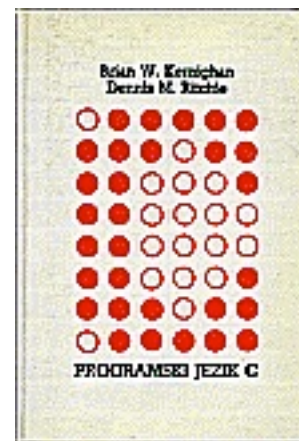
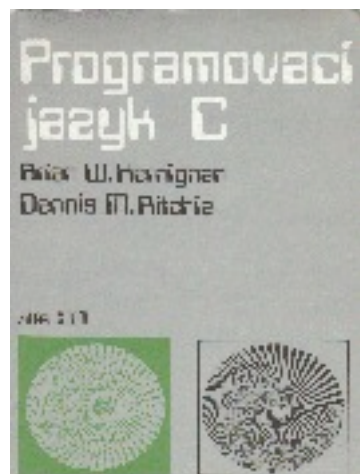
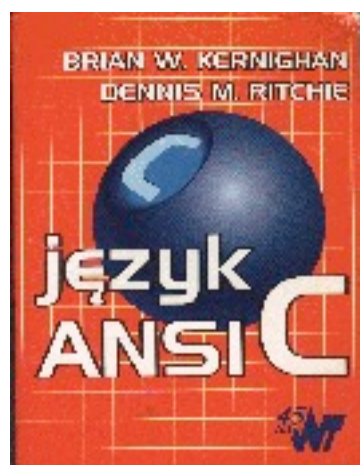


What Systems Language?





For 30 Years, There's Been Mostly One



For 30 Years, There's Been Mostly One

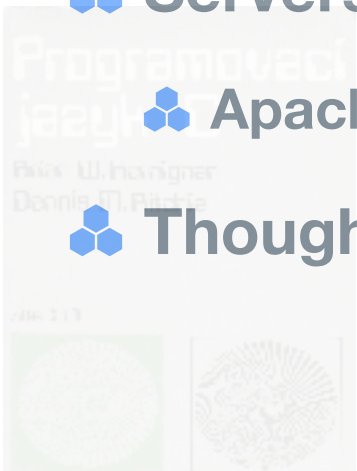
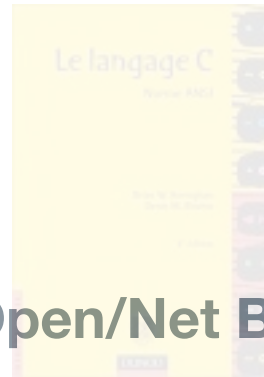
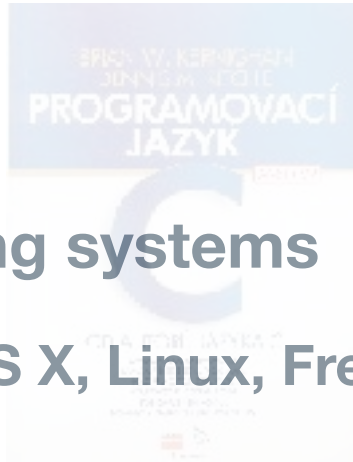
Operating systems

Mac OS X, Linux, Free/Open/Net BSD

Servers

Apache (HTTP), Bind (DNS), Sendmail (SMTP)

Though C++ for Windows, Mozilla, Internet Explorer, ...



And We Are Still Paying for This Choice...

You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。

So, You Want to Explore Your Own Systems Language?

- Follow in the footsteps of Pilot, SPIN, Singularity, ...
- What language constructs?
 - More safe, more secure, more expressive, more extensible
 - But maintain low-level control and performance of C
 - Keep systems crowd happy
- How to implement constructs?
 - Compilers are complex beasts
 - Parsing, semantic analysis, optimizations, code generation
 - gcc: 611,000 lines of code in platform-independent core



So, You Want to Explore Your Own Systems Language?

- Follow in the footsteps of Pilot, SPIN, Singularity, ...
- What language constructs?
 - More safe, more secure, more expressive, more extensible
 - But maintain low-level control and performance of C
 - Keep systems crowd happy
- How to implement constructs?
 - Compilers are complex beasts
 - Parsing, semantic analysis, optimizations, code generation
 - gcc: 611,000 lines of code in platform-independent core



So, You Want to Explore Your Own Systems Language?

- Follow in the footsteps of Pilot, SPIN, Singularity, ...
- What language constructs?
 - More safe, more secure, more expressive, more extensible
 - But maintain low-level control and performance of C
 - Keep systems crowd happy
- How to implement constructs?
 - Extensible source-to-source transformers
 - Perform parsing, semantic analysis, translation, and pretty printing
 - Leave optimizations and code generation to traditional compilers



This Talk

Introduction

Jeannie language

-  Extends Java with C and extends C with Java

Jeannie compiler

-  Leverages the xtc (eXTensible C) toolkit

Typical: a type checker generator

-  Builds on experiences with Jeannie compiler

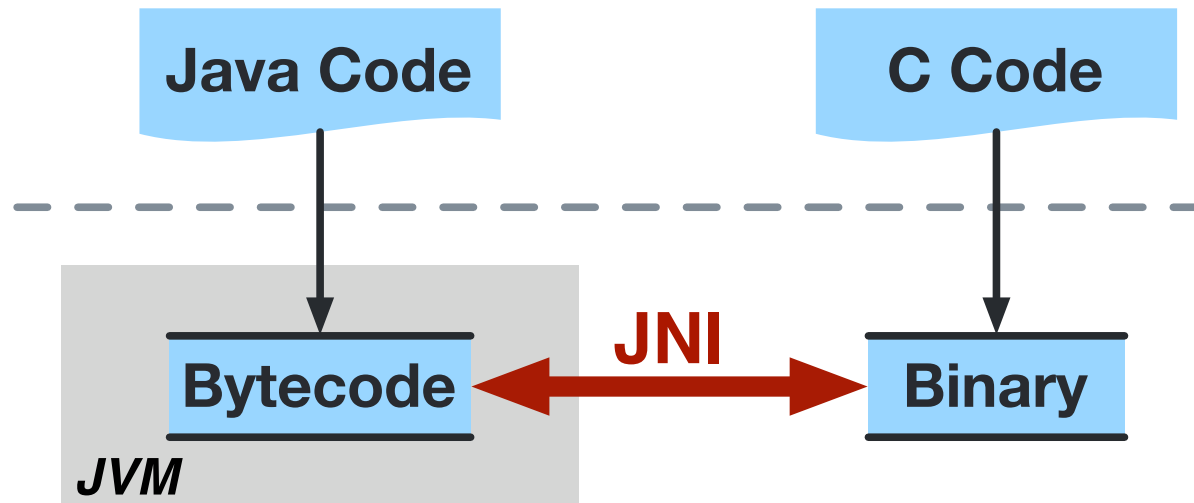
Conclusions



Point of Departure: Java Native Interface (JNI)

Foreign function interface (FFI) for Java

- Provides access to platform functionality, incl. operating system
- Enables reuse of legacy libraries
- Speeds up performance-critical code



Competing Goals for FFIs

Productivity

-  Writing and maintaining code

Safety

-  Preventing and detecting bugs

Efficiency

-  Crossing between languages

Portability

-  Moving between virtual machines, OS's, hardware



Java

```
package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}
```

C

```
#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;

    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);

    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}
```

Java

```
package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}
```

**Native method:
no body in Java**

C

```
#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;

    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);
    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}
```

Java

```
package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}
```

**Native method:
no body in Java**

C

```
#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;

    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);

    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}
```

**C function with
mangled name implements
Java method**

Java

```

package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}

```

**Native method:
no body in Java**

C

```

#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;

    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);

    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}

```

**C function with
mangled name implements
Java method**

**Read Java field
from C**

Java

```
package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}
```

Native method:
no body in Java

C

```
#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;
    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);
    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}
```

C function with
mangled name implements
Java method

Read Java field
from C

Throw Java
exception from C

Java

```

package my.net;
import java.io.IOException;

public class Socket {
    static {
        System.loadLibrary("Network");
    }
    protected int native_fd;
    protected native int available() throws IOException;
    ...
}

```

Native method:
no body in Java

C

```

#include <jni.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{
    int num, fd;

    jclass cls; jfieldID fid;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
    fd = (*env)->GetIntField(env, this, fid);

    if (ioctl(fd, FIONREAD, &num) != 0) {
        jclass exc =
            (*env)->FindClass(env, "Ljava/io/IOException;");
        (*env)->ThrowNew(env, exc, strerror(errno) );
        return 0;
    }
    return num;
}

```

- Productivity

Function with
mangled name implements
Java method

- Safety

Java method

- Efficiency

Read Java field
from C

+ Portability

Throw Java
exception from C

Say Hello to Jeannie

- ❖ A new programming language
 - ❖ Extends Java and C with each other
 - ❖ Compiles down to Java and C
 - ❖ Uses JNI to bridge between the two



```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
        IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

```
...  
}
```

Supports
all of C

```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
        IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

```
...  
}
```

Supports
all of C

Supports all of Java
plus Java types in C

```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
        IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

```
...
```

```
}
```

Supports
all of C

```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

Supports all of Java
plus Java types in C

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

Native methods have bodies

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
            IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

...

```
}
```

Supports
all of C

```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

Supports all of Java
plus Java types in C

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

Native methods have bodies

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
        IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

Just access a field

```
...  
}
```

Supports
all of C

```
`.C {  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <string.h>  
}
```

Supports all of Java
plus Java types in C

```
package my.net;  
import java.io.IOException;
```

```
public class Socket {  
    protected int native_fd;  
    protected native int available() throws IOException
```

Native methods have bodies

```
{  
    int num;  
    int fd = `this.native_fd` ;  
    if (ioctl(fd, FIONREAD, &num) != 0)  
        `throw new  
        IOException( `_newJavaString(strerror(errno))` );  
    return num;  
}
```

Just access a field

Just throw an exception

```
...  
}
```

**Supports
all of C**

```
`.C {
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
}
```

**Supports all of Java
plus Java types in C**

```
package my.net;
import java.io.IOException;
```

Native methods have bodies

```
public class Socket {
    protected int native_fd;
    protected native int available() throws IOException
```

Just access a field

```
{
    int num;
    int fd = `this.native_fd ;
    if (ioctl(fd, FIONREAD, &num) != 0)
```

Just throw an exception

```
    `throw new
        IOException( `_newJavaString(strerror(errno)) );
```

Nest to any depth

```
    return num;
```

```
}
```

```
...
```

```
}
```

Supports
all of C

```
`.C {
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
}
```

Supports all of Java
plus Java types in C

```
package my.net;
import java.io.IOException;
```

```
public class Socket {
    protected int native_fd;
    protected native int available() throws IOException
```

+ Productivity

Native methods have bodies

+ Safety

Just access a field

+ Efficiency

+ Portability

Just throw an exception

```
{
    int num;
    int fd = `this.native_fd ;
    if (ioctl(fd, FIONREAD, &num) == -1)
        `throw new
            IOException( `_newJavaString(strerror(errno)) );
    return num;
}
```

Nest to any depth

...

}



Typing

- Java primitives map to equivalent C primitives

```
if ( `( `(boolean) feof(stdin) ) ) ... ;
```

- Java references are opaque in C

- Can be copied but not dereferenced

- Can be used in nested Java expressions

- Are automatically widened

```
`List lst = `new ArrayList(10) ;
```

- C pointers, structs, and unions are illegal in Java

Typing

- Java primitives map to equivalent C primitives

```
if (`((`boolean)feof(stdin))) ... ;
```

- Java references are opaque in C

- JNI: all Java references map onto same C type**

- Can be copied but not dereferenced

```
typedef struct _jobject * jobject;
```

- Can be used in nested Java expressions

- Jeannie: preserves all Java types across C code**

- Are automatically widened

```
`List lst = `new ArrayList(10);
```

- C pointers, structs, and unions are illegal in Java

Array Access

❖ Nested expressions are simple

```
for (`int i=0, n=`ja.length; i<n; i++)  
    s += `ja[`i];
```

❖ But bulk access is faster

```
with (`int* ca=`ja) {  
    for (`int i=0, n=`ja.length; i<n; i++)  
        s += ca[i];  
}
```

❖ In our experiments, 70 times faster



This Talk

Introduction

Jeannie language

-  Extends Java with C and extends C with Java

Jeannie compiler

-  Leverages the xtc (eXTensible C) toolkit

Typical: a type checker generator

-  Builds on experiences with Jeannie compiler

Conclusions



Translation Scheme

```
public static native void f(int x)
`{ 1
  `int y = 0;
  `{ 2
    int z;
    z = 1 + `(y = 1 + `(x = 1) 4) 3 ;
    System.out.println(x);
    System.out.println(z);
  }
  printf("%d\n", y);
}
```

Translation Scheme

```
public static native void f(int x)
`{ 1
  `int y = 0;
  `{ 2
    int z;
    z = 1 + `(y = 1 + `(x = 1) 4) 3 ;
    System.out.println(x);
    System.out.println(z);
  }
  printf("%d\n", y);
}
```

A yellow speech bubble containing the string "1" points to the innermost expression `(x = 1)` in the assignment statement `z = 1 + (y = 1 + (x = 1)) ;`.

Translation Scheme

```
public static native void f(int x)
```

```
`{ 1
  `int y = 0;
  `{ 2
    int z;
    z = 1 + `(y = 1 + `(x = 1) 4) 3 ;
    System.out.println(x);
    System.out.println(z);
  }
  printf("%d\n", y);
}
```

“1”

“2”

Translation Scheme

```

public static native void f(int x)
{
    int y = 0;
    {
        int z;
        z = 1 + (y = 1 + (x = 1));
        System.out.println(x);
        System.out.println(z);
    }
    printf("%d\n", y);
}

```

Annotations in the code above:

- A yellow speech bubble with "1" points to the innermost expression `(x = 1)`.
- A yellow speech bubble with "2" points to the `printf` statement.

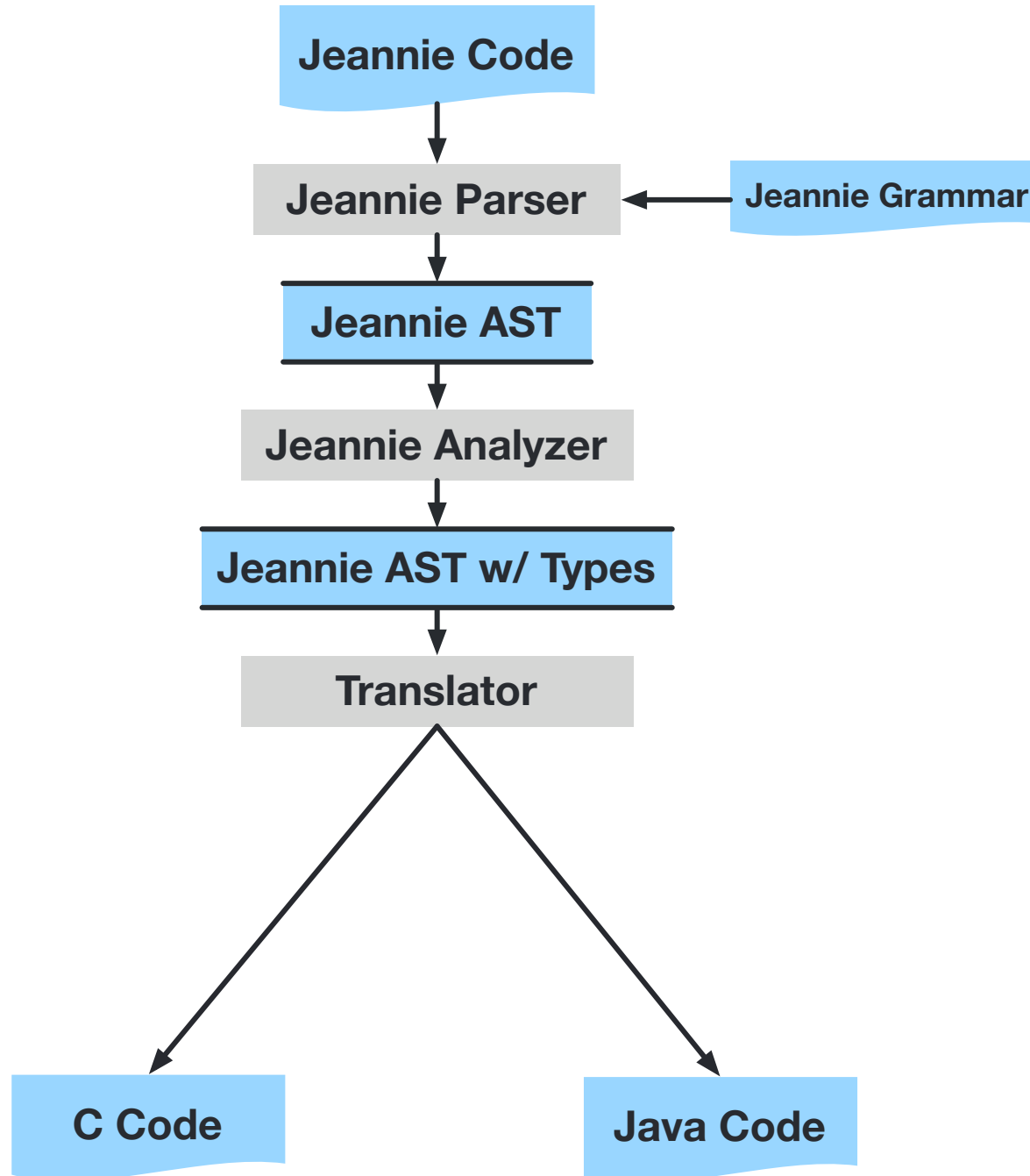
```

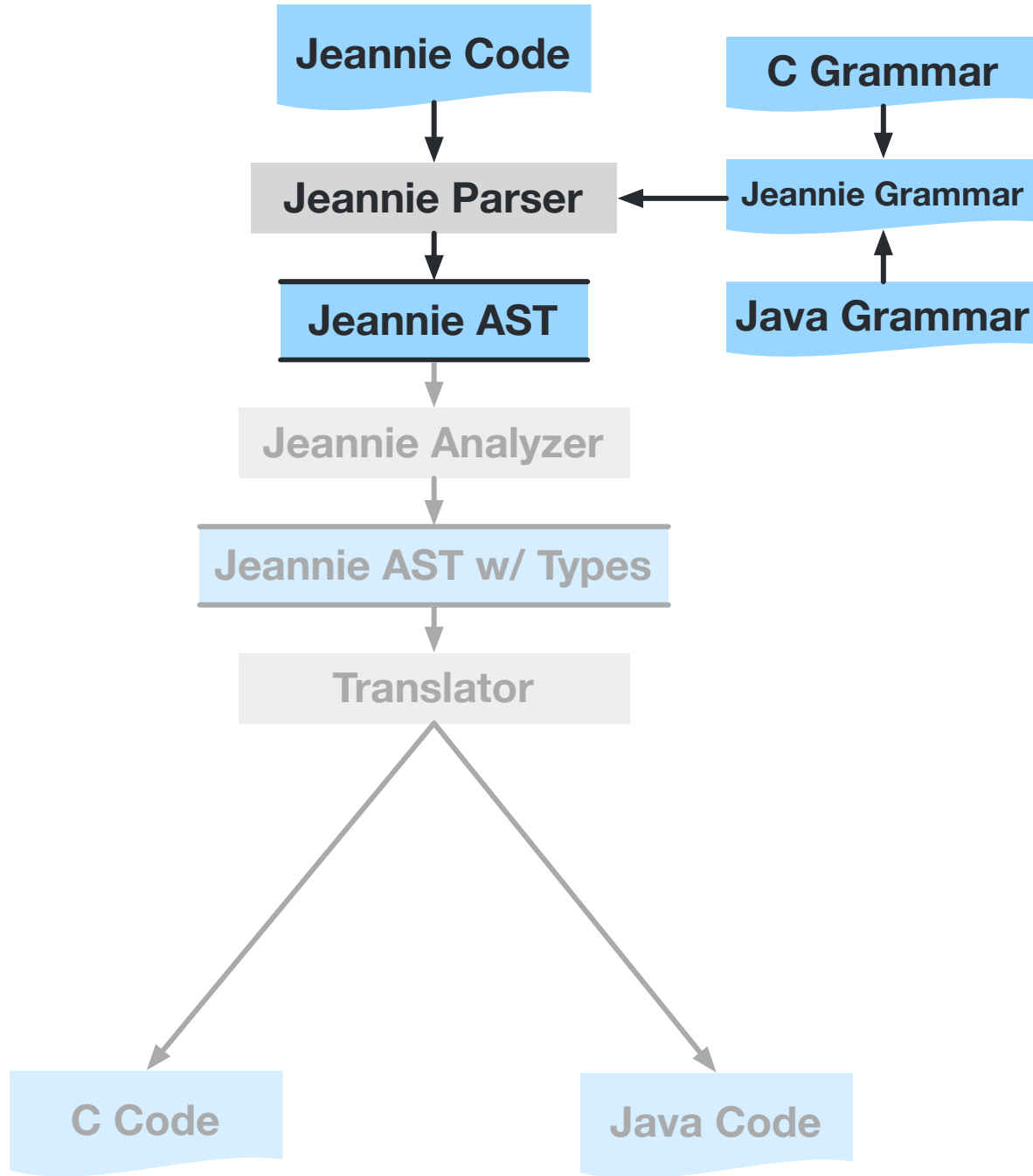
class JNIEnv {
    int x;
    int z;
    native void m1();
    void m2(CEnv)...
    native int m3(CEnv);
    int m4(CEnv)...
}

struct CEnv {
    jint y;
};

void C_m1(JNIEnv)...
jint C_m3(JNIEnv, CEnv)...

```





Rats!
parser
generator



```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

```
modify CBase;  
import Java;  
import Util;
```

```
Node UnaryExpression +=  
    <JavaInC>      JavaInCExpression  
/ <LogicalNot> ... ;
```

```
private generic JavaInCExpression =  
    JavaInC Java.UnaryExpression ;
```



Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

```
modify CBase;  
import Java;  
import Util;
```

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
  / <LogicalNot> ... ;
```

```
private generic JavaInCExpression =  
  JavaInC Java.UnaryExpression ;
```



Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

Apply changes
to module CBase

```
modify CBase;  
import Java;  
import Util;
```

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
/ <LogicalNot> ... ;
```

```
private generic JavaInCExpression =  
  JavaInC Java.UnaryExpression ;
```





Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

Apply changes
to module CBase

```
modify CBase;  
import Java;  
import Util;
```

Add new alternative

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
  / <LogicalNot> ... ;
```

```
private generic JavaInCExpression =  
  JavaInC Java.UnaryExpression ;
```





Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

Apply changes
to module CBase

```
modify CBase;  
import Java;  
import Util;
```

Add new alternative

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
  / <LogicalNot> ... ;
```

Create tree node

```
private generic JavaInCExpression =  
  JavaInC Java.UnaryExpression ;
```





Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

Apply changes
to module CBase

```
modify CBase;  
import Java;  
import Util;
```

Add new alternative

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
  / <LogicalNot> ... ;
```

Create tree node

```
private generic “JavaInCExpression” =  
  JavaInC Java.UnaryExpression ;
```





Delay composition
until parser generation

```
module xtc.lang.jeannie.C(CBase, Java, Util);
```

Apply changes
to module CBase

```
modify CBase;  
import Java;  
import Util;
```

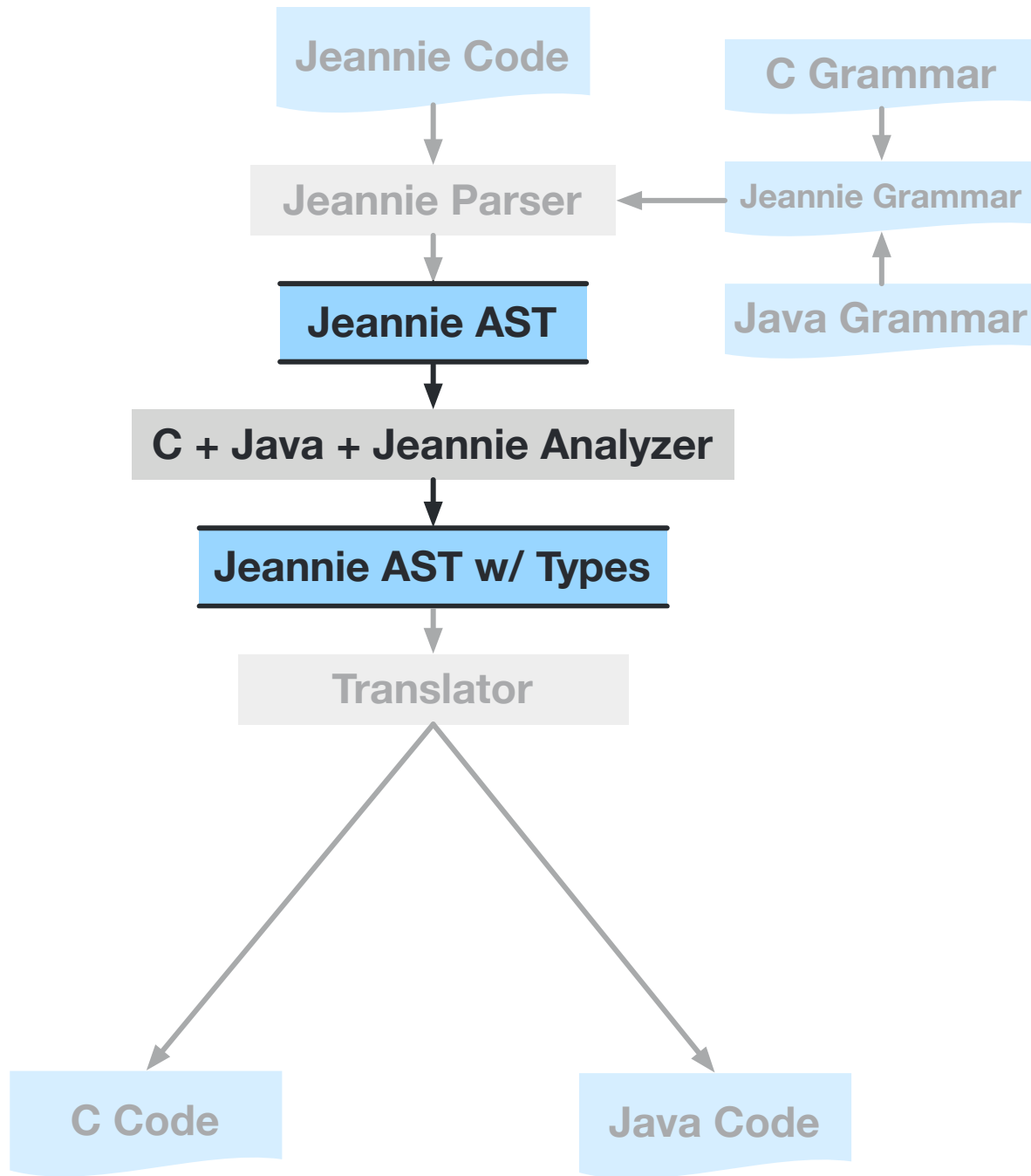
Add new alternative

```
Node UnaryExpression +=  
  <JavaInC>      JavaInCExpression  
  / <LogicalNot> ... ;
```

Create tree node

```
private generic “JavaInCExpression” =  
JavaInC Java.UnaryExpression ;
```





Rats!
parser
generator

xtc visitors,
common
type rep.



Dynamically Dispatched Visitors

Incant `dispatch(n)`

 Selects method through reflection, caches result

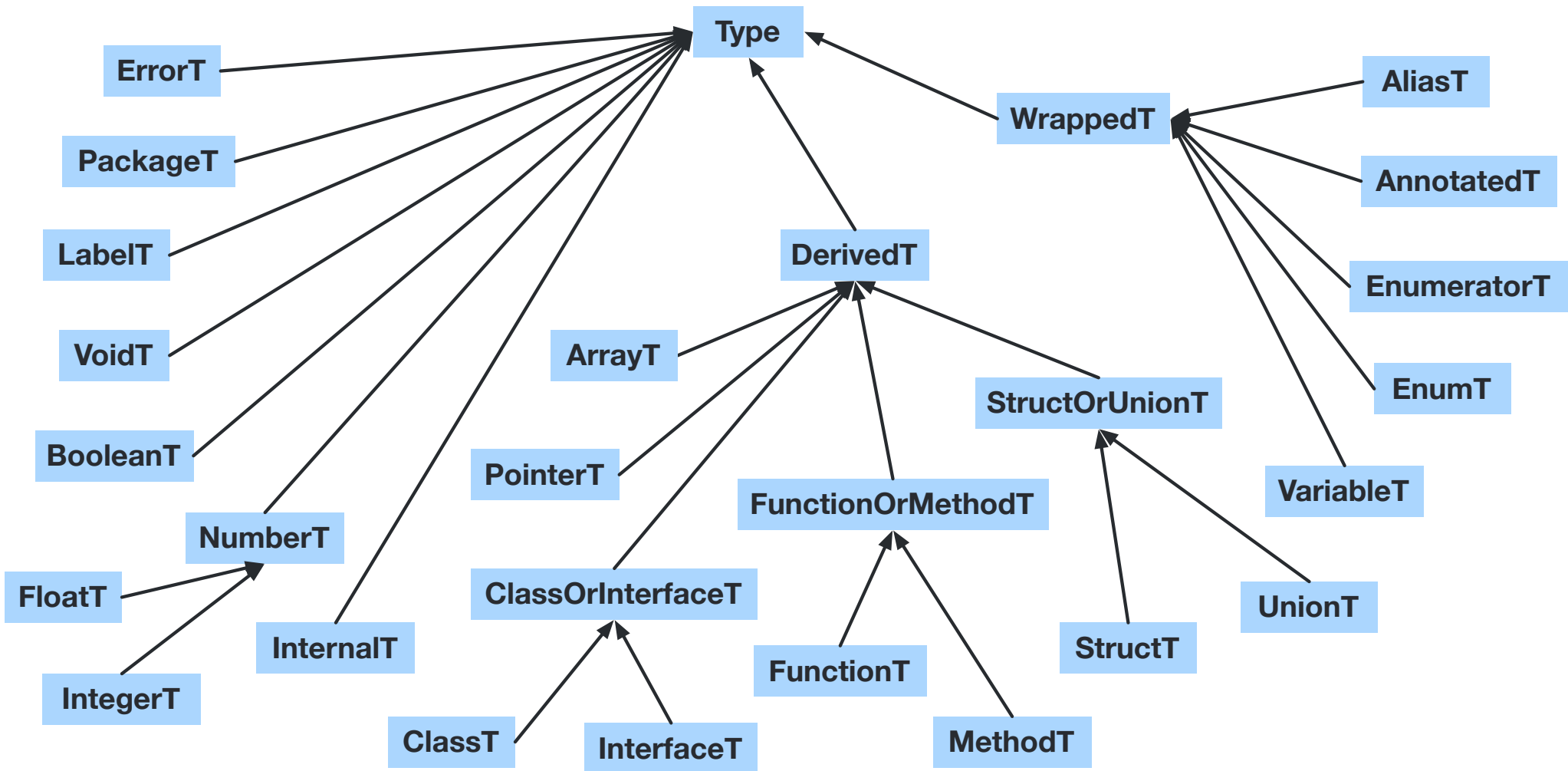
 First name-based: e.g., `visitInt(GNode)` for generic node named “Int”

 Then type-based: `visit(T)` with closest matching T for all nodes

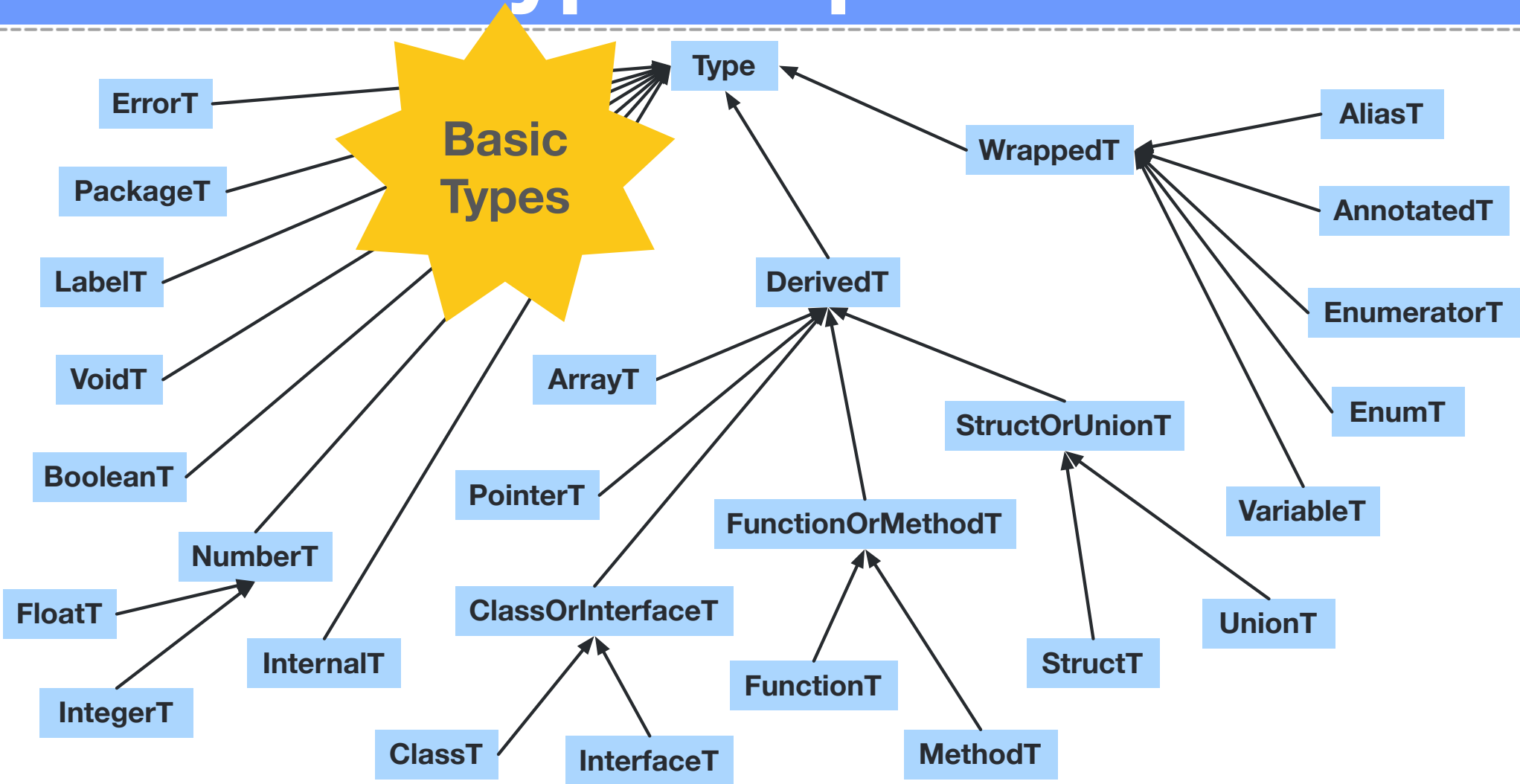
 Enables generic tree traversal

```
public void visit(Node node) {
    for (Object o : node)
        if (o instanceof Node) dispatch((Node)o);
}
```

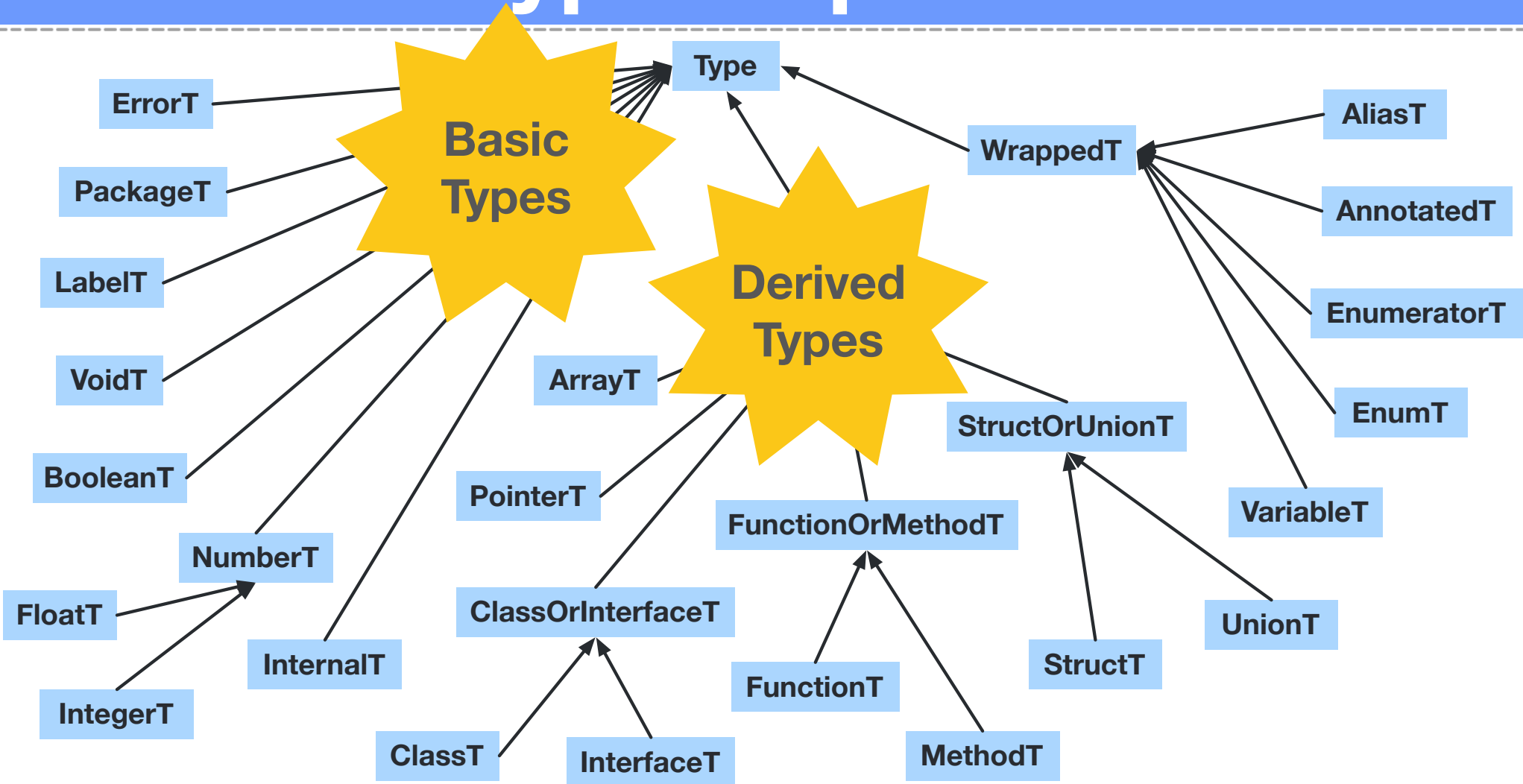
Common Type Representation



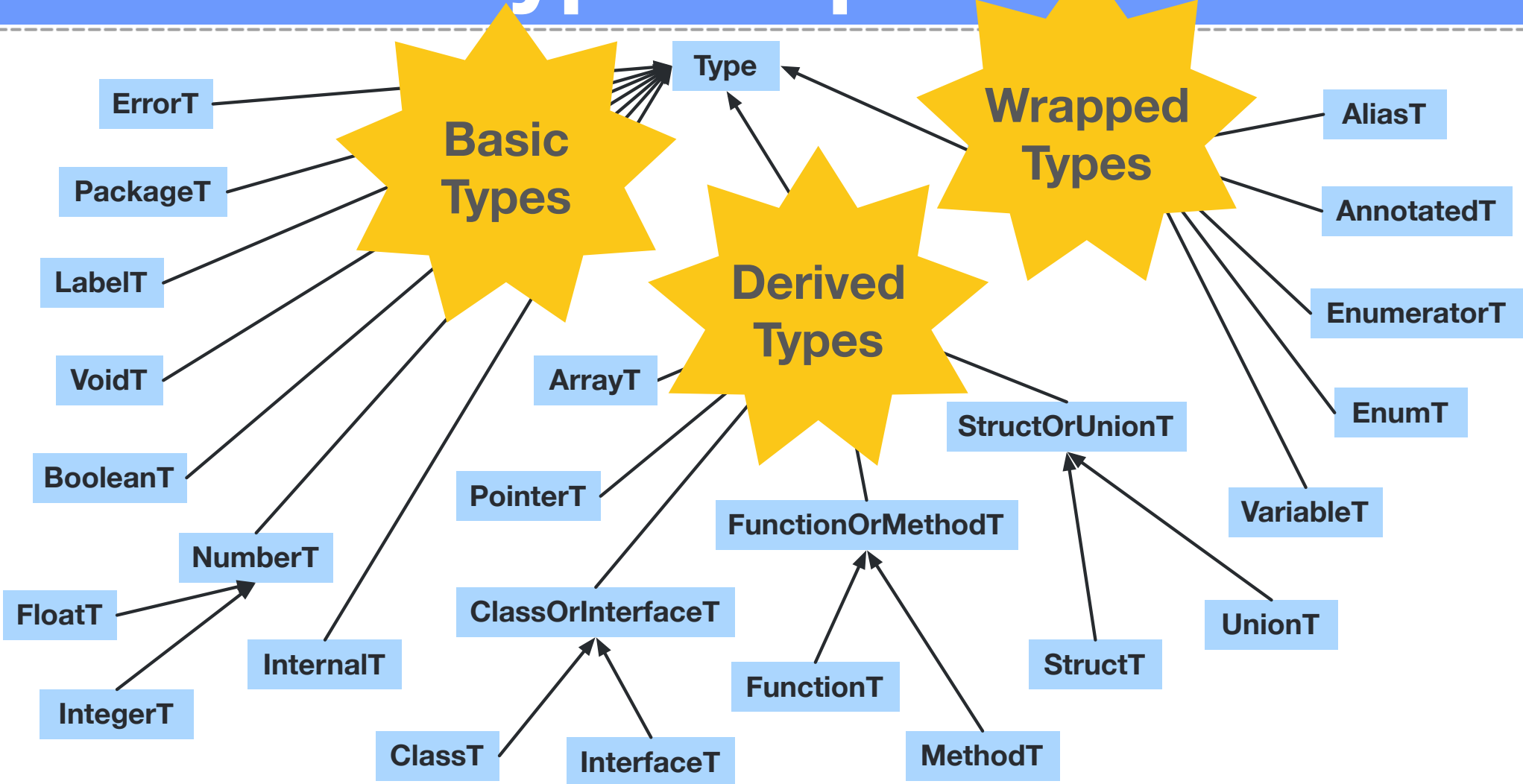
Common Type Representation

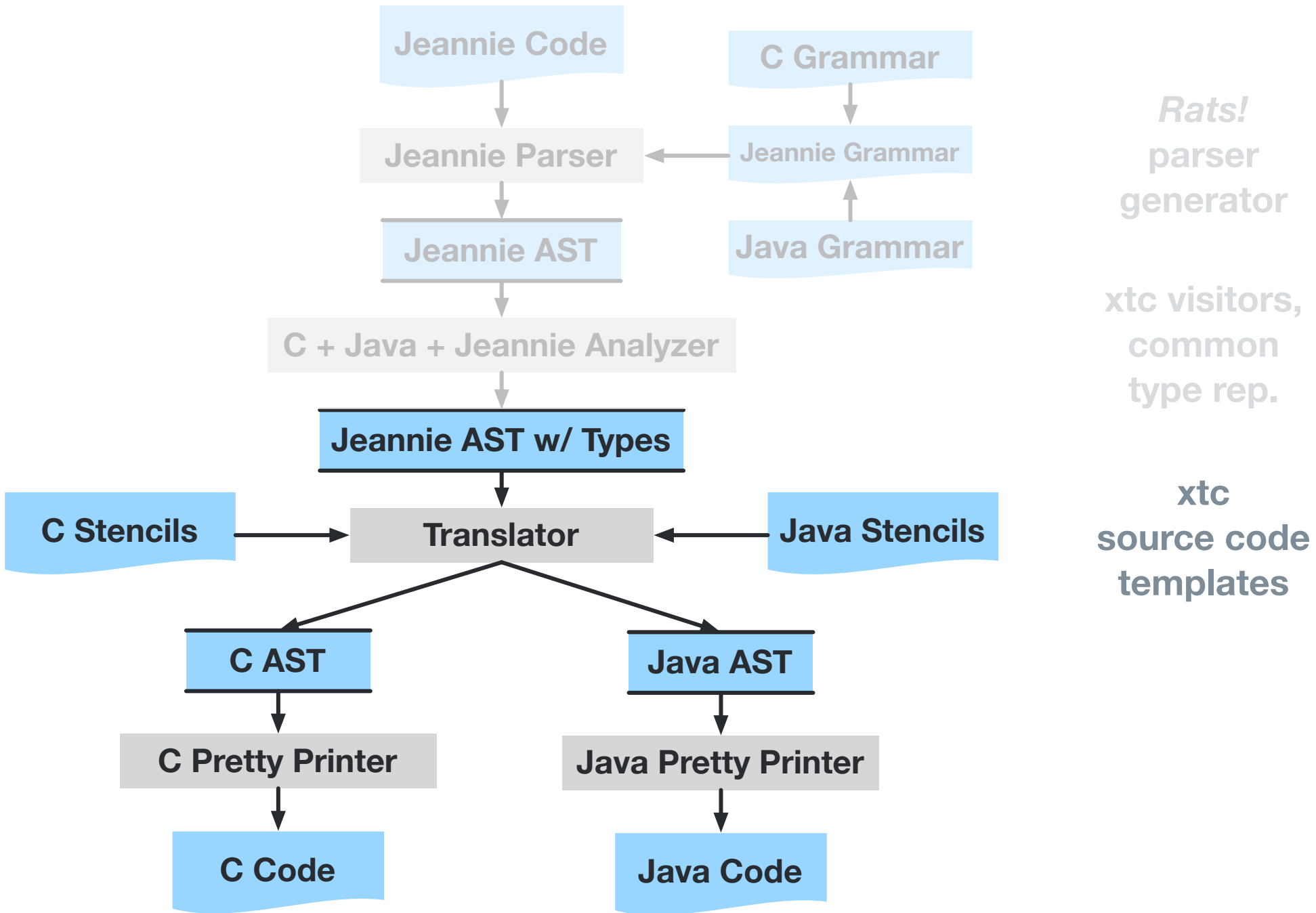


Common Type Representation



Common Type Representation





Source Code Templates aka Stencils

- Input: Concrete syntax with pattern variables

```
getThisDotField { this.#name }
```

- Output: Method that creates the corresponding tree snippet

```
public Node getThisDotField(String name) {  
    Node v$1 = GNode.create("ThisExpression", null);  
    Node v$2 = GNode.  
        create("SelectionExpression", v$1, name);  
    return v$2;  
}
```

- Leverage *Rats!*, dynamic trees, and dynamic visitors

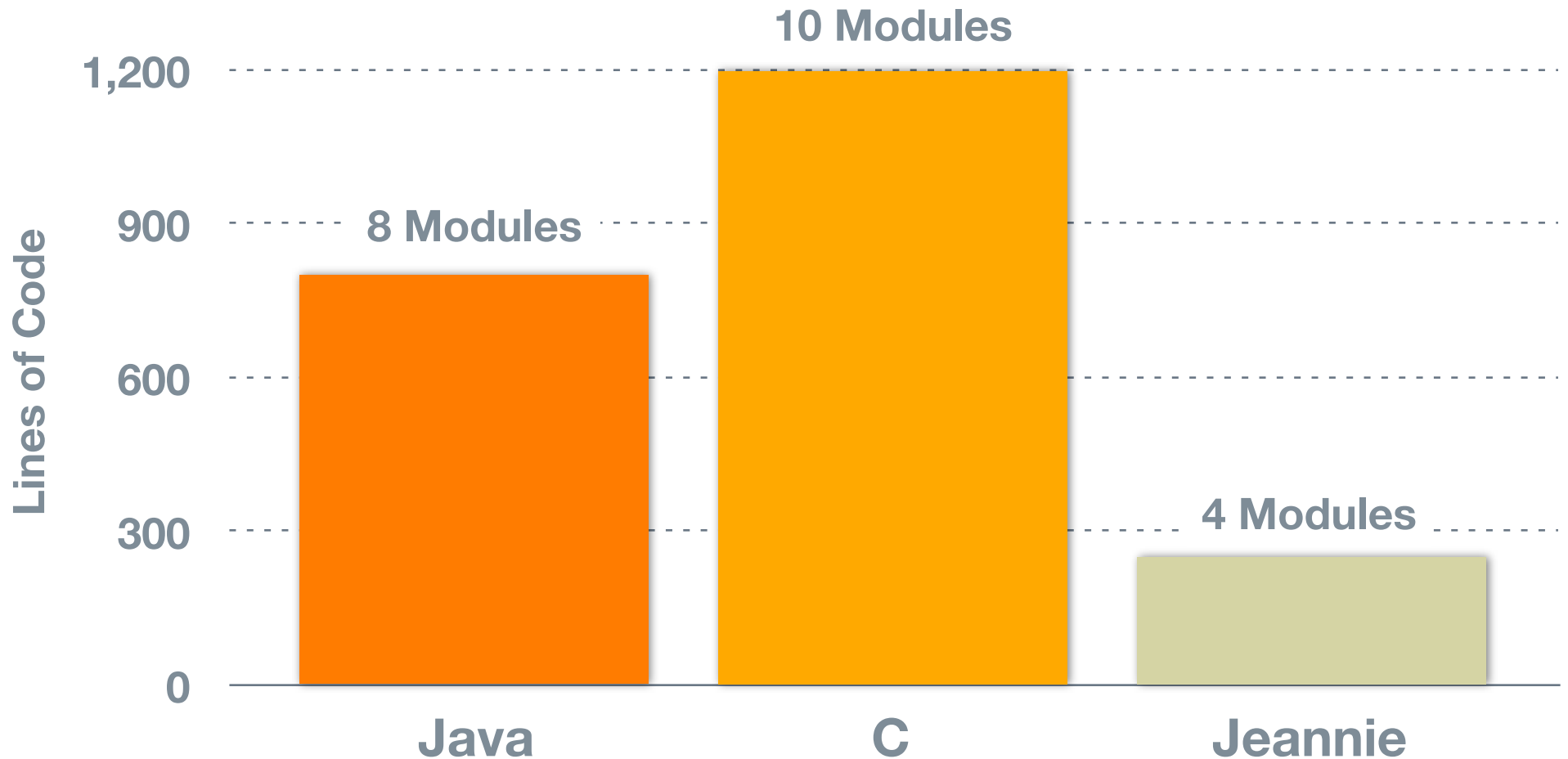
- 80 lines each for C & Java grammars; 400 lines for code generator



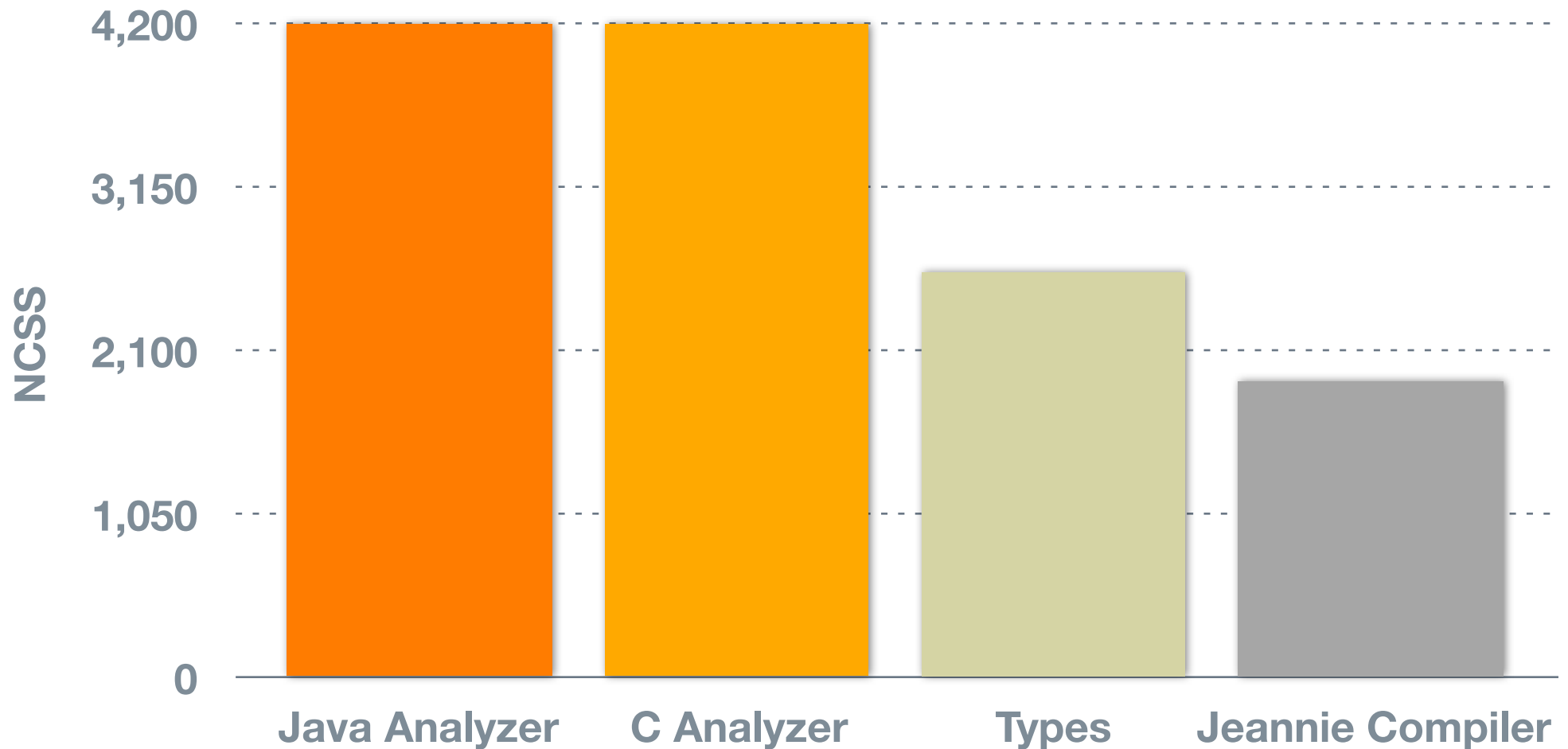
How Complex Is the Composition?



Scalable Composition: Jeannie Syntax

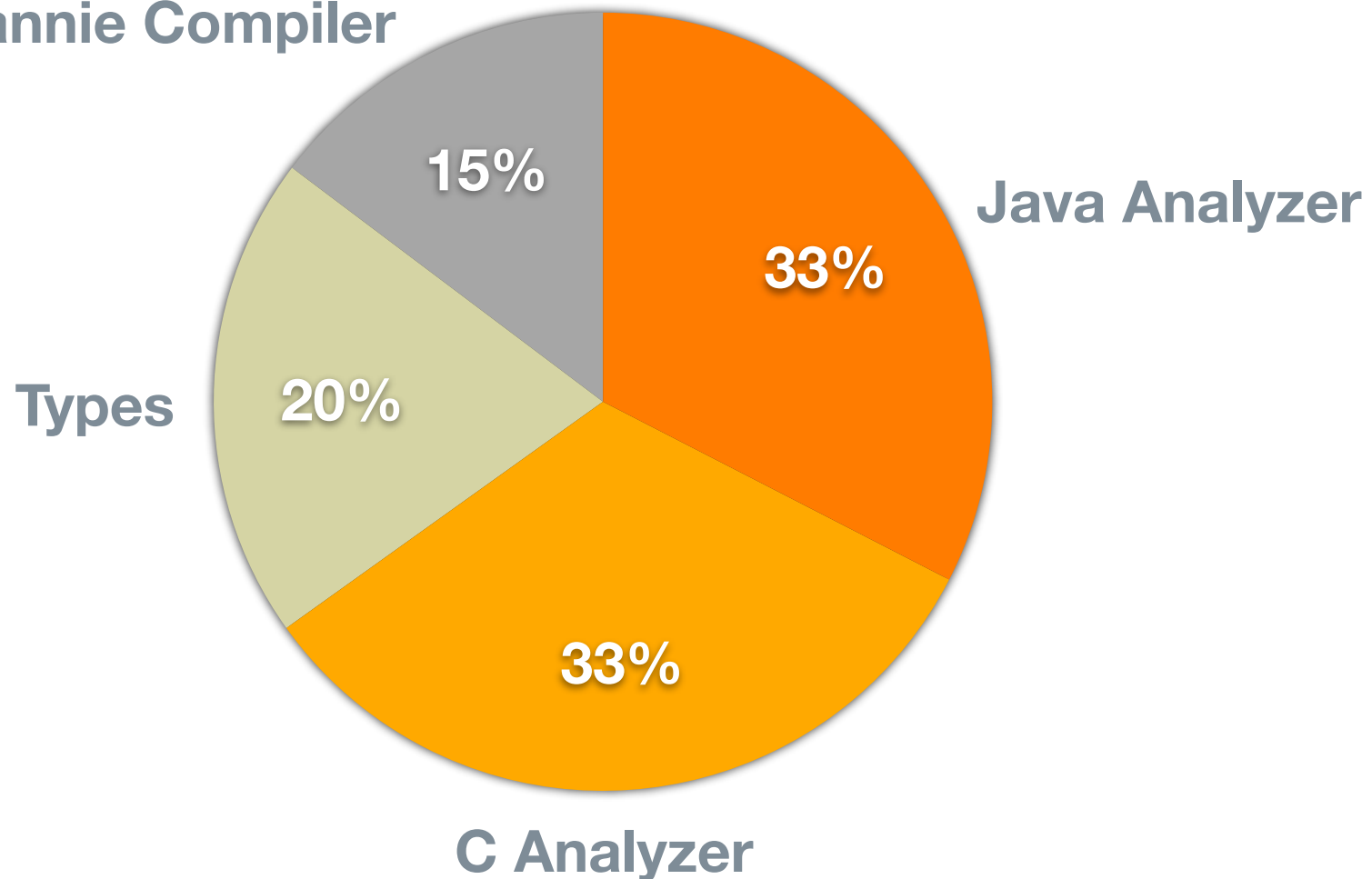


Scalable Composition: Jeannie Semantics, Translation



Scalable Composition: Jeannie Semantics, Translation

Jeannie Compiler



This Talk

Introduction

Jeannie language

-  Extends Java with C and extends C with Java

Jeannie compiler

-  Leverages the xtc (eXTensible C) toolkit

Typical: a type checker generator

-  Builds on experiences with Jeannie compiler

Conclusions



Can We Do Better?

- ❁ **xtc is extensible, but supports only limited static typing**
 - ❁ Dynamically typed syntax trees
 - ❁ Dynamically dispatched visitors
 - ❁ Dynamically typed type objects
 - ❁ Every type variable is declared as `xtc.type.Type`
- ❁ **Picking the right type representation is hard**
 - ❁ Jeannie's types are the result of several iterations

Can We Do Better?

- ❁ xtc is extensible, but supports only limited static typing
 - ❁ Dynamically typed syntax trees
 - ❁ Dynamically dispatched visitors
 - ❁ Dynamically typed type objects
 - ❁ Every type variable is declared as `xtc.type.Type`
- ❁ Picking the right type representation is hard
 - ❁ Jeannie's types are the result of several iterations

Yet, type checker correctness is critical!



Enter Typical

- 🔹 ML-based language for writing semantic analyses
 - 🔹 Builds on rigorous formal foundation
 - 🔹 Provides precise notation for type system's case analysis
 - 🔹 Variants (i.e., type-safe unions) and pattern matching
 - 🔹 Domain-specific, declarative extensions
 - 🔹 Seamlessly integrates with *Rats!* and xtc
 - 🔹 Automatically infers statically typed view of generic tree
 - 🔹 Compiles down to Java

The Simply Typed Lambda Calculus with Integers

$\lambda x:\text{int} . x$



The Simply Typed Lambda Calculus with Integers

$(\lambda x:\text{int} . x) 1$



The Simply Typed Lambda Calculus with Integers

$(\lambda x:\text{int} . x) 1$

$\Rightarrow [x \rightarrow 1] x$



The Simply Typed Lambda Calculus with Integers

$(\lambda x:\text{int} . x) 1$

$\Rightarrow [x \rightarrow 1] x$

$\Rightarrow 1$

The Simply Typed Lambda Calculus with Integers

$(\lambda x:\text{int} . x) 1$

$\Rightarrow [x \rightarrow 1] x$

$\Rightarrow 1$

$\lambda x:\text{int} \rightarrow \text{int} . x$



The Simply Typed Lambda Calculus with Integers

$(\lambda x:\text{int} . x) \ 1$

$\Rightarrow [x \rightarrow 1] x$

$\Rightarrow 1$

$(\lambda x:\text{int} \rightarrow \text{int} . x) \ \lambda x:\text{int} . x$

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * type_spec * expression
  | Application of expression * expression
  | Identifier of string
  | IntegerConstant of string ;
```

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

```
mltype raw_type =
  IntegerT | FunctionT of type * type ;
```

```
scope Abstraction (id, _, body) ->
  Scope(Anonymous("lambda"), [id, body]) ;
```

```
namespace default : type =
  Identifier (id) -> SimpleName(id) ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
          | FunctionT (param, res), param ->
              res
          | _ -> error "mistyped application"
        end
  | Abstraction (id, type, body) ->
      let param = get_type type in
      let _ = define id param in
      let res = analyze body in
      { type = FunctionT (param, res) }
  | Identifier _ as id ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

The Type Checker for the Simply Typed λ -Calculus

```

mltype expression =
  | Abstraction of
      expression * expression
  | Application expression
  | Identifier
  | Integer

mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;

mltype raw_type =
  IntegerT | FunctionT of type * type ;

scope Abstraction (id, _, body) ->
  Scope(Anonymous("lambda"), [id, body]) ;

namespace default : type =
  Identifier (id) -> SimpleName(id) ;

```

Automatically inferred
tree declaration

```

mlvalue analyze = function
  | Application (lambda, expr) ->
    let tl = analyze lambda
    and tr = analyze expr in begin
      match tl.type, tr with
      | FunctionT (param, res), param ->
        res
      | _ -> error "mismatched application"
    end
  | Abstraction (id, type, body) ->
    let param = get_type type in
    let _ = define id param in
    let res = analyze body in
    { type = FunctionT (param, res) }
  | Identifier _ as id ->
    lookup id
  | IntegerConstant _ ->
    { type = IntegerT } ;

mlvalue get_type = function
  | FunctionType (param, res) ->
    FunctionT (get_type param, get_type res)
  | IntegerType ->
    IntegerT ;

```

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application expression
  | Identifier
  | Integer
```

Automatically inferred tree declaration

```
mltype type_ =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

```
mltype type_ =
  IntegerType ;
```

Type representation

```
scope Abstraction (id, _, body) ->
  Scope(Anonymous("lambda"), [id, body]) ;
```

```
namespace default : type =
  Identifier (id) -> SimpleName(id) ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
          | FunctionT (param, res), param ->
              res
          | _ -> error "mistyped application"
        end
  | Abstraction (id, type, body) ->
      let param = get_type type in
      let _ = define id param in
      let res = analyze body in
      { type = FunctionT (param, res) }
  | Identifier _ as id ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application expression
  | Identifier
  | Integer
```

Automatically inferred
tree declaration

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

Type representation

```
scope Abstraction (id, body) ->
  Scope (Abstraction (id, body)) ;
```

Scoping and
namespaces

```
namespace Identifier ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
          | FunctionT (param, res), param ->
              res
          | _ -> error "mismatched application"
        end
  | Abstraction (id, type, body) ->
      let param = get_type type in
      let _ = define id param in
      let res = analyze body in
      { type = FunctionT (param, res) }
  | Identifier _ as id ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application expression
  | Identifier
  | Integer
```

Automatically inferred
tree declaration

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

Type representation

```
scope Abstraction (id : string, body) ->
  Scope (Abstraction id body) ;
```

Scoping and
namespaces

```
namespace Identifier ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
        | FunctionT (param, res), param ->
            res
        | _ ->
            "Application"
      end
  | Abstraction (id, body) ->
      let tl = analyze body
      let tr = analyze body
      let { type } = tl
      { type = res }
  | Identifier id ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;

mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

Type rules and
error checking



The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application of expression
  | Identifier
  | Integer
```

Automatically inferred tree declaration

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

Type representation

```
mltype scope =
  | Abstraction (id : string, body : scope)
  | Application (id : string, body : scope) ;
```

Scoping and namespaces

```
mltype namespace =
  | Identifier (id : string, type : type_spec) ;
```

```
mltype namespace =
  | Identifier (id : string, type : type_spec) ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
        | FunctionT (param, res), param ->
            res
        | _ ->
            "application"
      end
  | Abstraction (id, body) ->
      let tl = analyze body
      let tr = analyze body
      in { type = FunctionT (tl.type, tr.type) }
  | Identifier id ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

Type rules and error checking

9 + 29 lines of code

Inference of Tree Declaration

🔹 **Goal: provide a statically typed view on dynamic tree**

🔹 **Approach**

🔹 **Sort generic nodes into semantic categories, i.e., variants**

🔹 **“Abstraction, Application, Identifier, and IntegerConstant nodes belong to expression type”**

🔹 **Fill in types of nodes' children, i.e., constructors**

🔹 **“Application nodes have two children of expression type”**



```
module xtc.lang.TypedLambda ;

public variant Node Expression =
  Application EndOfFile ;

generic Application =
  Application BasicExpression
  / yyValue:BasicExpression ;

Node BasicExpression =
  Abstraction
  / Identifier
  / IntegerConstant
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

Add Application node

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

Add Application node

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

Add Abstraction,
Identifier, and
IntegerConstant nodes

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

Add Application node

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

Add Abstraction,
Identifier, and
IntegerConstant nodes

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =  
  Application EndOfFile ;
```

Type as Application of
expression * expression

```
generic Application =  
  Application BasicExpression  
  / yyValue:BasicExpression ;
```

```
Node BasicExpression =  
  Abstraction  
  / Identifier  
  / IntegerConstant  
  / OPEN Application CLOSE ;
```

Add Abstraction,
Identifier, and
IntegerConstant nodes

```
module xtc.lang.TypedLambda;
```

Create expression variant

```
public variant Node Expression =
```

8 variant annotations for C grammar

2 changes to grammar to avoid type errors

```
generic Application BasicExpression
```

7 variant annotations for Java 1.4 grammar

```
/ yyValue CustomExpression;
```

1 more annotation for Java 5 grammar

No more annotations for Jeannie grammar

```
Node BasicExpression =
```

Leverages annotations in C and Java grammars

```
Abstraction
```

```
/ Identifier
```

```
/ IntegerConstant
```

```
/ OPEN Application CLOSE ;
```

Add Abstraction,
Identifier and
IntegerConstant nodes

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application of expression
  | Identifier
  | IntegerConstant
```

Automatically inferred
tree declaration

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

Type representation

```
mltype scope =
  | IntegerConstant ;
```

```
scope Abstraction (id : string, body) ->
  Scope (Abstraction (id, body)) ;
```

Scoping and
namespaces

```
namespace Identifier ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
      let tl = analyze lambda
      and tr = analyze expr in begin
        match tl.type, tr with
          | FunctionT (param, res), param ->
              res
          | _ -> "Error: bad application"
        end
  | Abstraction (id, body) ->
      let tl = analyze body
      let tr = analyze body
      let { type } = tl
      { type = FunctionT (tr, res) }
  | Identifier _ ->
      lookup id
  | IntegerConstant _ ->
      { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
      FunctionT (get_type param, get_type res)
  | IntegerType ->
      IntegerT ;
```

Type rules and
error checking



λ -Calculus Type Representation

❖ Declare raw structure

```
m1type raw_type =  
  IntegerT | FunctionT of type * type ;
```

❖ Separately declare attributes

❖ λ -calculus: none

❖ C: storage class, qualifiers, constant value, ...

❖ Typical automatically combines declarations into ML record

```
m1type type = { type : raw_type } ;
```

λ -Calculus Type Representation

Declare raw structure

`ml type raw_type =
integer | function of type * type`

• Simplifies application-specific typing constraints

Separately declare attributes

• Update types relative to previous records

`{ record with field = new_value }`

• C: λ -calculus, storage class, quantifiers, constant value, ...

• Declare new attribute, add corresponding rules

• Typical automatically combines declarations into ML record

`ml type type = { type : raw_type } ;`



Scoping and Namespaces

❊ Declare scopes

```
scope Abstraction (id, _, body) ->  
  Scope(Anonymous("lambda"), [id, body]) ;
```

- ❊ Generated code automatically synchronizes current scope with syntax tree traversal (think concurrent visitor)

❊ Declare namespaces

```
namespace default : type =  
  Identifier (id) -> SimpleName(id) ;
```

- ❊ Generated code automatically maps nodes to <name, namespace> for symbol table operations

Scoping and Namespaces

⊕ Declare scopes

```
scope Abstraction (id, _, body) ->  
  Scope(Anonymous("lambda"), [id, body]) ;
```

⊕ Eliminates boiler-plate code

⊕ Generated code automatically synchronizes current scope

with symbols ⊕ Enter scope, exit scope, mangle name

⊕ Declare namespaces ⊕ Just access symbol table

```
namespace default : type =  
  Identifier (id) -> SimpleName(id) ;
```

⊕ define, is_defined, lookup

⊕ Generated code automatically maps nodes to <name, namespace>
for symbol table operations

The Type Checker for the Simply Typed λ -Calculus

```
mltype expression =
  | Abstraction of
      expression * expression
  | Application of expression
  | Identifier
  | IntegerConstant
```

Automatically inferred tree declaration

```
mltype type_spec =
  | FunctionType of type_spec * type_spec
  | IntegerType ;
```

Type representation

```
mltype scope =
  | IntegerConstant ;
```

Scoping and namespaces

```
scope Abstraction (id : string, body) ->
  Scope (Abstraction id body) ;
```

```
namespace Identifier ;
```

```
mlvalue analyze = function
  | Application (lambda, expr) ->
    let tl = analyze lambda
    and tr = analyze expr in begin
      match tl.type, tr with
      | FunctionT (param, res), param ->
        res
      | _ -> "Error: bad application"
    end
  | Abstraction (id, body) ->
    let tl = analyze body
    let tr = analyze body
    let { type } = tl
    { type = FunctionT (tl.type, tr.type) }
  | Identifier id ->
    lookup id
  | IntegerConstant _ ->
    { type = IntegerT } ;
mlvalue get_type = function
  | FunctionType (param, res) ->
    FunctionT (get_type param, get_type res)
  | IntegerType ->
    IntegerT ;
```

Type rules and error checking

Type Rules and Error Checking

❖ Traverse syntax tree and map nodes to their types

```
| Application (lambda, expr) ->
  let tl = analyze lambda
  let tr = analyze expr in begin
    match tl.type, tr with
      | FunctionT (param, res), param -> res
      | _ -> error "mismatched application" end
```

❖ Represent type errors by “no-information” monad

❖ Every Typical type has a bottom value (think null)

❖ Operations return bottom on bottom argument (unlike null)

❖ Error constructs silently pass through bottom,
thus avoiding cascading error messages

Type Rules and Error Checking

- Traverse syntax tree and map nodes to their types

```
| Application (lambda, expr) ->  
  let tl = analyze lambda  
  let tr = analyze expr in begin  
    match tl.type, tr with  
    | function! (param, res), param -> res  
    | _ -> error "mismatched application" end
```

- No need to thread error conditions through type checker**

- “If no previous error, check for new error”**

- Represent type errors by “no-information” monad

- Just check for and report errors**

- Every Typical type has a bottom value (think null)

- Operations return bottom on bottom argument (unlike null)

- Error constructs silently pass through bottom,
thus avoiding cascading error messages



Putting It All Together



Two Real-World Type Checkers

Typical

-  Implements higher-order functions, parametric polymorphism, parameterized data types, and Hindley-Milner type inference

-  Also checks patterns for irredundancy and exhaustiveness

 -  I.e., cover all cases with no duplicate clauses

-  Is used by Typical compiler (think self-hosting)

C: C99 with common gcc extensions

-  Type checks entire Linux 2.6 kernel, also gcc 4.1 regression tests

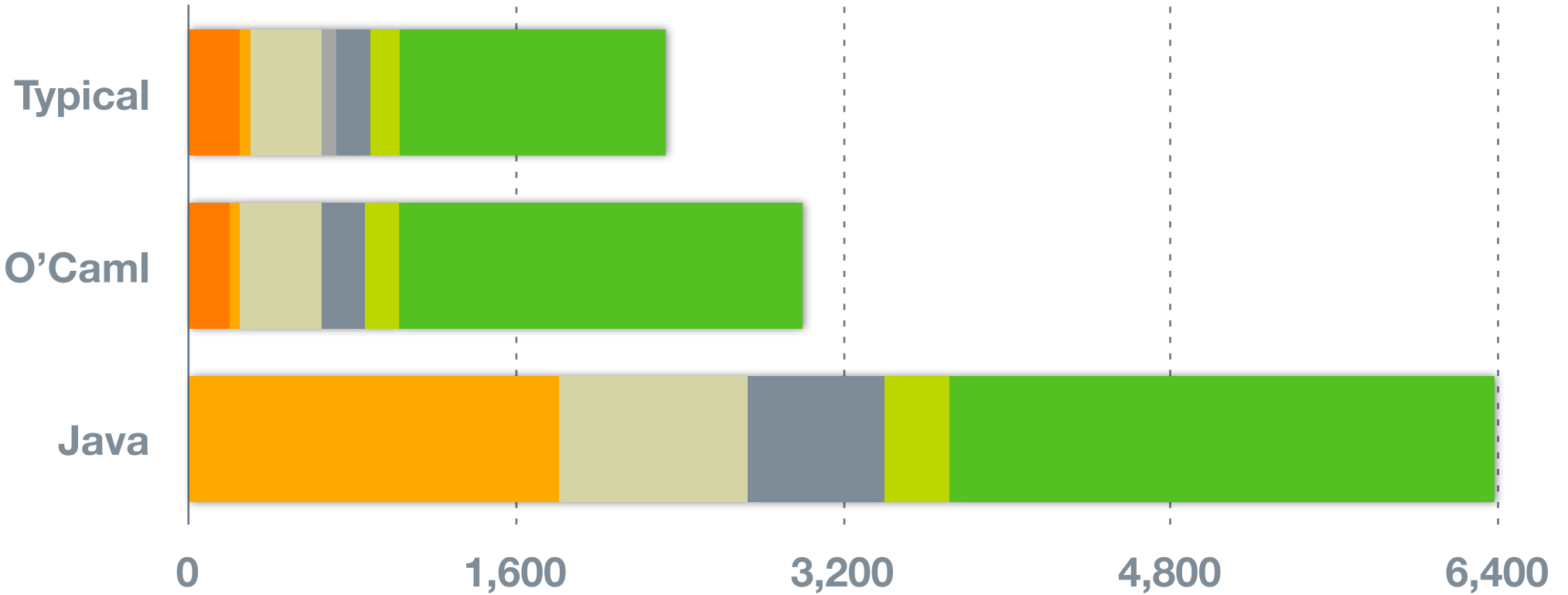
-  Is comparable to xtc's Java version and a port to O'Caml





Typical Is Concise

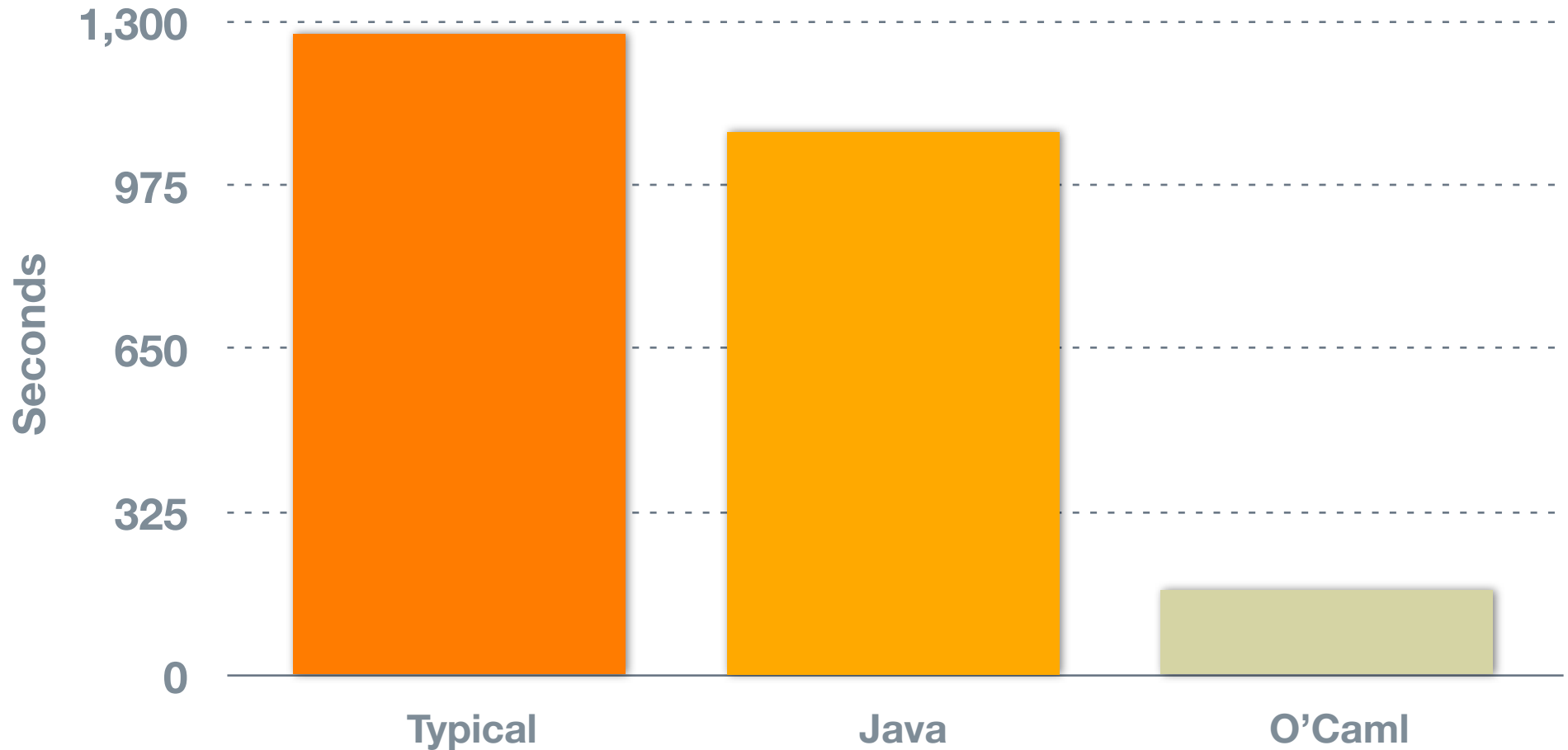
AST Types Type Ops Naming Specifiers Init. Rules



Lines of Code for Typical & O'Caml – NCSS for Java



Typical-Generated Code Performs Well Enough



Related Work

Attribute grammars: Silver, JastAdd, xoc

- Attribute grammars: Silver, JastAdd, xoc
 - Eliminate the need for manual scheduling of processing phases
 - But require complex attribute relationships, artificial productions

Rewrite rules: Stratego

- Rewrite rules: Stratego
 - Greatly simplify the transformation of syntax trees
 - But have little support for semantic analysis

Language support: J&

- Language support: J&
 - Provides much of the flexibility of dynamic nodes & visitors
 - But is less integrated with syntax, a little more verbose



One More Thing...



Rats! — From Systems to Programming Languages

- Iteration 1: Types and semantic actions are strings
- Iteration 2:
 - Resolve type names to classes, determine subtype relationships
 - Avoid dynamic instanceof tests
 - Regex match against “yyValue” in semantic actions
 - Allow actions in productions with automatically determined values
- Iteration 3: Represent types explicitly
 - Automatically deduce statically typed syntax tree
- Iteration 4: Parse semantic actions (in future version)
 - Support parameterized productions





Conclusions



Conclusions

- 🔷 *Rats!* and *xtc* make compilers more easily extensible
 - 🔷 Enable scalable composition of Java and C into Jeannie

Conclusions

- ❖ ***Rats!* and xtc make compilers more easily extensible**
 - ❖ Enable scalable composition of Java and C into Jeannie
- ❖ **Dynamic and static typing can be complementary**
 - ❖ Dynamic typing for flexibility
 - ❖ Static typing for safety

Conclusions

- ❖ ***Rats!* and xtc make compilers more easily extensible**
 - ❖ Enable scalable composition of Java and C into Jeannie
- ❖ **Dynamic and static typing can be complementary**
 - ❖ Dynamic typing for flexibility
 - ❖ Static typing for safety
- ❖ **Don't be afraid of functional programming**
 - ❖ Used to great effect by both *Rats!* and Typical



<http://cs.nyu.edu/rgrimm/xtc/>

