http://whiley.org

@whileydave

http://github.com/DavePearce/whiley

Victoria University
of Wellington, New Zealand
Te Whare Wananga o te Upoko o te Ika a Maui Aotearoa

The Whiley Programming Language

David J. Pearce
Victoria University of Wellington
New Zealand

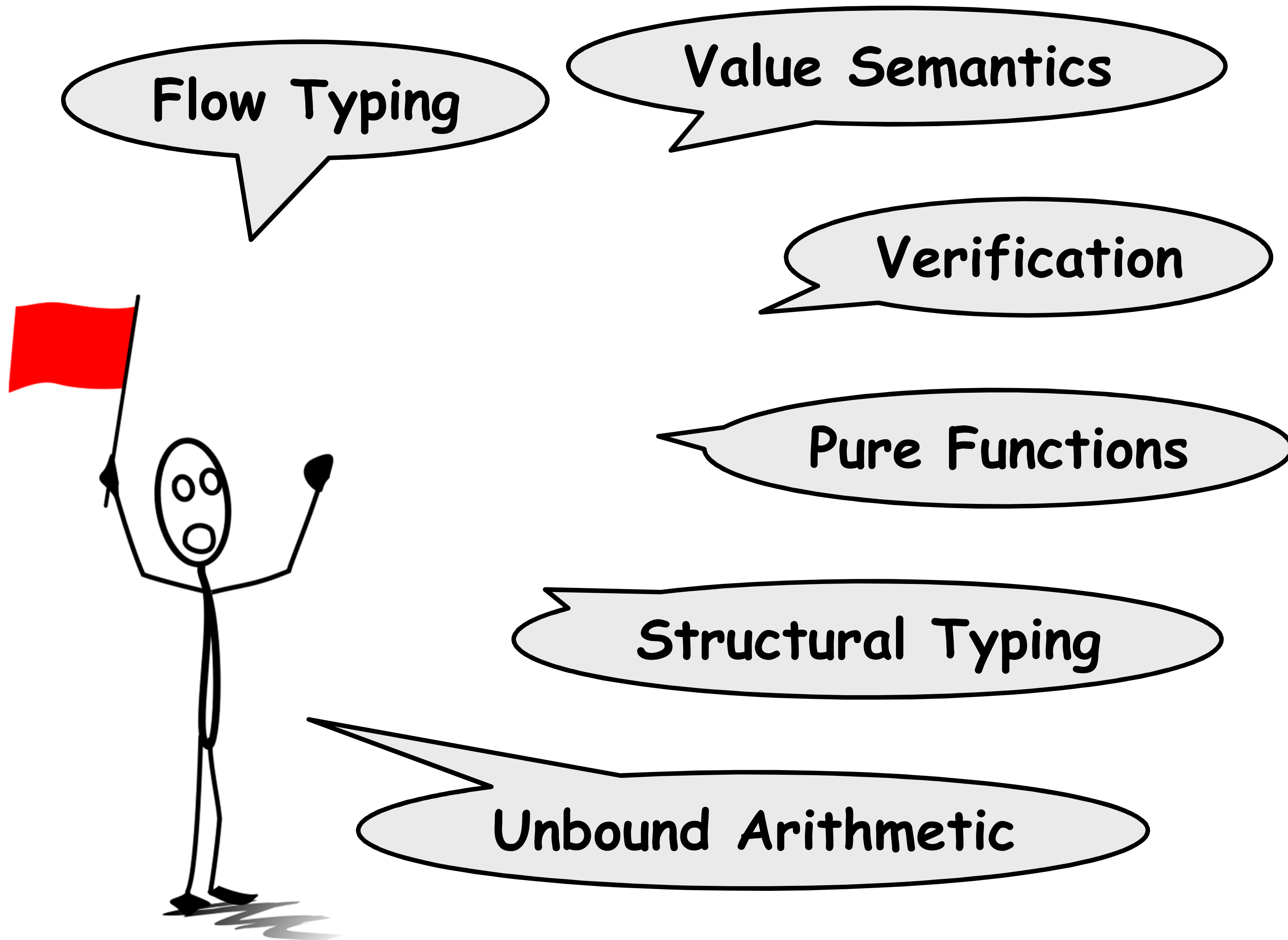
http://whiley.org

Overview

- What is Whiley?
  - **Hybrid** functional / imperative language
  - Designed specifically for **verification**
  - Compiles to **JVM** (also prototype C backend)

- Why another language?
  - Verification is **really hard**
  - Many features of Java it **even harder**!
  - I think it's basically **impossible** for Java
  - See **ESC/Java** and **JML** as good efforts here

What's Interesting about Whiley?



A Zoo of Unusual Types!

- **Primitives**:
  - e.g. `any` `null` `bool` `int` `real` `char`
- **Collections** (lists, sets, maps):
  - e.g. `[int]` `{string}` `{int=>string}`
- **Records** and **Tuples:**
  - e.g. `{int x, int y}` `(int,int)`
- **Unions** and **Intersections**:
  - e.g. `int|null` `int&null`
- **Negations**
  - e.g. `!int`

Flow Typing

Flow Typing

```
int sum([int] items):
    r = 0
    for item in items:
        r = r + item
    return r
```

- A **flow-sensitive** approach to type checking
- Types declared only for **parameters and returns**
- Variables can have **different types**!
- Conditionals and/or assignments cause **retyping**

Flow Typing

```
define Circle as {int x, int y, int r}
define Rect as {int x, int y, int w, int h}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

- Type tests **automatically retype** variables!
  - (even on the false branch)

Flow Typing & Unions

```
null|int indexOf(string str, char c):
   …
[string] split(string str, char c):
     idx = indexOf(str,c)
     …
```

- Cannot treat `null|int` like an `int`
- Must distinguish cases by **explicit type testing**

Flow Typing & Unions

```
null|int indexOf(string str, char c):
    …

[string] split(string str, char c):
    idx = indexOf(str,c)
    if idx is int:
        below = str[0..idx]
        above = str[idx..]
        return [below,above]
    else:
        return [str]
```

Can safely treat `x` as `int` here

- Cannot treat `null|int` like an `int`
- Must distinguish cases by **explicit type testing**

Flow Typing & Recursive Types

```
define LinkedList as null | Link
define Link as {int dat, LinkedList next}

int sum(LinkedList l):
  if l == null:
    return 0
  else:
    return l.dat + sum(l.next)
```

- Support general **tree-like** structures, similar to ADTs
- Like ADTs, recursive types also have **value semantics**

Verification

Verification

```
define nat as int where $ >= 0

nat f(int x):
    return x
```

A compile time error!

- Function `f()`:
  - Accepts an **arbitrary** integer …
  - Should return a **natural** number …
  - But, this implementation is **broken**!

```
define nat as int where $ >= 0

nat f(int x):
    if x >= 0:
        return x
    else:
        return 0
```

OK, because x implicitly a nat

- Function `f()`:
  - Accepts an **arbitrary** integer …
  - Returns a **natural** number …
  - This implementation **satisfies** the spec!

Verification

```
define nat as int where $ >= 0
define pos as int where $ > 0

nat g(pos x):
    return x
```

OK, because pos implies nat

- Function `g()`:
  - Accepts a **positive** number ...
  - And returns a **natural** number ...
  - But, how to know `pos` subtypes `nat` ?

Verification

```
define nat as int where $ >= 0
define pos as int where $ > 0

pos h(nat x):
    return x + 1
```

OK, because nat+1 gives pos

- Function `h()`:
  - Accepts a **natural** number ...
  - And returns a **positive** number ...
  - But, how to know `nat+1` gives `pos` ?

Verification

```
define nat as int where $ >= 0
define pos as int where $ > 0

pos h1(nat x):
    return x + 1

int h2(int x) requires x>=0, ensures $>0:
    return x + 1
```

- Function `h1()` and `h2()` are identical

Verification

```
define nat as int where $ >= 0

nat sum([nat] list):
   r = 0
   for x in list where r >= 0:
       r = r + x
   return r
```

Ok, because adding nat to nat gives nat

- Function `sum()`:
  - Accepts a **list of natural** numbers …
  - Then **adds** them together …
  - And returns a **natural** number.

Value Semantics

Value Semantics

```
define Point as {int x, int y}

Point translate(Point p, int x, int y):
    p.x = p.x + x
    p.y = p.y + y
    return p
```

Such assignments don't affect caller's state

- **Everything** is pass-by-value (a.k.a value semantics)
- Data propagates only via **return**
- I/O and other side-effects **not permitted**
- Data may be updated **in place**

Value Semantics – *Performance*

```
define int18 as int where 1 <= $ && $ <= 8
define Pos as { int18 row, int18 col }

Board move(Board b, Pos o, Pos n, Piece p):
    b[o.col][o.row] = null
    b[n.col][n.row] = piece
    return b
```

- Value semantics (naïve implementation):
  - **Copy** board for call to move()
  - Copy again **for each** assignment in move()
  - This is very **inefficient**!!!

- **Reference counting** can really help here...

Value Semantics - *Thoughts*

- Item 24, Effective Java
  - Make Defensive Copies when Needed

"It is essential to make a defensive copy of each mutable parameter to the constructor"

-- Josh Bloch

Structural Subtyping

Structural Subtyping

```
define IntList as null | IntLink
define IntLink as {int dat, IntList next}

define AnyList as null | AnyLink
define AnyLink as {any dat, AnyList next}

AnyList f(IntList l):
    return l
```

- Types are **structural** not nominal (like e.g. Java)
- Here, `IntList` implicitly subtypes `AnyList`
- No equivalent to "extends" or "implements"

Structural Subtyping

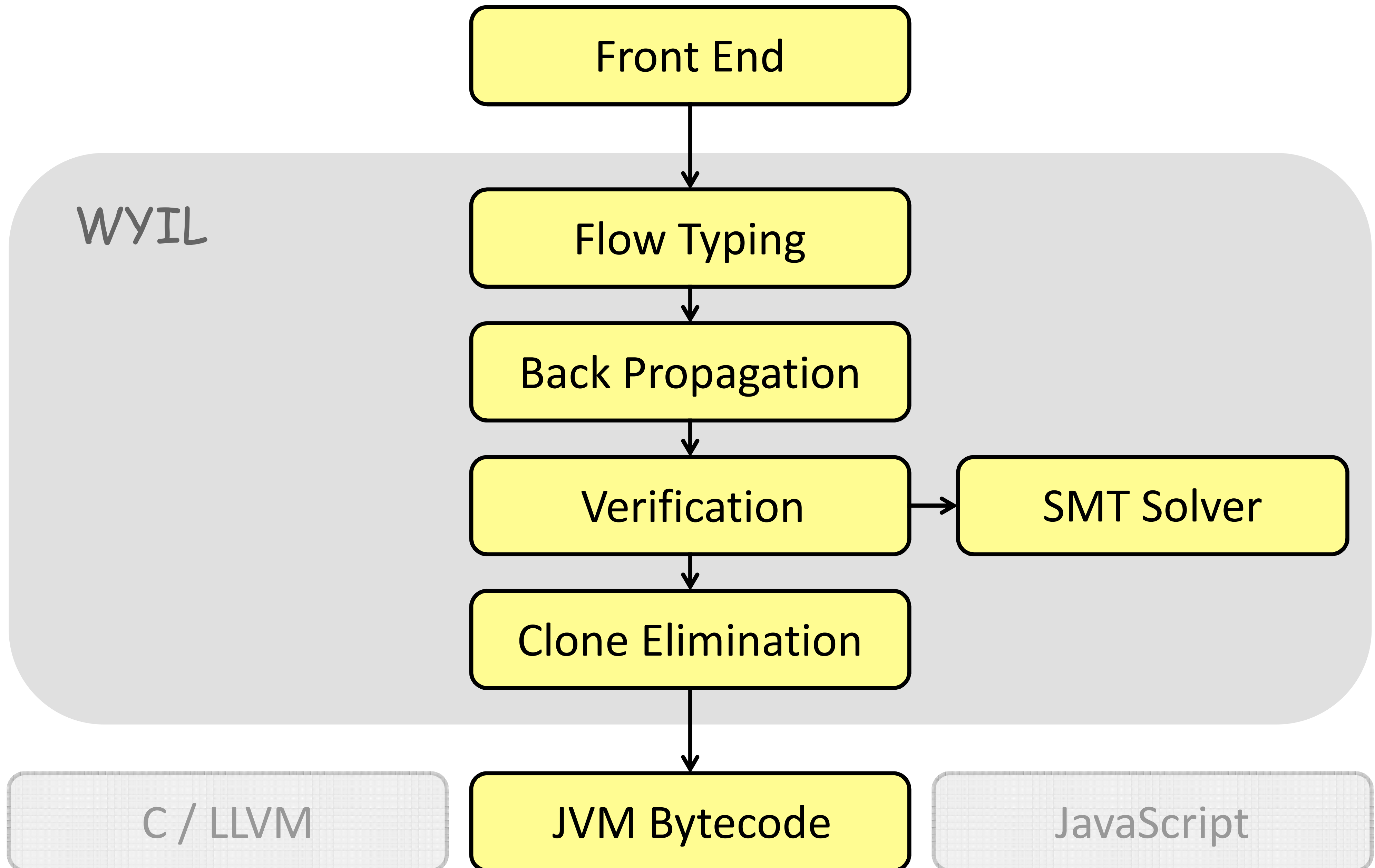
```
public define Rectangle as {int x, int y}
public define Border as {int x, int y}

real area(Rectangle r):
    return r.x * r.y
```

- `Rectangle` and `Border` indistinguishable
- Can be in different **files** and **packages**
- Can be written by **different** people at **different** times

Implementation

Compiler Overview

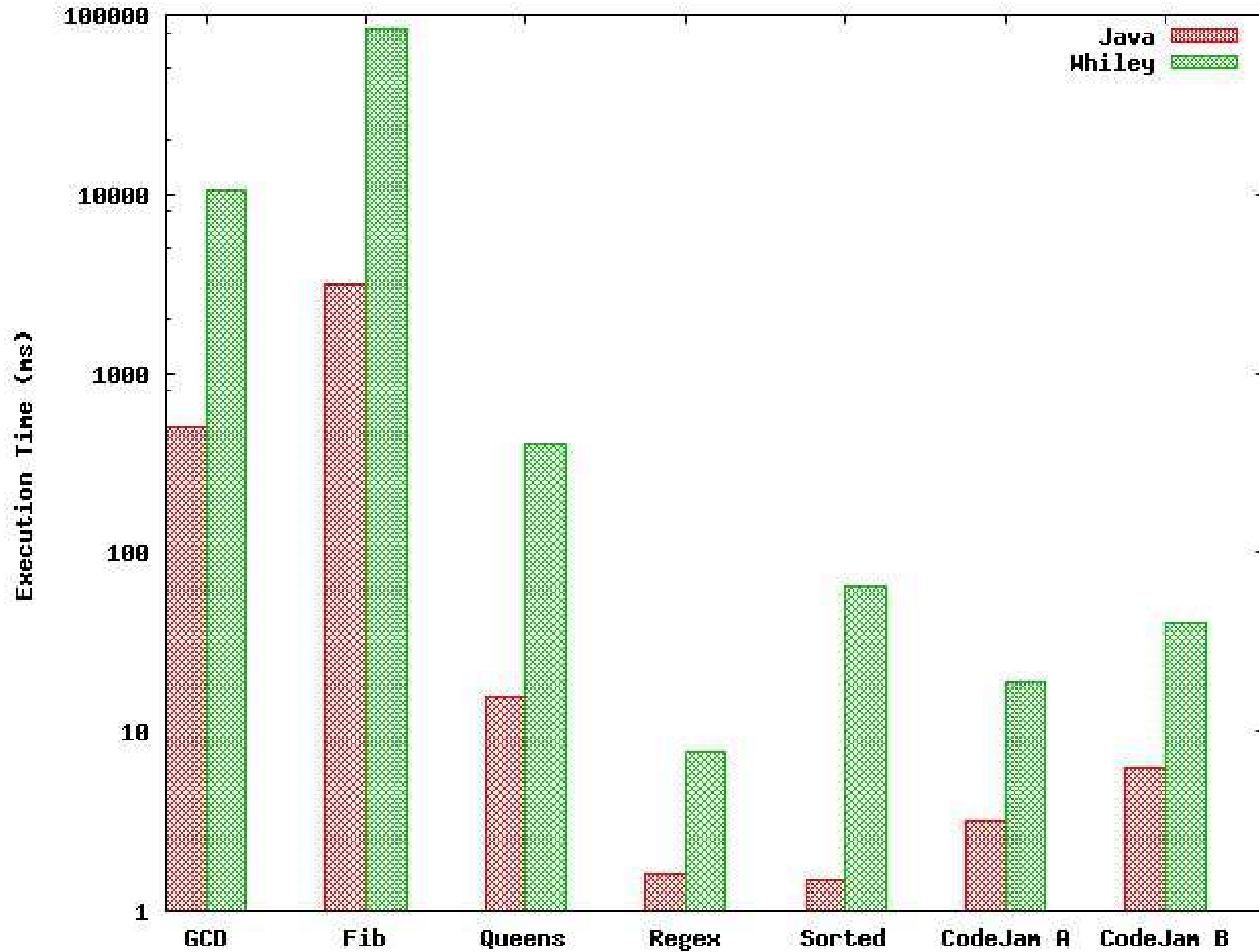


Whiley Intermediate Language

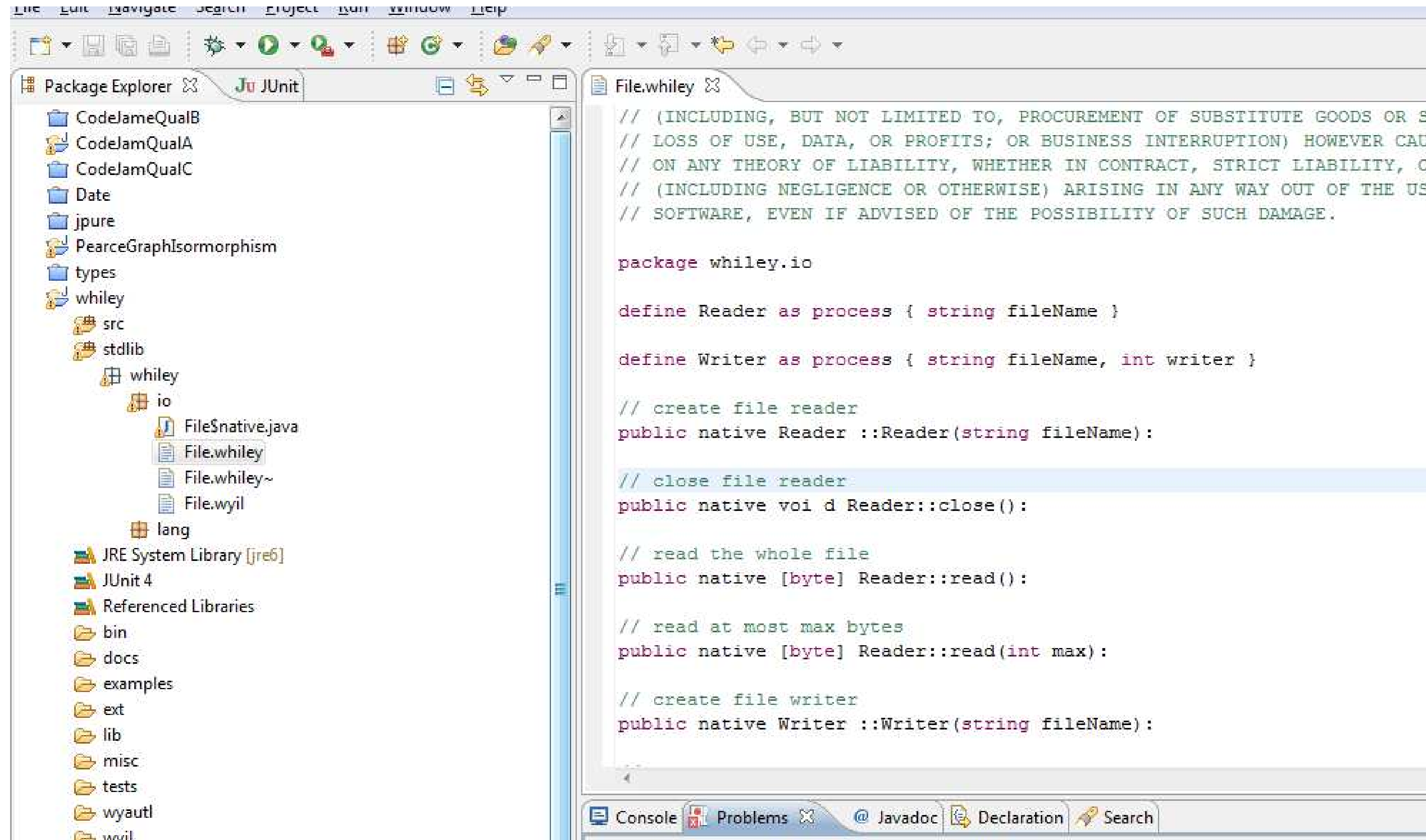
```
int f(int):
ensures:
    const %1 = 0
    assertge %0, %1
body:
    const %1 = 0
    iflt %0, %1 goto label
    return %0
.label
    neg %0 = %0
    return %0
```

```
nat f(int x):
    if x >= 0:
        return x
    else:
        return -x
```

Performance



Eclipse Plugin



- **Update Site:** http://whiley.org/eclipse

http://whiley.org

@whileydave

http://github.com/DavePearce/whiley