Program Representations & Parsing
Program Representations

• What are these 'representations'?
Program Representations

- What are these 'representations'?
- What are they for?
Program Representations

• What are these 'representations'?
• What are they for?
• How can they be created?
Program Representations

• What are these 'representations'?
• What are they for?
• How can they be created?
• What can they do, what can't they do?
Program Representations

• What are these 'representations'?  
• What are they for?  
• How can they be created?  
• What can they do, what can't they do?  
• What do they look like in practice?
Program Representations

- What are these 'representations'?  
- What are they for?  
- How can they be created?  
- What can they do, what can't they do?  
- What do they look like in practice?  
- What tools can we use to work with different representations?
Program representations: an overview
Representations for Creation

Source Code

```java
/**
 * The Hello class implements an application that
 * simply prints "Hello World!" to standard output.
 */
public class Hello {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```
Representations for Creation

Source Code

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Visual Programming Languages
Representations for Description

UML
Representations for Description

**UML**

- **SelectionPanel**
  - `productName`
  - `status`
  - `sendStatus()`

- **Controller**
  - `+Id`
  - `+dispenseProduct()` (Output: Product)
  - `+dispenseCoin()` (Output: Coin)
  - `+comparePrice()`
  - `+checkAvailability()`
  - `+updateProductQuantity()`
  - `productQuantity` (Input: Product)

- **CoinCollector**
  - `+coinPrice`
  - `+count`
  - `+countCoinQ()`

- **CoinDispenser**
  - `+countQ()`
  - `+dispense()`

- **Product**
  - `+price`
  - `+quantity`
  - `+updateQuantity()`
  - `+discardProduct()`

- **ProductDispenser**
  - `+product`
  - `+dispense()`

- **Call Diagram**

  1. **SelectionPanel**
  2. **CoinCollector**
  3. **Product**
  4. **CoinDispenser**
  5. **Controller**
  6. **ProductDispenser**

  Events:
  - 1: Select Item
  - 2: Dispense Product
  - 3: Deliver Coin
  - 4: Check Price
  - 5: Update Quantity
  - 6: Dispense Coin
  - 7: Dispense Product

  States:
  - 1: Select Item
  - 2: Dispense Product
  - 3: Deliver Coin
  - 4: Check Price
  - 5: Update Quantity
  - 6: Dispense Coin
  - 7: Dispense Product
Representations for Description

UML

Controller
- id
- dispenseProduct()
- dispenseCoin()
- checkQuantity()
- dispense()

SelectionPanel
- productName
- status
- sendStatus()

CoinCollector
- coin
- count
- coins

CoinDispenser
- coinCount
- dispense()

Product
- price
- quantity
- updateQuantity()
- retrieveProduct()

ProductDispenser
- product
- dispense()

Call Diagram

“Code City”
## Representations for Execution

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. C, C++</td>
<td></td>
</tr>
</tbody>
</table>
Representations for Execution

Source Code → Assembly / Machine Code

- e.g. C, C++

Source Code → Bytecode → Assembly / Machine Code

- e.g. Java (Java Bytecode), C# (Common Intermediate Language) > JIT
### Representations for Execution

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. C, C++</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Bytecode</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Java (Java Bytecode), C# (Common Intermediate Language)</td>
<td>&gt; JIT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Lower level language source code</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Haskell (Core &gt; STG &gt; C-- &gt; C/ASM/LLVM)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Representations for Execution

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Bytecode</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. C, C++</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Bytecode</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Java (Java Bytecode), C# (Common Intermediate Language)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Lower level language source code</th>
<th>Assembly / Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Haskell (Core &gt; STG &gt; C- - &gt; C/ASM/LLVM)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Representations to work with
Intermediary Representations

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Intermediary Representations

Concrete Syntax Tree (CST)

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Intermediary Representations

Concrete Syntax Tree (CST)

```
226  compilationUnit
227   :   packageDeclaration? importDeclaration* typeDeclaration* EOF
228

261  typeDeclaration
262   :   classDeclaration
263     | interfaceDeclaration
264     | ';'
265

271  classDeclaration
272   :   normalClassDeclaration
273     | enumDeclaration
274     ;
275
276  normalClassDeclaration
277   :   classModifier* 'class' Identifier typeParameters? superclass? superinterfaces? classBody
278     ;

1729  Identifier
1730   :   JavaLetter JavaLetterOrDigit*
1731   ;

21
```

```
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Concrete Syntax Tree (CST)

```java
public class Demo {
    public void run() {
        for (int i = 1; i< 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Intermediary Representations

Concrete Syntax Tree (CST)

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Intermediary Representations

Concrete Syntax Tree (CST)
- 1:1 mapping of source code to a tree representation
- Consists of simple tokens (i.e. strings, numbers)

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
Concrete Syntax Tree (CST)
• 1:1 mapping of source code to a tree representation
• Consists of simple tokens (i.e. strings, numbers)

Abstract Syntax Tree (AST)
• Only necessary parts
• Possibly consisting of complex node objects
• Possibly restructured for simplicity
Intermediary Representations

Concrete Syntax Tree (CST)
- 1:1 mapping of source code to a tree representation
- Consists of simple tokens (i.e. strings, numbers)

Abstract Syntax Tree (AST)
- Only necessary parts
- Possibly consisting of complex node objects
- Possibly restructured for simplicity
Intermediary Representations

Concrete Syntax Tree (CST)
- 1:1 mapping of source code to a tree representation
- Consists of simple tokens (i.e. strings, numbers)

Abstract Syntax Tree (AST)
- Only necessary parts
- Possibly consisting of complex node objects
- Possibly restructured for simplicity

Implementation specific object model
- E.g. representation created/used by the compiler
- Referencing other parts of the system
Intermediary Representations

Concrete Syntax Tree (CST)
• 1:1 mapping of source code to a tree representation
• Consists of simple tokens (i.e. strings, numbers)

Abstract Syntax Tree (AST)
• Only necessary parts
• Possibly consisting of complex node objects
• Possibly restructured for simplicity

Implementation specific object model
• E.g. representation created/used by the compiler
• Referencing other parts of the system

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```
### Intermediary Representations

#### Concrete Syntax Tree (CST)
- 1:1 mapping of source code to a tree representation
- Consists of simple tokens (i.e. strings, numbers)

#### Abstract Syntax Tree (AST)
- Only necessary parts
- Possibly consisting of complex node objects
- Possibly restructured for simplicity

#### Implementation specific object model
- E.g. representation created/used by the compiler
- Referencing other parts of the system

---

```java
public class Demo {
    public void run() {
        for (int i = 1; i < 100; i++) {
            if (i % 3 == 0 || i % 5 == 0) {
                System.out.println(i);
            }
        }
    }
}
```

---

**Flowchart**

1. **parse**
2. **resolve**
3. **compile?**
   - Fully qualified object model
   - ...
From source code to...
From Source code to ...

1) Lexing and Parsing
2) Fully qualified name resolution
3) Building an object model
4) Writing output from the model
   • Bytecode / assembly / machine code
   • Source code of a different language
   • Code metrics
   • Metadata, e.g. lookup tables, API documentation
1) Lexing and Parsing
1) Lexing and Parsing

- **Scanner**
  - read one char at a time
  - identify tokens
  - build CST/AST

- **Lexer**

- **Parser**
1) Lexing and Parsing

- **Lexer**
  - read one char at a time
  - identify tokens
  - build CST/AST

- **Parser**
  - feeds
  - Calls

**Diagram**:
- Scanner feeds to Lexer
- Lexer calls Parser
- Parser calls Scanner
1) Lexing and Parsing

read one char at a time  Scanner

identify tokens  Lexer

build CST/AST  Parser

if (x <= 3) { x = 3; }
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next token

Parser

build CST/AST

if (x <= 3) { x = 3; }
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

build CST/AST
Parser

if (x <= 3) { x = 3; }

give me the next char
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

```
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next char

build CST/AST

Parser

if (x <= 3) { x = 3; }

i
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next char

build CST/AST

Parser

if (x <= 3) { x = 3; }

if
1) Lexing and Parsing

- **Lexer**: read one char at a time
- **Scanner**: identify tokens
- **Parser**: build CST/AST

```java
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

- Read one char at a time
- Identify tokens
- Build CST/AST

Scanner

Lexer

Parser

```
if (x <= 3) { x = 3; }
```

```
if (keyword)
```
1) Lexing and Parsing

Scanner

identify tokens

Lexer

read one char at a time

Parser

build CST/AST

if (x <= 3) { x = 3; }

if (keyword)

if

thank you
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

if (x <= 3) { x = 3; }

if (keyword)

if

845 ifThenStatement
846 : 'if' '(' expression ')' statement
847 ;
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

```java
if (x <= 3) { x = 3; }
```

```
if (keyword)
```

```
ifThenStatement
: 'if' '(' expression ')' statement
;```
1) Lexing and Parsing

- **Lexer**: read one char at a time, identify tokens
- **Scanner**: feeds, calls, give me the next token
- **Parser**: feeds, calls, build CST/AST

```java
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

read one char at a time  
Scanner

identify tokens  
Lexer

build CST/AST  
Parser

give me the next char

if (x <= 3) { x = 3; }

if_

exp
stmt
1) Lexing and Parsing

- **Lexer**: reads one char at a time
- **Scanner**: identifies tokens
- **Parser**: builds CST/AST

Example code:
```
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

- **Read one char at a time**: Scanner
- **Identify tokens**: Lexer
- **Build CST/AST**: Parser

```
if (x <= 3) { x = 3; }
```

```
if_ (

(exp stmt

Not interested

(stmt

( (opening brace)
```
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

build CST/AST
Parser

give me the next token

if (x <= 3) { x = 3; }
1) Lexing and Parsing

read one char at a time

identify tokens

build CST/AST

Scanner

Lexer

Parser

if \( x \leq 3 \) \{ x = 3; \}
1) Lexing and Parsing

- Read one char at a time: Scanner
- Identify tokens: Lexer
- Build CST/AST: Parser

Example code:
```plaintext
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next char

build CST/AST

Parser

```c
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

build CST/AST
Parser

if \( x \leq 3 \) \{ \ x = 3; \ }

if \_\_ \_ \_\_ 

if

exp

stmt
1) Lexing and Parsing

read one char at a time  Scanner
identify tokens  Lexer
build CST/AST  Parser

if (x <= 3) { x = 3; }

if_ (x_

x (variable id)

if

exp
stmt
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if (x_

x (variable id)

if

exp

stmt

x
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

give me the next token
Parser

if (x <= 3) { x = 3; }

if__(x__
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next char

build CST/AST

Parser

if \( (x \leq 3) \) { \ x = 3; }

if\_\_(x\_}
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

```
if (x <= 3) { x = 3; }
```

```
if(x_
```
1) Lexing and Parsing

read one char at a time

Scanner

give me the next char

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if_(x_<

if

exp

stmt

x
1) Lexing and Parsing

read one char at a time → Scanner
identify tokens → Lexer
build CST/AST → Parser

```
if (x <= 3) { x = 3; }
```

`if_(x_<=`)

`<=(lessthan or equal)`

```
if
  exp
  stmt
x
```
1) Lexing and Parsing

read one char at a time  
Scanner

identify tokens  
Lexer

if (x <= 3) { x = 3; }

if (x <=

<= (lessthan or equal)

if

exp

stmt

x

relationalExpression
relationalExpression '<' shiftExpression
relationalExpression '>' shiftExpression
relationalExpression '<=' shiftExpression
relationalExpression '>=' shiftExpression
relationalExpression 'instanceof' referenceType
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

```java
if (x <= 3) { x = 3; }
```

```java
if (x <=
```

```java
<= (lessthan or equal)
```

```java
relationalExpression
    : shiftExpression
    | relationalExpression '<' shiftExpression
    | relationalExpression '>' shiftExpression
    | relationalExpression '<=' shiftExpression
    | relationalExpression '>=' shiftExpression
    | relationalExpression 'instanceof' referenceType
```

```
exp
stmt
```

```
x exp
```
1) Lexing and Parsing

read one char at a time
Scanner
identify tokens
Lexer
build CST/AST
Parser

if (x <= 3) { x = 3; }
if (x_<=

if (exp <= stmt)

exp
stmt
x
exp

<=
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

build CST/AST
Parser

give me the next char

if (x <= 3) { x = 3; }
if_(x_<=
1) Lexing and Parsing

- **Lexer** reads one char at a time.
- **Scanner** identifies tokens.
- **Parser** builds CST/AST.

Example code snippet:
```
if (x <= 3) { x = 3; }
```
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

give me the next char

if (x <= 3) { x = 3; }

if_(x_<=_}
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

```
if (x <= 3) { x = 3; }
```

```
if_(x_<=_3
```
1) Lexing and Parsing

read one char at a time

Scanner

give me the next char

Lexer

identify tokens

Parser

build CST/AST

if (x <= 3) { x = 3; }

if _(x_<=_3
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)

3 (number literal)

if

exp stmt

x exp

<=

parse tree
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)

3 (number literal)

hold on to that...

if

exp

stmt

<=

x

exp
1) Lexing and Parsing

read one char at a time  
Scanner
  feeds
  Calls
  identify tokens  
Lexer
  feeds
  Calls
  build CST/AST  
Parser

if (x <= 3) { x = 3; }

if_(x_<=_3

3 (number literal)
1) Lexing and Parsing

read one char at a time  
Scanner

identify tokens  
Lexer

build CST/AST  
Parser

if (x <= 3) { x = 3; }

if (x_<=_3)

3 (number literal)

if

exp

stmt

<=

x

exp

3
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

give me the next token

Parser

build CST/AST

if (x <= 3) { x = 3; }

if_(x <= 3
1) Lexing and Parsing

read one char at a time → Scanner → give me the next char
identify tokens → Lexer
build CST/AST → Parser

```java
if (x <= 3) { x = 3; }
```

```
if_(x_<=_3
```
1) Lexing and Parsing

read one char at a time
Scanner
identify tokens
Lexer
build CST/AST
Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)
1) Lexing and Parsing

read one char at a time
Scanner
identify tokens
Lexer
build CST/AST
Parser

```java
if (x <= 3) { x = 3; }
```

```java
if_(x_<=_3)
```

```java
) (closing brace)
```
1) Lexing and Parsing

- **Scanner**: reads one char at a time
- **Lexer**: identifies tokens
- **Parser**: builds CST/AST

```java
if (x <= 3) { x = 3; }
```

Good to know:
- if(x <= 3)
- ) (closing brace)
1) Lexing and Parsing

read one char at a time → Scanner
identify tokens → Lexer
build CST/AST → Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)

) (closing brace)
1) Lexing and Parsing

read one char at a time

Scanner

identify tokens

Lexer

build CST/AST

Parser

if (x <= 3) { x = 3; }

if(x_<=_3)

) (closing brace)
1) Lexing and Parsing

read one char at a time
Scanner
identify tokens
Lexer
build CST/AST
Parser

if (x <= 3) { x = 3; }

if_(x_<=_3)
1) Lexing and Parsing
1) Lexing and Parsing

read one char at a time
Scanner

identify tokens
Lexer

build CST/AST
Parser

if (x <= 3) { x = 3; }

... (opening bracket)
x (variable id)
= (assignment op)
3 (number literal)
; (statement)
} (closing bracket)
2) Fully qualified name resolution

```java
package com.hertz.core;
interface Rental { ... }

package com.hertz.core;
import com.hertz.billing.Bill;
public class Car implements Rental {
    ... Bill b = new Bill();
}

package com.hertz.billing;
import org.apache.commons.math3.fraction.Fraction;
public class Bill {
    ... Fraction f = new Fraction(3,5);
    f.add(5);
    ...)
```
2) Fully qualified name resolution

```
package com.hertz.core;
interface Rental { ... }
```

```
package com.hertz.core;
import com.hertz.billing.Bill;
public class Car implements Rental {
    ... Bill b = new Bill();
}
```

```
package com.hertz.billing;
import org.apache.commons.math3.fraction.Fraction;
public class Bill {
    ... Fraction f = new Fraction(3,5);
    f.add(5);
    ... 
}
```
2) Fully qualified name resolution

```java
class Car implements Rental {
    ... Bill b = new Bill();
}
```

```java
class Bill {
    ... Fraction f = new Fraction(3,5);
    f.add(5);
    ...
}
```

```java
package com.hertz.core;
interface Rental {
    ...
}
```

```java
package com.hertz.core;
import com.hertz.billing.Bill;
import com.hertz.core.Car;
public class Bill {
    ... Rental r = new Bill();
    ...
}
```

```java
package com.hertz.billing;
import org.apache.commons.math3.fraction.Fraction;
public class Bill {
    ... Fraction f = new Fraction(3,5);
    f.add(5);
    ...
}
```
2) Fully qualified name resolution

```java
interface com.hertz.core.Rental { ... }

public class com.hertz.core.Car implements com.hertz.core.Rental {
    ... com.hertz.billing.Bill b = new com.hertz.billing.Bill();
}

package com.hertz.billing;
import org.apache.commons.math3.fraction.Fraction;
public class com.hertz.billing.Bill {
    ... org.apache.commons.math3.fraction.Fraction f =
        new org.apache.commons.math3.fraction.Fraction(3,5);
        f.add(5);
    ... }
```
3) Building an Object model
4) Writing output from the model

- Bytecode / Assembly / Machine code
  - Generate machine or VM executable code
4) Writing output from the model

- Bytecode / Assembly / Machine code
  - Generate machine or VM executable code
- Source code of a different language
  - Code synthesis: generating source code from the object model
4) Writing output from the model

- **Bytecode / Assembly / Machine code**
  - Generate machine or VM executable code
- **Source code of a different language**
  - Code synthesis: generating source code from the object model
- **Code metrics**
  - Analyze the object model to compute metrics, i.e. number of classes, methods, complexity etc. > next week
4) Writing output from the model

- Bytecode / Assembly / Machine code
  - Generate machine or VM executable code
- Source code of a different language
  - Code synthesis: generating source code from the object model
- Code metrics
  - Analyze the object model to compute metrics, i.e. number of classes, methods, complexity etc. > next week
- Metadata, e.g. lookup tables, API documentation
  - IDE needs info on classes/methods for code completion
  - HTML API Docs are hyperlinked
Some examples!
A concrete example

• **Interactive:**
  1) Write hello world
  2) Use a parser (ANTLR) to build a CST
  3) Use a compiler (Javac) to create bytecode
  4) Use a decompiler (Javap/Javac/HDF) to show assembly
  5) Use ASM to modify bytecode
Tools
Working with Source code

• Syntax Highlighters
  – Pygments, Highlight, ruby-rouge

• Parser Generators
  – ANTLR, YACC

• Language-specific parsers
  – javac, Roslyn (C#), cpython all have APIs
Compilers have APIs

- Javac, Eclipse JDT Core provide in-depth APIs for compiled project exploration and manipulation
- Other languages have their own tools
Decomp. / Bytecode Manipulation

• **OW2 ASM**
  - Small and fast, a bit hard to learn

• **Apache BCEL**
  - Old, much slower, but easy to use

• **Java agents**
  - Usually used at runtime
  - E.g. monitoring, debugging, even exploits...
To conclude...
Program Representations

• What are these 'representations' and what are they for?
  – Depends on what you need!
Program Representations

• What are these 'representations' and what are they for?
  – Depends on what you need!

• How can they be created?
  – Programming language native and 3\textsuperscript{rd} party tools
Program Representations

• **What are these 'representations' and what are they for?**
  - Depends on what you need!

• **How can they be created?**
  - Programming language native and 3\textsuperscript{rd} party tools

• **What can they do, what \textit{can't} they do?**
  - Depends, e.g. parsers only know file-local knowledge
Program Representations

- **What are these 'representations' and what are they for?**
  - Depends on what you need!

- **How can they be created?**
  - Programming language native and 3rd party tools

- **What can they do, what *can't* they do?**
  - Depends, e.g. parsers only know file-local knowledge

- **What do they look like in practice?**
  - Complicated at times, requires some getting used to!
Program Representations

- What are these 'representations' and what are they for?
  - Depends on what you need!

- How can they be created?
  - Programming language native and 3rd party tools

- What can they do, what *can't* they do?
  - Depends, e.g. parsers only know file-local knowledge

- What do they look like in practice?
  - Complicated at times, requires some getting used to!

- What tools can we use to work with different representations?
  - Parsers, Compilers and all kinds of transformation tools
Program Representations & Parsing

Thanks - and keep experimenting! -