# SWEN430 - Compiler Engineering

## Lecture 3 - Parsing I

David J. Pearce

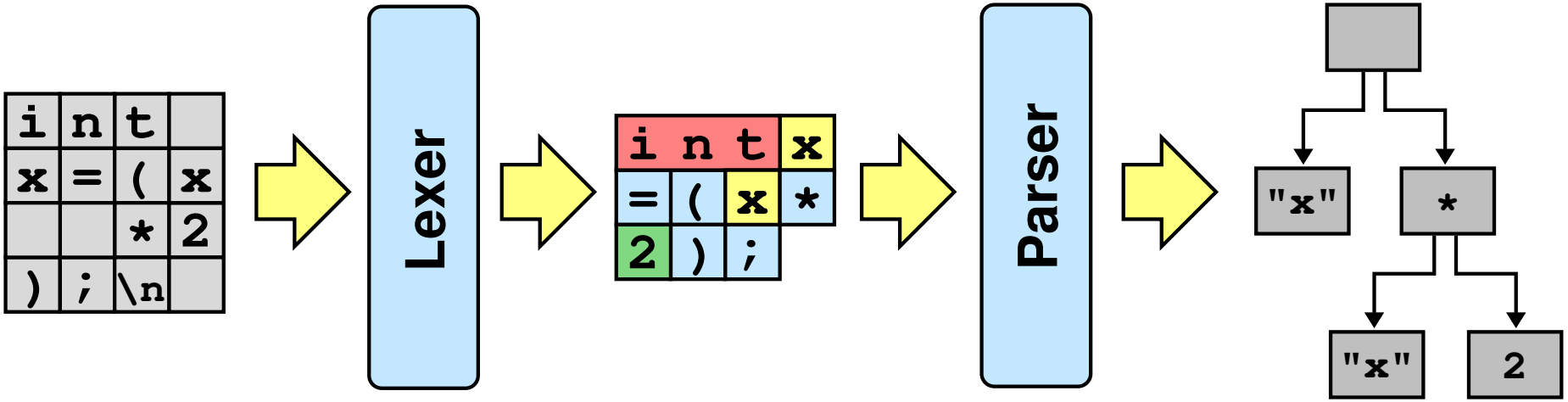
*School of Engineering and Computer Science*
*Victoria University of Wellington*

# Parsing

*"***Parsing***,* **syntax analysis***, or* **syntactic analysis** *is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin* pars (orationis)*, meaning part (of speech).*"

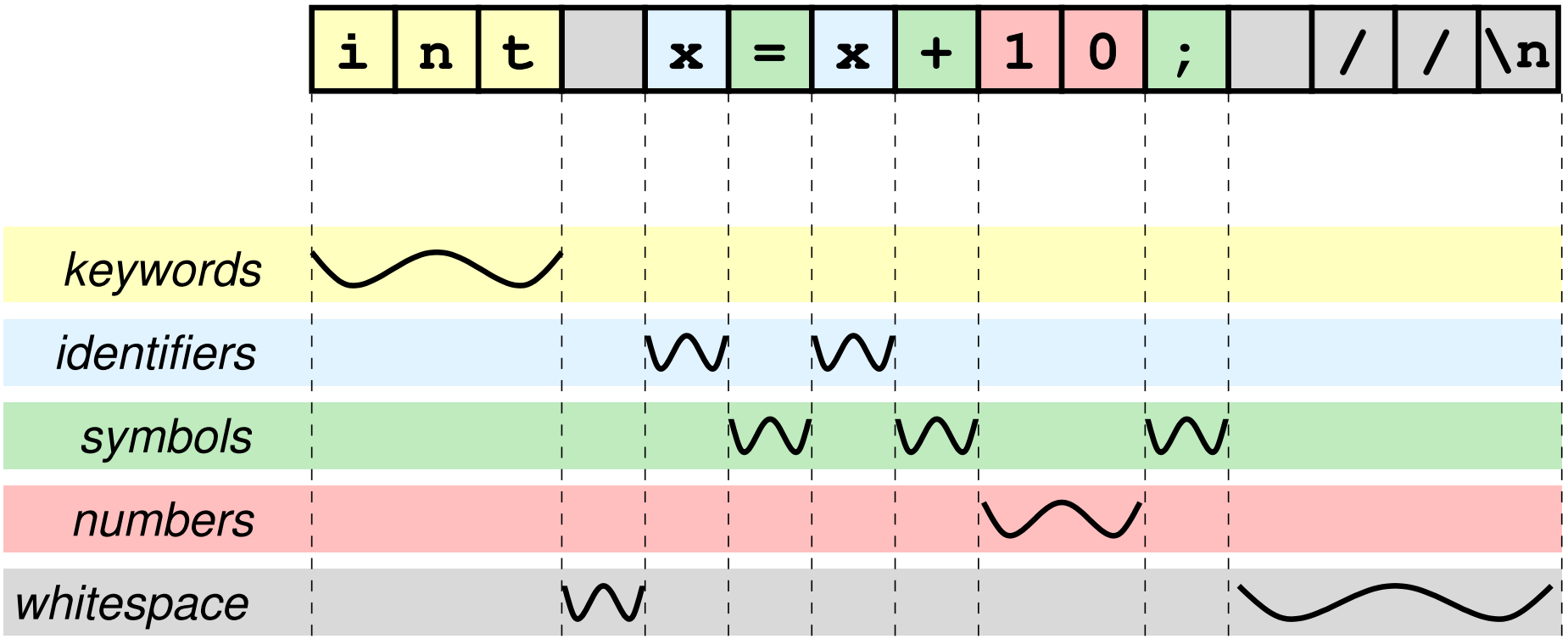
*–Wikipedia*

# Parsing Overview




- **Lexer.** Turns *characters* into *tokens*.
- **Parser.** Turns *tokens* into *Abstract Syntax Tree (AST)*.

# Lexing

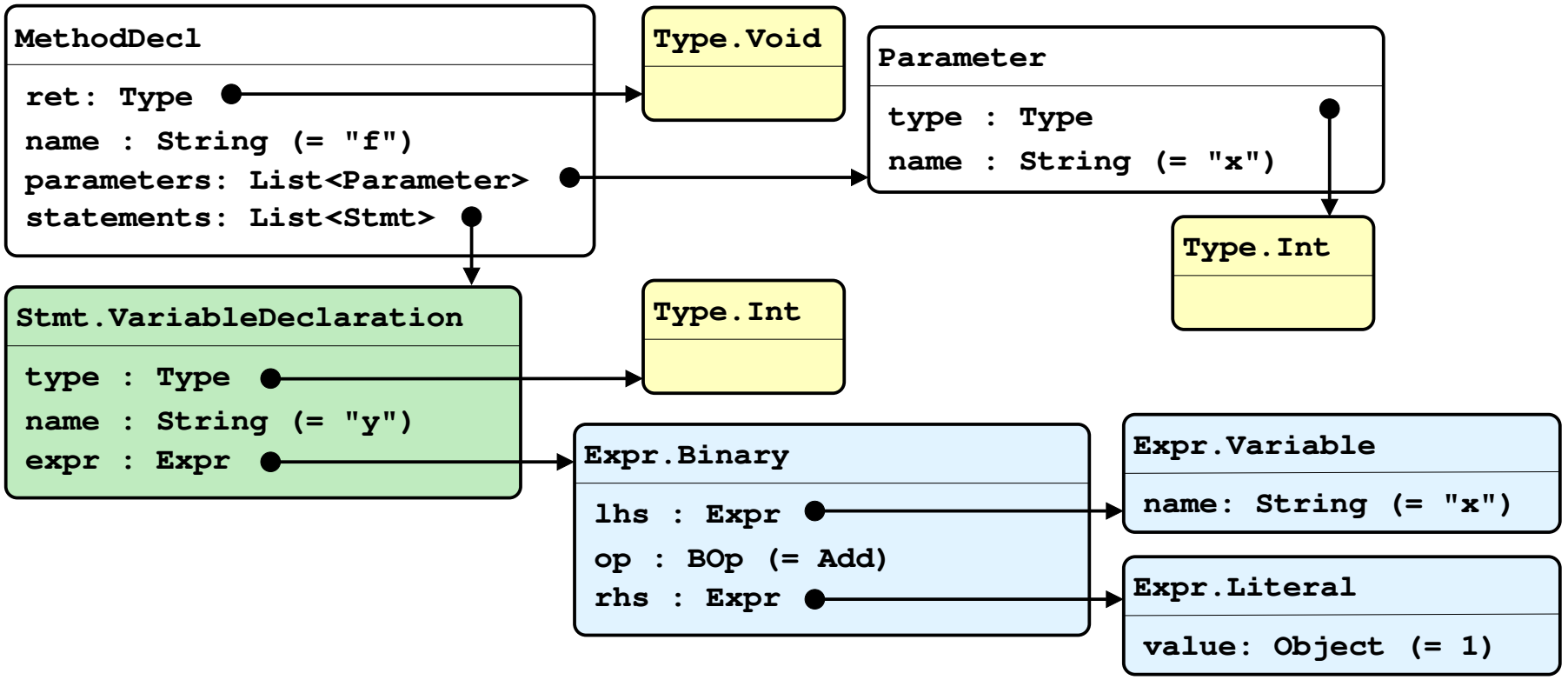



- Characters grouped into **tokens** of related kind.
- **Whitespace** is typically ignored.

# Lexing (Continued)

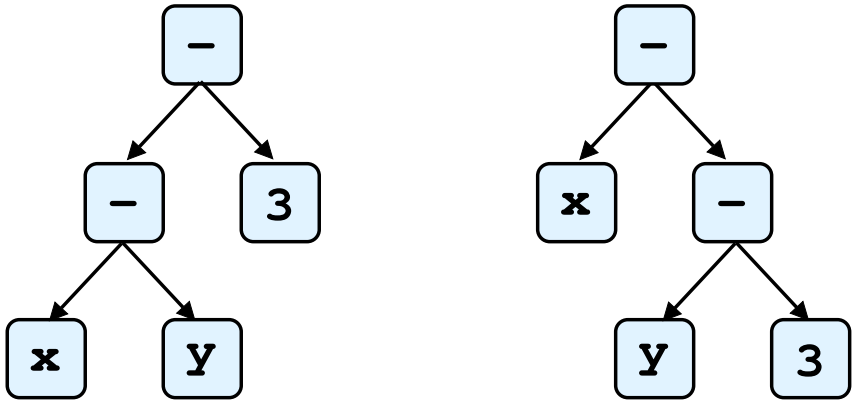
- **Keywords:** `assert` `bool` `break` `case` `continue` `default` `do` `else` `false` `for` `if` `int` `null` `return` `switch` `type` `true` `void` `while`
- **Symbols:** `.` `,` `:` `;` `|` `(` `)` `{` `}` `[` `]` `+` `-` `*` `/` `%` `<` `<=` `>=` `>` `==` `!=` `&&` `||` `!`
- **Identifiers:** $[\texttt{a}-\texttt{zA}-\texttt{Z\$\_}][\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9\$\_}]^*$
- **Numbers:** $[\texttt{0}-\texttt{9}]^+$

# Abstract Syntax Tree (AST)




```
void f(int x) { int y = x + 1; }
```

# Parse Trees

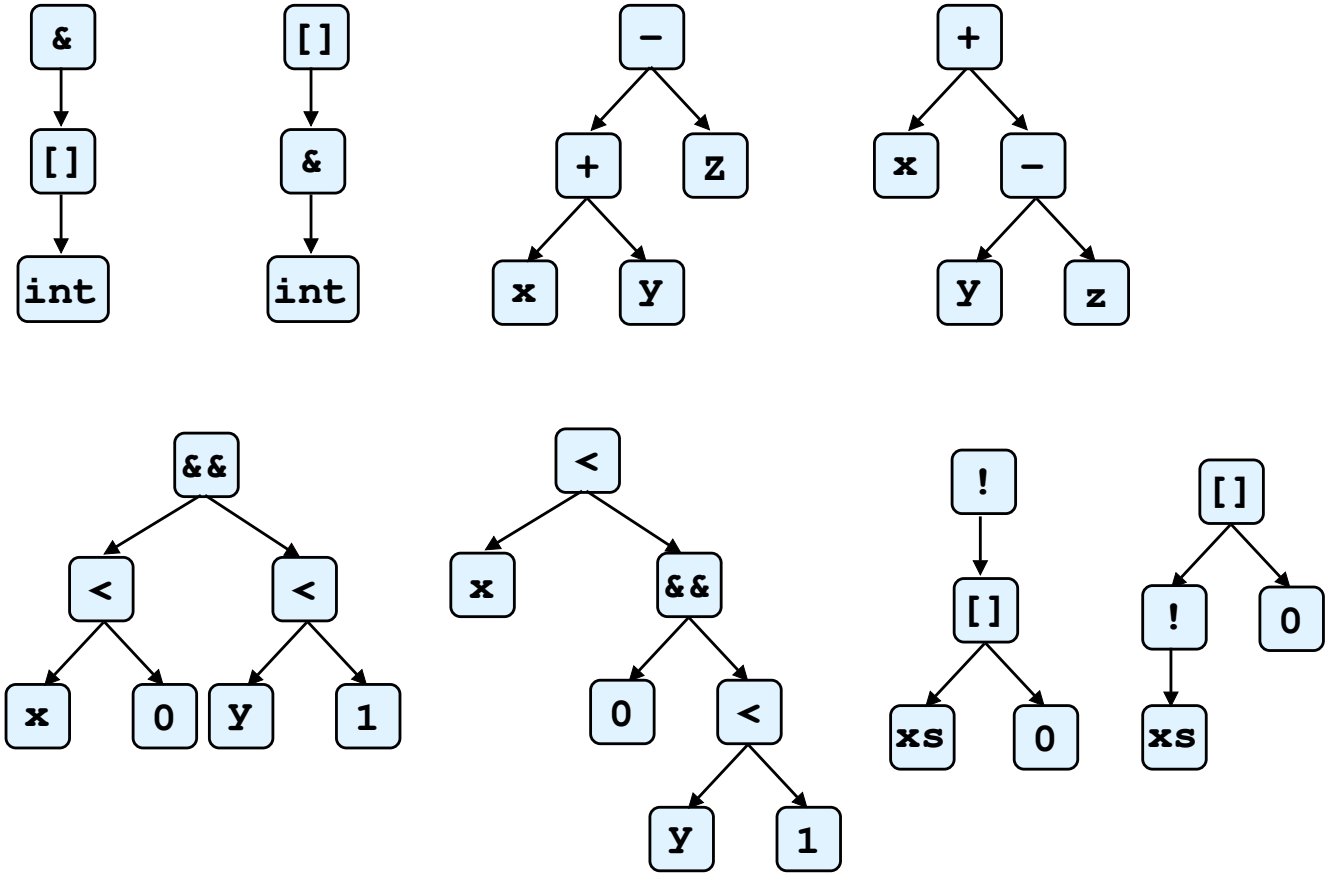



*(left)* *(right)*

```
int z = x - y - 3;
```

- **Left Parse Tree:** corresponds with `(x - y) - 3`
- **Right Parse Tree:** corresponds with `x - (y - 3)`
- *Which way around should it be?*

# More Parse Trees!




- **Problematic Types.** `&int[]`
- **Problematic Exprs.** `x + y - z` `x < 0 && y < 1` `!xs[0]`

# Recursive Decent Parsing

*"In computer science, a* **recursive descent parser** *is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure implements one of the nonterminals of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes."*

*–Wikipedia*

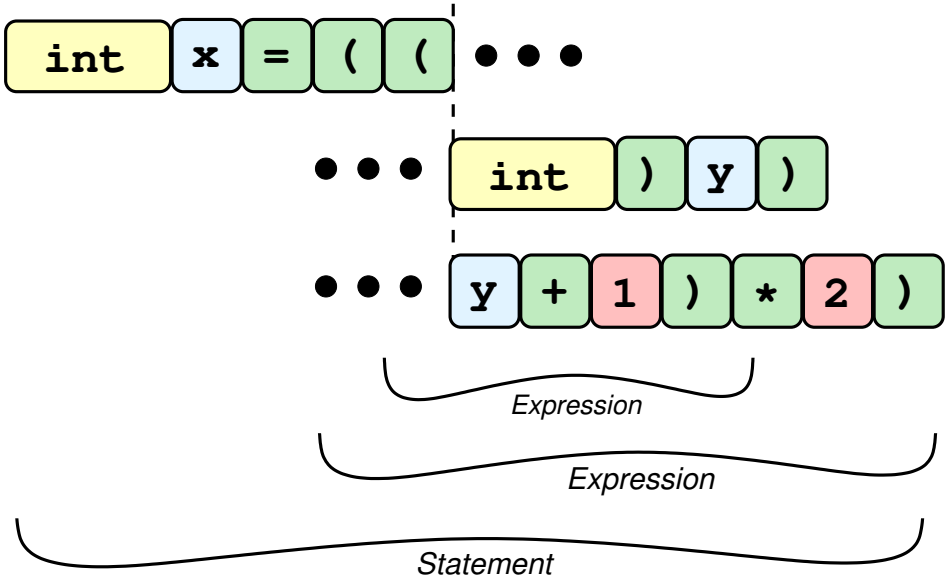
# WhileLang Parser

```
private Expr parseTerm(Context context) {
    checkNotEof();

    int start = index;
    Token lookahead = tokens.get(index);

    if (lookahead instanceof LeftBrace) {
        match("(");
        // First, attempt to parse cast expression
        try {
            Type t = parseType();
            match(")");
            Expr e = parseExpr(context);
            return new Expr.Cast(t,e);
        } catch(Exception ex) {
            // Now, attempt to parse brace
            index = start;
            match("(");
            Expr e = parseExpr(context);
            checkNotEof();
            match(")");
            return e;
        }
```

- **(lexer)** `whilelang.compiler.Lexer`
- **(parser)** `whilelang.compiler.Parser`

# Disambiguation

**Lookahead** *is the number of tokens in front of the current position that the parser must look in order to determine what kind of construct follows.*




- Parser normally looks ahead **at least one** token.
- *How much lookahead required for* WHILE*?*

# **Syntax Errors** versus **Semantic Errors**

**Syntax Error**

Input cannot be parsed into an AST.

**Semantic Error**

Input can be parsed into an AST, but fails other requirements.

- **Incorrect** input files may still parse correctly

```
int f(bool x) {
  return x + 1;
}
```

- Above **can be** parsed into Abstract Syntax Tree
- But, errors **will occur** later during compilation

# Parser Context

```
int add(int x, int x) { return x + 1; }
```

```
int broken(int x) { break; }
```

```
type rec is { int f, bool f }
```

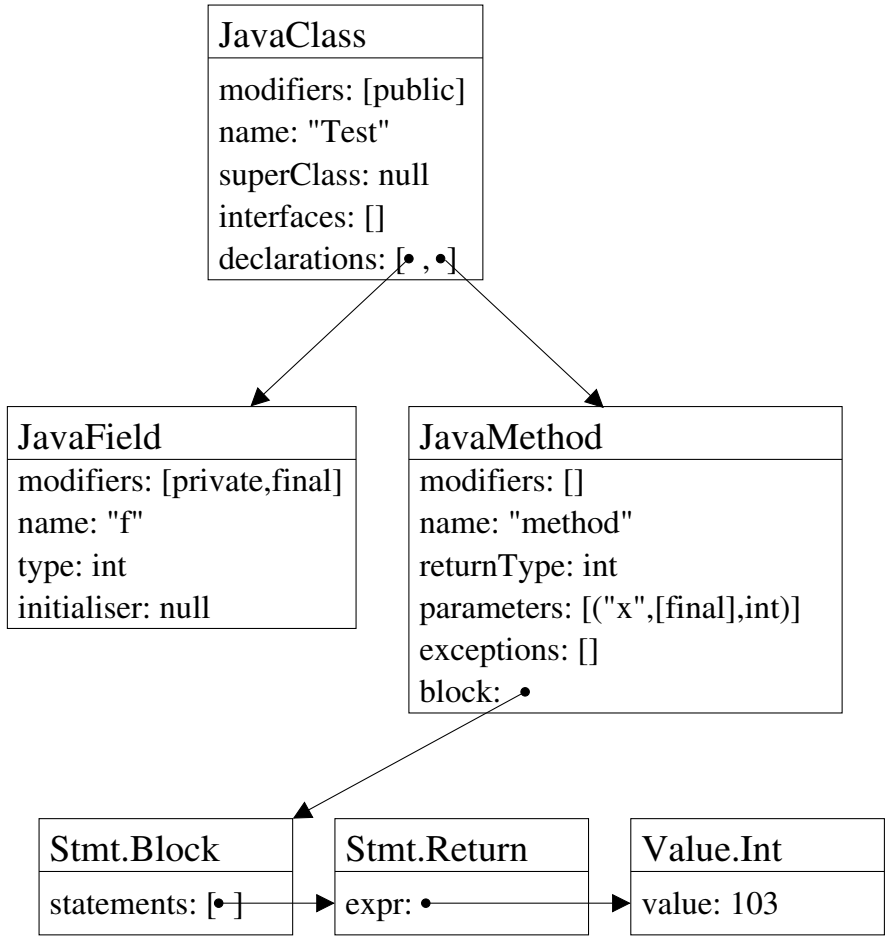
```
void f(int x) { switch x { case 0: case 0: } }
```

- By maintaining **context**, WHILE parser can eliminate more than just syntax errors.

# Parsing Java

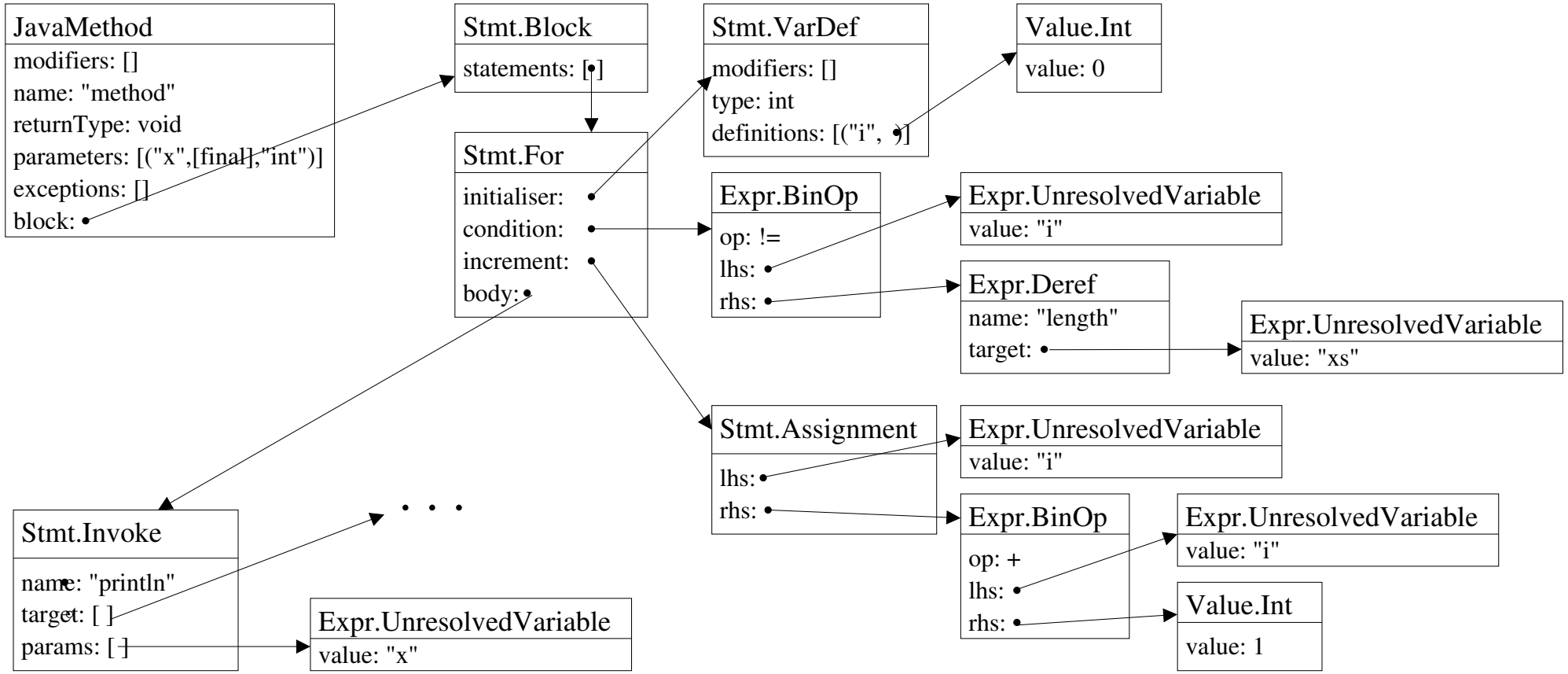
# **Example:** Java (Abstract Syntax Tree)

```
public class Test {
 private final int f;

 int method(int final x) {
  return 103;
}}
```



# **Example:** Java (Abstract Syntax Tree)

```
void method(int[] xs) {
 for(int i=0;i!=xs.length;i=i+1) System.out.println(x);
}
```



# **Example:** Java (Variables)

```
class Test {
 int x;

 class Inner { int f() { return x; } }

 int f(int r) {
  { int y; r = r + y; }
  { int z; }
  r = r + z;
  return r;
}}
```

Example:

- Access to `x` should resolve to a non-local field access
- Access to `y` should resolve to a local variable
- Access to `z` should cause syntax error

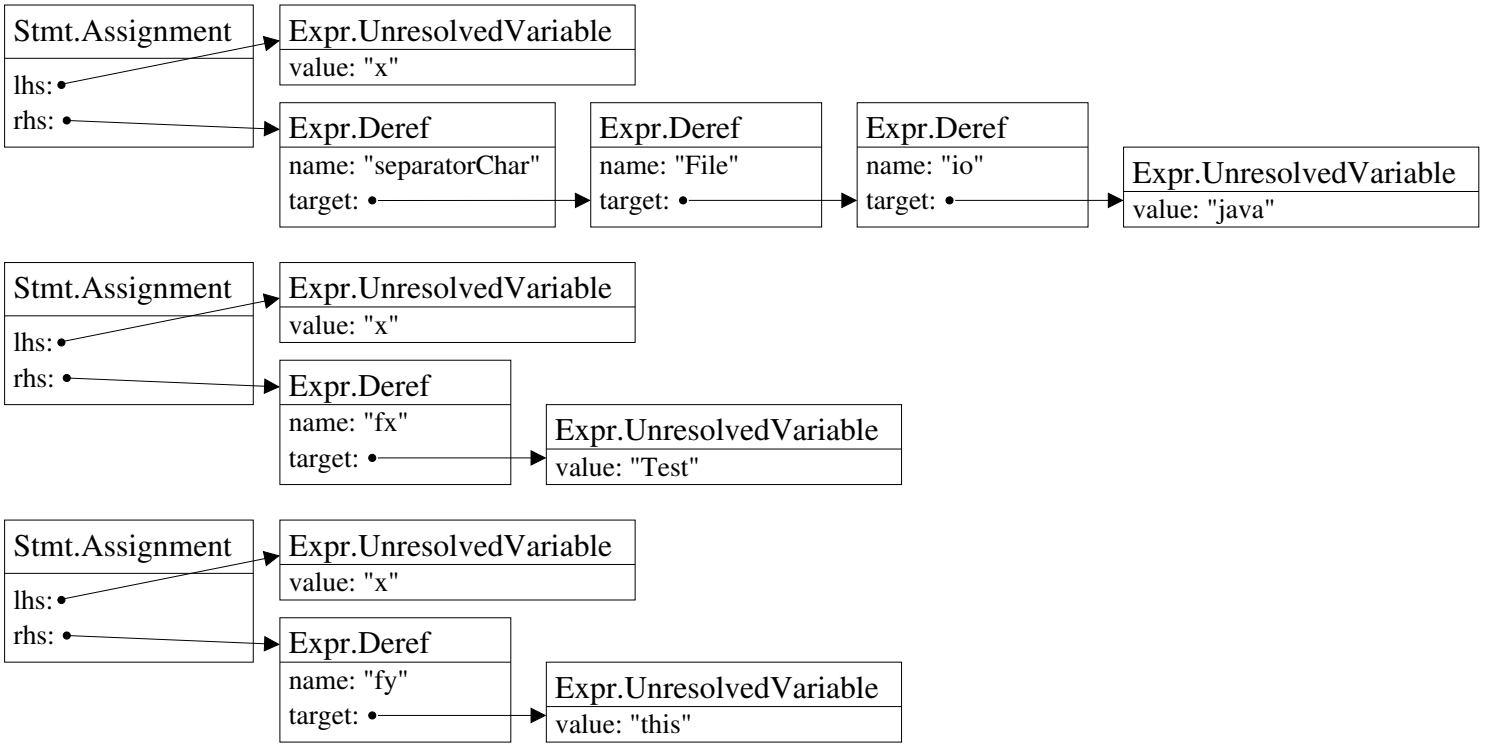
Unresolved Variables

- Are placeholders for variable accesses
- Are converted into field, local or non-local accesses later

# **Example:** Java (Qualified Access)

```
class Test {
 static int fx; int fy;
 void f(int x) { x=java.io.File.separatorChar; x=Test.fx; x=this.fy; }
}
```

- Initial AST for three statements is:




- What AST do we want?