INNOPOLIS UNIVERSITY

Javdin Parser

Course: [F25] Compiler Construction
Date: November 6, 2025
Team: 806

Team Information

Team 806

- **Timofey Ivlev**
- **George Selivanov**

Project: Javdin

Java **d**ynamic **in**terpreter - A dynamic language interpreter with a Bison-based parser

Technology Stack

Core Technologies

- **Source Language**: Project D (academic dynamic language)
- **Implementation Language**: Java 17
- **Parser Development Tool**: **CUP (Construction of Useful Parsers)**
  - Bison-based parser generator for Java
  - Generates LR parsers from grammar specifications
  - Website: https://www2.cs.tum.edu/projects/cup/

Build & Testing Tools

- **Build System**: Maven 3.6+
- **Testing Framework**: JUnit 5
- **Assertion Library**: AssertJ
- **Code Coverage**: JaCoCo

Target Platform

- **JVM** (Java Virtual Machine)
- Runs on any platform with Java 17+

Semantic Analysis Implementation

We implemented two main components:

1. **SemanticAnalyzer**
    - Performs non-modifying semantic validation
    - Detects 4 types of semantic errors:
        1. Return outside function check
        2. Break/Continue outside loop check
        3. Undeclared variable check
        4. Duplicate declaration check
2. **Optimizer**
    - Performs AST-modifying optimizations
    - Uses symbol table for scope management
    - Implements 4 optimization techniques:
    - Pass 1: Collect used variables
    - Pass 2: Apply optimizations
        1. Constant folding
        2. Unused variable removal
        3. Dead branch elimination
        4. Unreachable code removal

Semantic Checks (4 Types)

1. Return Outside Function

Rule: Return statements can only appear inside function bodies.

Error Example (`docs/demos/semantic-check-1-return-outside-function-error.d`):

```
// Example 1.1: Return Outside Function (ERROR)
// This code demonstrates the semantic check for return statements outside
functions

var x := 10
print "Starting program"

return x  // ERROR: Return statement outside function

print "This line is unreachable"
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-1-return-outside-function-error.d"
-q
```

Or use demo script:

```
./demo-semantic-analysis.sh 1
```

Fixed Version (`docs/demos/semantic-check-1-return-outside-function-fixed.d`):

```
// Example 1.1: Return Outside Function (FIXED)
// This shows the corrected version with return inside a function

var x := 10
print "Starting program"

var double := func(n) is
    return n * 2  // CORRECT: Return statement inside function
end

print double(x)
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-1-return-outside-function-fixed.d"
-q
```

Or use demo script:

```
./demo-semantic-analysis.sh 2
```

---

2. Break Outside Loop

Rule: Break statements can only appear inside loop bodies (while, for).

Error Example (`docs/demos/semantic-check-2-break-outside-loop-error.d`):

```
// Example 1.2: Break Outside Loop (ERROR)
// This code demonstrates the semantic check for break statements outside
```

```
loops

var count := 0
print "Starting..."

if count = 0 then
    exit  // ERROR: Break statement outside loop
end

print "Done"
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-2-break-outside-loop-error.d" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 3
```

Fixed Version (`docs/demos/semantic-check-2-break-outside-loop-fixed.d`):

```
v// Example 1.2: Break Outside Loop (FIXED)
// This shows the corrected version with break inside a loop

var count := 0
print "Starting..."

loop
    print count
    count := count + 1

    if count = 5 then
        exit  // CORRECT: Break statement inside loop
    end
end

print "Done"
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-2-break-outside-loop-fixed.d" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 4
```

---

3. Undeclared Variables

Rule: Variables must be declared before use.

Error Example (docs/demos/semantic-check-3-undeclared-variable-error.d):

```
// Example 1.3: Undeclared Variable (ERROR)
// This code demonstrates the semantic check for using variables before
declaration

print "Starting program"

var x := y + 10  // ERROR: Variable 'y' is not declared

print x
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-3-undeclared-variable-error.d" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 5
```

Fixed Version (docs/demos/semantic-check-3-undeclared-variable-fixed.d):

```
// Example 1.3: Undeclared Variable (FIXED)
// This shows the corrected version with proper variable declaration

print "Starting program"

var y := 5       // CORRECT: Variable declared before use
var x := y + 10

print x
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-3-undeclared-variable-fixed.d" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 6
```

---

4. Duplicate Declarations

Rule: Variables cannot be declared twice in the same scope.

Error Example (`docs/demos/semantic-check-4-duplicate-declaration-error.d`):

```
// Example 1.4: Duplicate Declaration (ERROR)
// This code demonstrates the semantic check for duplicate variable
declarations

var x := 10
print x

var x := 20  // ERROR: Variable 'x' is already declared

print x
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-4-duplicate-declaration-error.d" -
q
```

Or use demo script:

```
./demo-semantic-analysis.sh 7
```

Fixed Version (`docs/demos/semantic-check-4-duplicate-declaration-fixed.d`):

```
// Example 1.4: Duplicate Declaration (FIXED)
// This shows the corrected version using different variable names

var x := 10
print x

var y := 20  // CORRECT: Different variable name

print y
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/semantic-check-4-duplicate-declaration-fixed.d" -
q
```

Or use demo script:

```
./demo-semantic-analysis.sh 8
```

Optimizations (4 Types)

1. Constant Folding

Goal: Evaluate constant expressions at compile time instead of runtime.

Example (`docs/demos/optimization-1-constant-folding.d`):

```
// Example 2.1: Constant Folding
// This demonstrates constant expression simplification

var result1 := 2 + 3
var result2 := 10 * 5
var result3 := 100 / 4
var result4 := 20 - 7

var flag1 := true and false
var flag2 := true or false

var comparison1 := 5 < 10
var comparison2 := 15 = 15

print result1
print result2
print result3
print result4
print flag1
print flag2
print comparison1
print comparison2
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/optimization-1-constant-folding.d --optimize" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 9
```

View AST Changes:

```
diff ast-before-optimization.xml ast-after-optimization.xml
```

2. Unused Variable Removal

Goal: Remove variables that are declared but never used, reducing memory footprint.

Example (`docs/demos/optimization-2-unused-variables.d`):

```
// Example 2.2: Unused Variable Removal
// This demonstrates detection and removal of unused variables

var used := 10
var unused1 := 20     // WARNING: Unused variable 'unused1'
var unused2 := 30     // WARNING: Unused variable 'unused2'

print used

var alsoUnused := 40  // WARNING: Unused variable 'alsoUnused'
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/optimization-2-unused-variables.d --optimize" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 10
```

Expected Output: Reports 3 unused variables detected and removed.

View AST Changes:

```
diff ast-before-optimization.xml ast-after-optimization.xml
```

3. Dead Branch Elimination

Goal: Remove unreachable branches in conditionals with constant conditions.

Example (`docs/demos/optimization-3-dead-branch-elimination.d`):

```
// Example 2.3: Dead Branch Elimination
// This demonstrates removal of unreachable branches in conditionals
```

```
var x := 10

if true then
    print "This will always execute"
else
    print "This branch is dead and will be removed"
end

if false then
    print "This branch is dead and will be removed"
else
    print "This will always execute"
end

if 5 < 10 then
    print "Constant comparison - always true"
else
    print "Dead branch"
end
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/optimization-3-dead-branch-elimination.d --
optimize" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 11
```

View AST Changes:

```
diff ast-before-optimization.xml ast-after-optimization.xml
```

---

4. Unreachable Code Removal

Goal: Remove code that can never be executed (after return/break/continue).

Example (`docs/demos/optimization-4-unreachable-code.d`):

```

```

```
// Example 2.4: Unreachable Code Removal
// This demonstrates detection and removal of unreachable code after return

var compute := func(x) is
    if x < 0 then
        return 0
        print "Unreachable after return"  // WARNING: Unreachable code
        var y := 10                        // WARNING: Unreachable code
    end

    return x * 2
end

print compute(5)
print compute(-3)
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/optimization-4-unreachable-code.d --optimize" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 12
```

View AST Changes:

```
diff ast-before-optimization.xml ast-after-optimization.xml
```

---

5. Combined Optimizations

Example (`docs/demos/optimization-5-combined.d`):

```
// Example 2.5: Combined Optimizations
// This demonstrates multiple optimizations working together

var x := 2 + 3              // Constant folding: becomes 5
var unused := 100           // Unused variable removal
var y := x * 2

if true then                // Dead branch elimination
    print y
```

```
else
    print "Dead code"
end

var result := 10 = 10       // Constant folding: becomes true

if result then
    print "Always true"
end

var compute := func(val) is
    if val < 0 then
        return 0
        print "Unreachable"     // Unreachable code removal
        var z := 50             // Unreachable code removal
    end
    return val * 2
end

print compute(5)
```

Run Demo:

```
mvn exec:java -Dexec.mainClass="com.javdin.demo.SemanticAnalysisDemo" \
  -Dexec.args="docs/demos/optimization-5-combined.d --optimize" -q
```

Or use demo script:

```
./demo-semantic-analysis.sh 13
```

Implementation Details

Symbol Table Management

- **Scope**: Tracks variables declared in a specific scope
- **SymbolTable**: Stack of scopes for nested scoping
- Supports function scopes and loop scopes
- Handles scope entry/exit automatically during traversal

Error Handling

- Errors collected during analysis
- Multiple errors reported at once
- Line and column information preserved
- Non-fatal: continues checking after error