

# Language I Compiler → WebAssembly

## Language I Programming Language Compiler

### Compiler Construction Project

**Team:** Hmm

**Member 1:** Mikhail Trifonov → Parser, Analyzer

**Member 2:** Kirill Efimovich → Lexer, CodeGen

# Project Overview

Source Language	Language I (imperative)
Implementation Language	Java
Parser	JavaCC
Target Platform	WebAssembly (WAT)

## Capabilities:

- Full Language I → WebAssembly compilation
- Execution via `wasmtime`
- Comprehensive testing suite (100+)

# Compiler Architecture

**Source Code (.i) → Lexer → Parser → Analyzer → Code Generation → WebAssembly (.wat)**

**Main Components:**

1. **Lexer** - Lexical analyzer
2. **Parser** - Syntax parser (Bison based)
3. **SemanticAnalyzer** - Semantic analysis
4. **CodeGenerator** - WebAssembly generation
5. **Compiler** - Compiler itself

# Task Description

**Goal:** Complete-featured Language I compiler → WebAssembly Text Format (WAT)

**Technology:**

- **Parser:** JavaCC - Bison based parser
- **Code Generation:** Direct translation to WebAssembly
- **Execution:** `wasmtime` runtime

# Language I - Overview

## Characteristics:

- General-purpose imperative language
- Static typing with type inference
- Minimalist syntax

## Program Structure:

```
Program ::= { SimpleDeclaration | RoutineDeclaration }
```

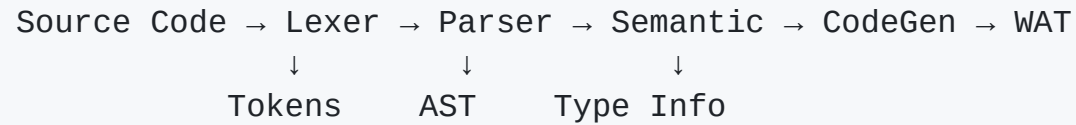
## Three entity categories:

1. Variables
2. Types
3. Routines

# Target Language: WebAssembly (WAT)

- **Format:** WebAssembly Text format
- **Platform:** WebAssembly (WASM) virtual machine
- **Execution:** `wasmtime` runtime
- **Extension:** `.wat`

# Project Architecture



## Components:

- **Lexer:** Tokenization, keyword recognition
- **Parser (JavaCC):** AST construction, syntax validation
- **Semantic Analyzer:** Type checking, scope management, AST modification
- **Code Generator:** Translation to WebAssembly

# Data Types in Language I

## Predefined Types:

Type	Description
<code>integer</code>	Whole numbers
<code>real</code>	Real numbers
<code>boolean</code>	true / false

## Custom Types:

- **Records:** `record { ... } end`
- **Arrays:** `array [size] Type`

# Variables and Declarations

## Syntax:

```
var Identifier : Type [ is Expression ]  
var Identifier is Expression
```

## Examples:

```
var x : integer is 42  
var y is 3.14          // real (type inferred)  
var arr : array [10] integer  
var flag is true       // boolean
```

# Custom Types

## Records:

```
type Point is record
  var x : integer is 0
  var y : integer is 0
end

var p : Point
p.x := 10
```

## Arrays:

```
type Vector is array [3] real
var v : Vector
```

# Expressions and Operators

## Operator Precedence (highest → lowest):

1. Unary: `not`, parentheses `()`
2. Arithmetic: `*`, `/`, `%`
3. Arithmetic: `+`, `-`
4. Comparisons: `<`, `<=`, `>`, `>=`, `=`, `/=`
5. Logical: `and`
6. Logical: `or`, `xor`

# Control Structures (1/2)

## Conditional Statement:

```
if x > 0 then  
  print true  
else  
  print false  
end
```

## While Loop:

```
var i : integer is 0  
while i < 10 loop  
  print i  
  i := i + 1  
end
```

# Control Structures (2/2)

## For with Range:

```
for i in 1..10 loop  
  print i  
end  
  
for j in 10..1 reverse loop  
  print j  
end
```

## For over Array:

```
for element in arr loop  
  print element  
end
```

# Routines (1/2)

## Syntax:

```
routine Identifier ( Parameters ) [ : Type ] is Body end
```

## Example:

```
routine inc(x : integer) : integer => x + 1
```

# Routines (2/2)

## Function:

```
routine add(a : integer, b : integer) : integer is  
  return a + b  
end
```

## Recursion:

```
routine factorial(n : integer) : integer is  
  if n <= 1 then return 1  
  else return n * factorial(n - 1) end  
end
```

# Scope Management

## Rules:

- Scope begins at declaration
- Extends to end of block
- Nested scopes can shadow outer ones

## Example:

```
var global : integer is 10
routine ex() is
  var local : integer is 20
  if true then
    var shadowed : integer is 30
  end
end
```

# Project Structure

```
src/main/java/com/languagei/compiler/  
├── lexer/  
├── parser/  
├── ast/  
├── semantic/  
├── codegen/  
├── Main.java  
└── Compiler.java
```

# Lexical Analyzer (Lexer)

## Tasks:

- Tokenization of source code
- Keyword recognition
- Comment and whitespace handling
- Position tracking for errors

**Keywords:** var, type, routine, if, then, else, while, for, loop, return

# Tokens and Data Structures

## Tokens:

```
public enum TokenType {  
    VAR, ROUTINE, TYPE, RECORD, ARRAY,  
    INTEGER_TYPE, REAL_TYPE, BOOLEAN_TYPE,  
    IF, THEN, ELSE, WHILE, FOR, LOOP,  
    ASSIGN, PLUS, MINUS, MUL, DIV, MOD,  
    LT, LE, GT, GE, EQ, NE, AND, OR, XOR, NOT  
}
```

# Parser Technology - JavaCC

## **JavaCC (Java Compiler Compiler):**

- Automatic parser generation
- Built-in AST support

# Parser (1/2)

**Technology:** JavaCC generates parser and AST nodes

**Grammar (Part 1):**

```
Program ::= { SimpleDeclaration | RoutineDeclaration }  
SimpleDeclaration ::= VariableDeclaration | TypeDeclaration  
VariableDeclaration ::= "var" Identifier ":" Type ["is" Expression]  
                    | "var" Identifier "is" Expression  
TypeDeclaration ::= "type" Identifier "is" Type
```

# Parser (2/2)

## Grammar (Part 2):

```
Type ::= "integer" | "real" | "boolean"  
      | ArrayType | RecordType | Identifier  
ArrayType ::= "array" "[" [Expression] "]" Type  
RecordType ::= "record" {VariableDeclaration} "end"
```

# Building AST

## Input Code

```
var a is 4 + 2 * 3 - 7
var b is a + 2
var cond : boolean is false and not true
var complex is 20 - (2 + 5) = 7 + 2 * 3

for i in 1..5 loop
  print i
end

print a
print b
```

## AST Output

```
Program
  VarDecl(a)
    Literal(3)
  VarDecl(b)
    Binary(PLUS)
      Identifier(a)
      Literal(2)
  VarDecl(cond)
    PrimitiveTypeNode
    Literal(false)
  VarDecl(complex)
    Literal(true)
  ForLoopNode
    Literal(1)
    Literal(5)
    BlockNode
      Print
        Identifier(i)
      Print
        Identifier(a)
      Print
        Identifier(b)
```

# Semantic Analysis (1/2)

## Tasks:

1. Expression type checking
2. Scope validation
3. Declaration correctness checking
4. Information collection for code generation
5. Constant folding
6. Removing false-conditioned blocks

# Semantic Analysis (2/2)

## Validation Types:

- Type compatibility in assignments
- Type compatibility in function calls
- Operator correctness for types
- Name resolution (symbol table)

# Code Optimization Example

## Original Code

```
var a : integer is 28

if a > 20 then
  print a
end
```

## Dead Code Elimination

```
var a : integer is 28

if false then
  print a
end
```

## AST

```
Program
  VarDecl(a)
    PrimitiveTypeNode
    Literal(28)
  If
    Binary(GT)
      Identifier(a)
      Literal(20)
    BlockNode
      Print
        Identifier(a)
```

## Optimized AST

```
Program
  VarDecl(a)
    PrimitiveTypeNode
    Literal(28)
```

# Constant Folding

## Input Code

```
var a is 4 + 2 * 3 - 7
var b is a + 2
var cond : boolean is false and not true
var complex is 20 - (2 + 5) = 7 + 2 * 3

for i in 1..5 loop
  print i
end

print a
print b
```

## AST Output

```
Program
  VarDecl(a)
    Literal(3)
  VarDecl(b)
    Binary(PLUS)
      Identifier(a)
      Literal(2)
  VarDecl(cond)
    PrimitiveTypeNode
    Literal(false)
  VarDecl(complex)
    Literal(true)
  ForLoopNode
    Literal(1)
    Literal(5)
    BlockNode
      Print
        Identifier(i)
  Print
    Identifier(a)
  Print
    Identifier(b)
```

# Semantic Checks Example

## Invalid Code

```
var a : boolean is true + 47

while 88 loop

end
```

## Compiler Errors

Compilation failed due to semantic errors:

ERROR: Invalid operand types for  
arithmetic operation at TEST.i:5:25

ERROR: Type mismatch: cannot assign  
void to boolean at TEST.i:5:1

ERROR: While condition must be  
boolean at TEST.i:6:1

# WebAssembly Code Generation (1/3)

## Code Generator Architecture:

```
CodeGenerator
├─ WebAssembly module
├─ WASI imports (fd_write, proc_exit)
├─ Global variables
├─ Functions (user-defined + built-in)
└─ Linear memory
```

# WebAssembly Code Generation (2/3)

## WASI Imports:

```
(import "wasi_snapshot_preview1" "fd_write"  
  (func $fd_write (param i32 i32 i32 i32) (result i32)))  
(import "wasi_snapshot_preview1" "proc_exit"  
  (func $proc_exit (param i32)))
```

# WebAssembly Code Generation (3/3)

## Main Functions:

1. `visitProgram()` - module generation
2. `visitRoutineDeclaration()` - function generation
3. `visitStatement()` - instruction generation
4. `visitExpression()` - expression generation

# WASM Standard Library

## Built-in Output Functions:

```
(func $print_int (param $val i32)
  (local.get $val)
  (call $int_to_string)
  (call $write_string))

(func $print_real (param $val f64) ...)
(func $print_bool (param $val i32) ...)
```

# WASM Memory Management

## WebAssembly Linear Memory:

```
(memory (export "memory") 1)
(global $heap_ptr (mut i32) (i32.const 0))
```

## Memory Allocation:

```
(func $allocate_array (param $size i32) (param $elem_size i32) (result i32)
  (local $total_bytes i32)
  (local.set $total_bytes
    (i32.mul (local.get $size) (local.get $elem_size)))
  ...)
```

# Code Generation Examples (1/2)

## Simple Program:

```
var x : integer is 42  
print x
```

## WebAssembly (simplified):

```
(func $_start  
  (local $x i32)  
  (i32.const 42)  
  (local.set $x)  
  (local.get $x)  
  (call $print_int))
```

# Code Generation Examples (2/2)

## Arithmetic:

```
var result : integer is (a + b) * 2
```

## WebAssembly (simplified):

```
(local.get $a)  
(local.get $b)  
(i32.add)  
(i32.const 2)  
(i32.mul)  
(local.set $result)
```

# How to run

## Commands:

```
# Compilation
java -jar target/compiler.jar compile test.i -o out.wat

# AST
java -jar target/compiler-i-1.0.0.jar ast test.i

# Compilation and Execution
java -jar target/compiler.jar run test.i -o out.wat
```

# Done

- Complete Language I → WebAssembly cycle
- Semantic analysis with type checking
- AST modification
- Optimized WAT code
- Memory management (heap allocation)
- Error handling with positions
- 100+ integration tests
- `wasmtime` integration

# Challenges and Solutions (1/2)

## 1. Parser Technology - Initial Attempt:

- Problem: Initially used Bison-based parser with GNU tools
- Issue: JNI bridge between C++ and Java caused integration problems
- Solution: Switched to JavaCC (Java Compiler Compiler) for seamless Java integration

## 2. WebAssembly Output:

- Problem: WASM generated but produced no output
- Issue: Missing WASI (WebAssembly System Interface) support
- Solution: Implemented WASI imports (fd\_write, proc\_exit) for I/O and process control

# Challenges and Solutions (2/2)

## 3. Lexer Input Recognition:

- Problem: Malformed input like "1.2.3" (multiple dots) caused parsing errors
- Solution: Enhanced lexer validation to properly distinguish between decimal numbers and invalid formats

## 4. Operator Precedence:

- Problem: Incorrect operator precedence in expression evaluation (e.g.,  $2 + 3 * 4$  was evaluated as  $(2 + 3) * 4 = 20$  instead of  $2 + (3 * 4) = 14$ )
- Solution: Fixed parser priority levels

# Honesty

- **Sorting Algorithm Bugs** - Bubble sort and similar algorithms have known issues; unknown how to fix
- **String and Char Types** - Not supported (not specified in Language I specification, decided out of scope)
- **WebAssembly Memory Optimization** - Only basic memory management implemented, no advanced WASM optimization

# Code Examples - Factorial

## Language I:

```
routine factorial(n : integer) : integer is
  if n <= 1 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
```

**Result:** factorial(5) = 120

# Code Examples - Arrays

## Language I:

```
var arr : array [5] integer
for i in 1..5 loop
  arr[i] := i * 2
end
for element in arr loop
  print element
end
```

**Result:** 2 4 6 8 10

# Code Examples - Records

```
type JobDetails is record
  var level : integer is 0
  var hours : integer is 0
end

type Job is record
  var salary : integer is 0
  var details : JobDetails
end

type Person is record
  var age : integer is 0
  var job : Job
end

var p : Person
p.age := 30
p.job.salary := 1000
p.job.details.level := 2
p.job.details.hours := 40

print p.age
print p.job.salary
print p.job.details.level
print p.job.details.hours
```

**Thank you for your attention!** 🎉

**GitHub:** <https://github.com/LuminiteTime/Compilers-Construction-Hmm>

---

**Questions?**