Simple Java: Project Documentation

Introduction

The motivation behind developing this compiler project came from my experience at school using the johnny software for simulating assembly programs. While learning with johnny, I realized that it didn’t help me integrate my high-level programming knowledge—specifically, I never truly understood how a language like Java is transformed into machine code. I envisioned a translator that could convert Java programs (as taught in class) into johnny assembly code, enabling students to connect their learning from high-level programming to low-level machine operations. Consequently, I began studying compiler fundamentals and implemented a compiler that translates Java code into a format compatible with johnny. Note that due to Java's complexity and the limitations of johnny assembly, this compiler emphasizes an understanding of compilation principles rather than being a tool for full-fledged object-oriented projects.

1 Compiler Features

This compiler is implemented in Java and is designed to compile simple Java programs. Its functionality is outlined as follows:

1.1 Overall

1.1.1 Supported Features

- **Basic Variable Declaration:**
  Supports declaration of int-type variables.
- **Declaration with Assignment:**
  Allows variables to be declared and assigned simultaneously.
- **Assignment Statements:**
  Handles standard assignment operations.
- **Conditional Statements:**
  Supports if statements.
- **Looping Constructs:**
  Supports while loops.
- **Function Definitions and Calls:**
  Functions are supported via an extended instruction set.
- **Expression Evaluation:**
  Evaluates expressions with proper operator precedence.
- **Output:**
  Supported through the extended instruction set.

1.1.2 Unsupported Features

- Numbers must not exceed 999.
- Negative numbers and decimals are not supported.
- Variables must have unique names (even across main and function scopes).
- Binary operations other than addition and subtraction (e.g., multiplication and division) are not supported (though functions can simulate these).
- Classes and objects are not supported.
- Data types such as string, char, float, double, and boolean are not supported.
- Multiple variable declarations on one line are not supported.
- Function calls within functions (including recursion) are not supported.
- Complex data structures, multi-file projects, and advanced syntax checking are not supported.

1.2 Roadmap Overview

	Lexer	Preprocessor	Parser	Generator	Assembly To Johnny
Var Declaration	✅	✅	✅	✅	✅
Var Computation	✅	✅	✅	✅	✅
Computation Priority	✅	✅	✅	✅	✅
Domain	✅	✅	✅	✅	✅
Input	✅	✅	❌	❌	✅
Output	✅	✅	✅	✅	✅
IF Statement	✅	✅	✅	✅	✅
While Loop	✅	✅	✅	✅	✅
For Loop	✅	✅	✅	✅	✅
Function	✅	✅	✅	✅	✅
Boolean	❌	❌	❌	❌	❌
Variable Name Domain	✅	❌	❌	❌	❌

1.3 Extended Instruction Set

Two additional instructions extend the base instruction set:

- **savi addr:**
  Enhances the original `save` instruction by saving a register's value into the memory address pointed to by `addr`.
- **jmpi addr:**
  Enhances the original `jmp` instruction by jumping to the memory address pointed to by `addr`.

1.4 Compilation Log

The compiler generates detailed logs during the compilation process, primarily used for debugging. These logs include:

- Data from each compilation stage (e.g., number of lines processed, variables defined, memory allocation details).
- Error messages that explain the reasons for compilation failures (without specific line numbers).

1.5 SimpleJava Subset Design

1.5.1 Overall Structure

- **Single Class per Program:**
  Each program must be written within a single class and include a `main` method as the entry point with the fixed signature:

```
public static void main() {
    // ...
}
```

- **Static Methods:**
  In addition to `main`, additional static methods can be defined:
  - They must return an int.
  - They can only take int parameters.
  - Method overloading is not supported.

```
public static int multiply(int a, int b) {
    int result = 0;
    while (b > 0) {
        result = result + a;
        b = b - 1;
    }
    return result;
}
```

- **Class Members:**
  While class members are allowed, the language does not support object-oriented features such as objects, member variables, inheritance, or polymorphism.
- **Exception Handling:**
  No try/catch/finally statements are supported.
- **Recursion:**
  Recursive function calls are not supported.

1.5.2 Data Types

- **Only Integer Type:**
  The only supported data type is `int`.

- **Booleans:**
  Booleans are represented by integers: 0 means false, and any nonzero value means true.
- **Unsupported Types:**
  There is no support for `String`, arrays, floating-point numbers, objects, or other data types.

1.5.3 Expressions and Operators

- **Arithmetic Operators:**
  Supported operators are `+` and `-`.
- **Comparison Operators:**
  Supported operators are `==`, `!=`, `<`, `>`, `<=`, and `>=`. These are used for conditional expressions, with results stored as integers (0 or 1).
- **Parentheses:**
  Parentheses can be used to control precedence.
- **Function Calls:**
  Static methods can be called within expressions. All parameters must be of type `int`, and the return type must be `int`.

1.5.4 Statements

The language supports the following statement types:

- **Variable Declarations:**
  Only int variables can be declared, optionally with an initializer:

```
int a;
int b = 10;
```

- **Assignment Statements:**
  For example:

```
a = 5;
a = a + 3;
```

- **Conditional Statements (Developing):**
  Supports if-else blocks:

```
if (a < 10) {
    a = a + 1;
} else {
    a = a - 1;
}
```

- **Loop Statements (Developing):**
  Only while loops are supported:

```
while (a != 0) {
    a = a - 1;
}
```

- **Function Calls (Developing):**
  Static methods can be called:

```
foo(3, a);
```

- **Return Statements (Developing):**
  In static methods (other than main), use return to provide an integer result:

```
return a + b;
```

1.5.5 Example

```
class SimpleProgram {
  public int mul(int a, int b) {
    int result = 0;
    while (b > 0) {
      result += a;
      b--;
    }
    return result;
  }
  public static void main(String[] args) {
    int x = 10;

    for(int i=0;i<x;i++){
      System.out.println(mul(i,2));
    }
  }
}
```

2 How to Use

This section will include detailed instructions accompanied by images and screenshots:

- **Code Generation:**
  Step-by-step instructions on how to compile a Java program.
- **UI Interface:**
  A guide to navigating the compiler's graphical interface.
- **RAM Data Editor:**
  Instructions for using the RAM data editor.
- **Importing to Johnny:**
  Steps to import the compiled assembly code into the johnny software.

Additionally, several sample programs are provided:

- An example demonstrating function usage.
- An example showcasing a more complex algorithm.
- An example program that triggers an error to illustrate unsupported features.

3 Compiler Internals

The compiler transforms the input program through multiple stages. Each stage converts the program into a lower-level representation until machine code, which johnny can recognize, is produced.

3.1 Compilation Process Overview

The input program is processed through a series of converters that progressively break it down into simpler forms:

3.2 Lexer

The Lexer tokenizes the input program into a sequence of tokens (keywords, operators, identifiers, etc.) and discards irrelevant elements like whitespace and newline characters.
Brief Description: The Lexer transforms source code into atomic tokens, forming the basis for subsequent analysis.
(A sample of Lexer output will be inserted later.)

3.3 Preprocessor

The Preprocessor cleans up the token stream. Since the compiler does not support class definitions, it removes class constructs and identifies the entry point (the main function) in the program.

3.4 Parser

The Parser constructs an Abstract Syntax Tree (AST) from the token stream.
AST Explanation: An Abstract Syntax Tree is a hierarchical representation of the program structure, capturing syntactic relationships among code elements and facilitating further analysis and code generation.
For functions, a second parsing pass is performed to locate and process definitions—this specialized procedure ensures that function-related constructs are correctly incorporated. The parsing process is recursive, handling nested structures naturally.
(An inheritance diagram of AST nodes and a sample of Parser output will be inserted later.)

3.5 Generator

The Generator traverses the AST to produce assembly code. Key aspects include:

- **Label Usage:**
  Labels are used instead of direct memory addresses to improve readability and debugging. They are later resolved into actual addresses.
- **Recursive Generation:**
  Each AST node implements a `generate()` method that recursively produces code.
- **Constant Management:**
  Constants encountered during parsing are recorded and eventually stored in a dedicated memory area.
- **Function Generation:**
  - Function definitions are generated and appended after the main function.
  - When calling a function, the current line number is stored in a fixed register before jumping into the function.
  - The function returns a value by storing it in a designated register and using `jmpi` to return control.
  - Function parameters are stored in a special memory area.
- **Temporary Variables:**
  Temporary variables used during expression evaluation are allocated from a dedicated area. Their addresses can be reused once they are no longer needed.

3.6 AsmToJohnny Converter

After generating assembly code, the AsmToJohnny Converter translates it into machine code recognized by johnny:

- **Label Resolution:**
  The converter scans the assembly code multiple times to resolve all labels accurately.
- **Special Label Handling:**
  - For lines containing only a label (i.e., empty lines), the label is moved to the end of the following line to ensure correct line numbering.
  - Special labels, such as those defined as “current line plus three” (`THIS_LINE_PLUS_THREE`) or those representing fixed registers, are processed with custom logic.

4 Memory Management

The compiler divides memory into specific regions, each serving a distinct purpose. The layout is as follows:

- **0–600: Program Area**
  Contains the compiled program code.
- **600–800: Constants Area**
  Stores numeric constants used during program execution.
- **801–850: Variable Area**
  Holds variables defined in the source code.
- **851–870: Temporary Variable Area**
  Used for storing intermediate results during expression evaluation.
- **871–890: Function Parameter Area**
  Dedicated to storing parameters passed to functions.
- **891–899: Fixed Pointers/Register Area**
  Reserved for special pointers and registers used for function calls and fixed operations.

Special addresses defined in this region include:
    - `OUTPUT_PointerAddress = 895`
    - `PARAMETER_PointerAddress = 896`
    - `RETURN_PointerAddress = 897`
    - `FUNCTION_Result_Address = 898`
    - `ERROR_DefaultAddress = 899`
- **900–949: Input Area**
  Reserved for storing input data.
- **950–999: Output Area**
  Reserved for storing output data.

5 Technical Challenges

Label Processing

Handling labels in the generated assembly involved several challenges:

- **Multiple Scans:**
  The converter performs multiple scans of the assembly code to resolve all labels accurately.
- **Empty Line Labels:**
  If a line contains only a label, it is shifted to the end of the next line to maintain proper line numbering.
- **Special Labels:**
  Labels such as `THIS_LINE_PLUS_THREE` (representing the current line number plus three) and those corresponding to fixed registers require specialized processing.

Function Call Limitations

Implementing a full function call stack in a constrained memory environment is complex. Therefore, the compiler does not support nested or recursive function calls:

- **Simplified Approach:**
  A simple jump mechanism using fixed registers is used instead of a full call stack.

6 Project Statistics

- Total lines of code: 3543
- The percentage of source code lines: 72%
- The percentage of comments: 11%
- The percentage of Black lines: 17%