Interpreter Report

CSCE4101 - Compiler Design (Fall 2024)
Instructor: Dr. Ahmed Rafea

Ahmed Badr - 900202868
Youssef Khaled - 900213467

November 24, 2024

Contents

1 Introduction

This report details the design and implementation of a simple interpreter for a language supporting variable declarations, arithmetic operations, arrays, and control flow constructs. The project demonstrates fundamental concepts of compiler design, such as lexical analysis, syntax parsing, and semantic analysis.

2 Attribute Grammar and Semantic Rules

The interpreter implements an attribute grammar with semantic rules for:

- **Variable Declarations**: Variables and arrays are stored in a symbol table with their attributes (type, size, etc.).
- **Expressions**: Arithmetic and relational expressions are evaluated using semantic actions, ensuring type safety.
- **Control Statements**: Logical conditions are evaluated to control the execution flow of blocks and statements.

3 Symbol Table Structure

The symbol table is implemented as an array of `SymbolEntry` structures. Each entry contains:

- **name**: Identifier name.
- **type**: Data type (`int` or `float`).
- **value**: Union supporting scalar or array values.
- **is_array**: Boolean flag for arrays.
- **array_size**: Size of the array (if applicable).
- **line_number**: Declaration line number.

4 Implementation Details

4.1 Scanner

The scanner (`scanner.l`) uses Flex to tokenize input, recognizing:

- Keywords: `int`, `float`, `if`, `while`, etc.
- Operators: `+`, `-`, `*`, `/`, `<`, `=`, etc.
- Identifiers, numbers, and delimiters.

4.2 Parser and Semantic Analysis

The parser (`interpreter.c`) employs a recursive descent approach. Key components include:

- **Expressions**: Handled by functions such as `additiveExpression()` and `term()`.
- **Statements**: Includes assignment, control flow (`if-else`, `while`), and input/output operations.
- **Semantic Actions**: Ensures type safety, array bounds checking, and variable initialization.

5 Handling Semantic Errors

Semantic errors are detected during parsing or evaluation, and the interpreter terminates with an appropriate message. Examples include:

- Use of undefined variables.
- Type mismatches in expressions.
- Array index out-of-bounds.
- Re-declaration of variables.

6 Implementation Challenges and Decisions

- **Array Support**: Handling dynamic memory allocation for arrays.
- **Error Reporting**: Capturing line and position for meaningful error messages.
- **Type Safety**: Ensuring compatible types during assignments and operations.

7 Running the Interpreter

7.1 Compilation

To compile the interpreter:

```
flex --prefix=scanner scanner.l
gcc lex.scanner.c interpreter.c -o interpreter
```

7.2 Execution

To run the interpreter on a test file:

```
./interpreter <test_file>
```

For example:

```
./interpreter test3.txt
```

8 Sample Run

Below is a sample run with `test3.txt`:

```
TOKEN: PROGRAM at line 1, position 1
TOKEN: ID (test3) at line 1, position 9
TOKEN: LBRACE at line 1, position 15
TOKEN: INT at line 2, position 5
TOKEN: ID (x) at line 2, position 9
TOKEN: SEMICOLON at line 2, position 10
TOKEN: ID (x) at line 3, position 5
TOKEN: ASSIGN at line 3, position 7
TOKEN: NUM (5) at line 3, position 9
TOKEN: SEMICOLON at line 3, position 10
TOKEN: WRITE at line 4, position 5
TOKEN: ID (x) at line 4, position 11
TOKEN: SEMICOLON at line 4, position 12
x = 5
TOKEN: ID (x) at line 5, position 5
TOKEN: ASSIGN at line 5, position 7
TOKEN: NUM (20) at line 5, position 9
TOKEN: SEMICOLON at line 5, position 11
TOKEN: WRITE at line 6, position 5
TOKEN: ID (x) at line 6, position 11
TOKEN: SEMICOLON at line 6, position 12
x = 20
TOKEN: RBRACE at line 7, position 1
DEBUG: Reached EOF successfully.

Final Symbol Table:
Name    Type    Value   Line
-----------------------------
x       int     20      2
```

9 Results Summary

The results from running all test cases are provided in `results.txt`. Key observations include:

- Correct outputs for variable assignments, expressions, and control structures.
- Proper handling of semantic errors, such as type mismatches and undefined variables.

10 Conclusion

The interpreter demonstrates the effective application of compiler design principles, providing support for variable declarations, arrays, and control flow. Future extensions may include support for functions, advanced data types, and enhanced error recovery.