Practice: Development of the CMM Interpreter

Introduction

The purpose of this project is to familiar you with the tools and principles of compiler, and give you experience in solving typical problems one encounters when using them to generate an interpreter.

In this programming project, you will design and implement an interpreter of CMM language (see appendix A), using the methods you learned in Compiler Principles course. The interpreter will read CMM source codes and make the lexical analysis, syntax analysis and semantic analysis on the codes. If it encounters errors, reports them. If no errors are encountered, your interpreter will give the execution result directly.

There are **4 checkpoints** along the way to make sure you will submit the final project in the end. The grade of each checkpoint contributes to your final grade. Some work should be completed individually; some work may be completed as a group. In the first 2 checkpoints, you should complete the work independently. In the 3rd checkpoint, you may complete the work as a group. In the last checkpoint every member of the group should present your work by your own and answer questions.

Notes:

Before you get started on implementing anything, we suggest you read through the requirements of the project carefully and have deep discussion with your group members, design a skeleton of your interpreter before coding and make your own schedule for time management. If you have no clue on how to start, don’t be shy about opening up the source code of TINY interpreter we provide or feel free to ask questions. You can learn a lot by just tracing through the code. If you can’t seem to make sense of it on your own, you can come to us in the lab hours or by emails. Make sure you do understand the requirement of the interpreter.

Have Fun!

Checkpoint 1

Tasks:

1. Set up your group, each group should not be exceeded 3 people.
2. Discuss with your group members, understand the requirements of the interpreter construction,
3. Read some relevant materials and get to know the development and functions of compilation technology and compilation tools.
   a) You may read chapter 1.7 in Compiler Construction and Practices to compare the difference between TINY language and CMM language.
   b) You may read the source code to learn how the TINY compiler did the compilation and get to know the skeleton of a compiler.
4. Choose Antlr or other tools and get to know its usage in detail.
   a) Give the introduction of the tool you choose, describe the environment installment.
   b) Take the **calculator** as an example to test the usage of the tool, and give the result.
   c) Analyze the CMM language, write its syntactic description file and test it in your tool to generate automatically the Lexer and Parser. (optional)
   d) Summarize the above contents, finalize the study report.

Submission

1. The group leader will submit your group members to TA.

2. You will discuss with your group to design the structure of the interpreter. **Each group** will submit an outline of your project, including the lexical and syntax of CMM you planned to interpret and the project skeleton of your interpreter. This task is worth **4%** of your overall grade. Most of points are allocated for your preparation and schedule of the project.

3. **Each of you** will submit a report, to describe what tool you choose to learn, and how to use it. Take the **calculator** and/or CMM as examples to use the tool you choose. This task is worth **6%** of your overall grade.

Checkpoint 2

Tasks:

1. You will implement the lexical analyzer individually. You may use Antlr or Javacc to create a scanner of the CMM language, or you may write the scanner from the scratch manually.
2. Your scanner will transform the source file from a stream of bytes into a series of tokens containing information that will be used by the later stages of the compiler.
3. If you don't know how to start, you may read Chapter 2.5 to understand how TINY compiler analyzes the input stream and make some modification on the sources code so that it can fit for CMM language.
4. You scanner should
   a) Skip over white space;
   b) Recognize all keywords, operators, delimiter, constants, identifiers;
   c) Recognize single-line and multiple-line comments;
   d) Record the line number and the position of all tokens;
   e) Output the token steam that you recognize.
   f) For each character that cannot be matched to any token pattern, report it and continue parsing with the next character.
5. Test your scanner with the test scripts.
6. Output result format may refer to Chap 2.5 "Program List 2-4" in "Compiler Construction Principles and Practice".
7. Describe the usage of your scanner and the testing you've done in a short report.

Note that lexical analysis is responsible only for correctly breaking up the input stream and categorizing each token by type. The scanner will accept syntactically bogus sequences such as:

```
int a [100} while i ;
```

Submission

This task is worth **10%** of your overall grade. Most of points are allocated for correctness, with the remainder reserved for the error handling explorations.

Each of you will implement the task individually.

,

Checkpoint 3

Task:

1. You will implement this task as a group or individually if you want.
2. Extend your CMM compiler to handle the syntax analysis phase. The parse will read CMM source programs, analyze the syntactic structure. At the end of parsing, if no syntax errors have been reported, the entire parse tree is printed. If it encounters errors, report it clearly and specifically as possible as it can. We expect that your parser has some rudimentary error recovery, and try not to stop parsing when it encounters the fist error.
3. Test your scanner with the test scripts.
4. Output result form may refer to Chap 4.4 “Program List 4-9” in “Compiler Construction Principles and Practice”.

Note that the syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of the CMM grammar. For example, the following program is valid according to the CMM grammar, but it is obviously not semantically valid. It should parse just fine, though.

```
real R[6];
int i=0;
while(i<7)
{
    R[i] = i;
    i=i+1;
}
```

Submission

This task is worth **20%** of your overall grade. Most of points are allocated for correctness, with the remainder reserved for the error handling explorations.

Checkpoint 4

Tasks:

Each group will implement a semantic analyzer for your interpreter based on the former work. Your semantic analyzer will transverse your parse tree and validate that the semantic rules of the language are being respected, printing appropriate error messages for violations. These rules including but not limited to:

1. Type checking;
2. Scoping and declaration check of variables, e.g. variables must be declared and can only be used in ways that are acceptable for the declared type;
3. New declarations don’t conflict with earlier ones.
4. The division of arithmetic expression should not be zero.
5. The index of an array should not be out of bound.
6. The test expression used in an IF statement must evaluate to a **bool** value.
7. so on…

If there is no error, the CMM codes will be translated and executed directly.

If there are some errors, report them, so as to help the programmer fix the mistake and move on.

As always, the first thing to do is to carefully read the requirements and take a look at the test scripts, deign your semantic analyzer before coding. Your program will be considered correct if it verifies the semantic rules and reports appropriate errors, but there are various ways to go about accomplishing this and ultimately how you structure things is up to you. There is no definitive way, but there are good and bad choices.

Finalize all contents including your design, implementation and tests into a report (see Appendix B)

Submission

This task is worth **60%** of your overall grade. The grading policy is as follows:

1. Correctness of basic fumctions with sufficient tests（30%）
2. Project report describes your implementation clearly and correctly that coincides with your source codes. Readable and clear structural code is also expected. (5%)
3. Bonus, e.g. user-friendly interface; additional functions, good design ideas, etc.（15%）

The final project submission includes: source codes of CMM interpreter, the execution program, the test scripts and the project report. All files should be compressed into one file named after your group id, e.g. the submitted file for group 12 should be named as “12.rar”.

Each member had at least 5 minutes to present your work and answer questions.

Appendix A. CMM Language

CMM (C Minus Minus) language is a simple C-like language. It is not an exact match to any of existed programming language. The feature set has been trimmed down and simplified to keep the programming project manageable.

The specification for CMM language includes syntax and semantics. You will need to refer to implementing the course project.

1. **Lexical Considerations**
   a) **Keywords**: The keywords are listed as follows. they are reserved, which means they cannot be used as identifiers or redefined.

      if else while read write int double break

   b) An **identifier** is a sequence of letters, digits, and underscores, starting with a letter. CMM is case-sensitive, e.g., if is a keyword, but IF is an identifier; hello and Hello are two distinct identifiers.

   c) Whitespace (i.e. spaces, tabs and carriage returns) serves to separate tokens, and will be ignored by scanner.

   d) A Boolean constant is either **true** or **false**. Like keywords, there words are reserved.

   e) An **integer** constant can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits. A hexadecimal integer must begin with **0x (**or **0X)** and is followed by a sequence of hexadecimal digits. Hexadecimal digits include the decimal digits and the letters **a** through **f** (either upper or lowercase). For example, the following are valid integers:

      12, 0134, 0xf0, 0X12AF

   f) A **double** constant is a sequence of decimal digits, a period, followed by any sequence of digits (maybe none). Thus, **.11** is not a valid double number, but 0.123 or 12. or 12.0 are.

   g) Operators and punctuation characters used by CMM includes:

      + – * / % < <= > >= = == != ; , [ ] ( ) { }

   h) A **single-line comment** is stated by // and extends to the end of the line. A **multiple-line comment** starts with /* and end with the first subsequent */. Any symbol is allowed in a multiple-line comment except the sequence */ which ends the current comment.

2. **CMM Grammar**

   A CMM program is a sequence of statements, including declaration, if-statement, while-statement, read-statement, write-statement and expression statement.

The reference grammar is given in a variant of extended EBNF where the tokens are in bold, and **ident**, **intconstant**, **doubleconstant** denotes the set of identifiers, integer numbers and double numbers. All nonterminal names begin with capitals.

Program	→	Stmt { Stmt }
Stmt	→	VarDecl | IfStmt | WhileStmt | BreakStmt | AssignStmt | ReadStmt | WriteStmt | StmtBlock
StmtBlock	→	**{** {Stmts} **}**
VarDecl	→	Type VarList;
Type	→	**int** | **double** | Type **[intconstant]**
VarList	→	**ident** { **, ident** }
IfStmt	→	**if** Expr Stmt [**else** Stmts]
WhileStmt	→	**while** Expr Stmts
BreakStmt	→	**break ;**
ReadStmt	→	**read (ident | ident[intconstant]);**
WriteStmt	→	**write(**Expr**);**
AssignStmt	→	Value **=** Expr **;**
Value	→	ident[**intconstant**] | **ident**
Constant	→	**intconstant | doubleconstant | true | false**
Expr	→	Expr + Expr | Expr **–** Expr | Expr ***** Expr | Expr **/** Expr | Expr **%** Expr | – Expr| Expr <= Expr | Expr < Expr | Expr > Expr | Expr >= Expr | Expr != Expr | Expr == Expr | (Expr) | Value **|** Constant

Types

- CMM is a strongly typed language: a specific type is associated with each variable, and variable may contain only values belonging to that type's range of values.
- The built-in base types are **int**, **double**, **bool**.
- Arrays are declared with size information. The number of elements in an array is set when declared and cannot be changed.
- The index used in an array selection expression must be of integer type.
- A runtime error is reported when indexing a location that is outside the bounds for the array.

Variables

- All identifiers must be declared.
- Variables can be declared of base type and array type.
- Each variable has a level of scoping. Inner scopes shadow outer scopes. (see error1_ID). Identifiers within a scope must be unique. Identifiers redeclared with a nested scope shadow the version in the outer scope (i.e. it is legal to have a local variable with the

same name as a global variable.)

Expressions

For simplicity, CMM dose not expect to allow co-mingling and conversion of types within expressions. (i.e. adding an integer to a double)

- The operand to a unary minus must be **int** or **double**. The result is the same type as the operand.
- The two operands to binary arithmetic operators (+ , –, *, /, %) must either be both **int** or both **double**. The result is of the same type as the operands.
- The two operands to binary relational operators (< , >, <=, >=, !=, ==) must either be both **int** or both **double**. The result type is **bool**.
- The operands for all expressions are evaluated left to right.
- Operator precedence from highest to lowest:

➢ [	array indexing
➢ !, –	unary minus, logical not
➢ *, /, %	multiply, divide, mod
➢ +, –	addition, subtraction
➢ <, <=, >, >=	relational
➢ =	assignment

All binary arithmetic operators and both binary logical operators are left-associative. Assignment and the relational and equality operators do not associate. Parentheses may be used to override the precedence and/or associativity.

Control structures

CMM control structures are based on the C/Java versions and generally behave somewhat similarly.

- An **else** clause always joins with the closest unclosed **if** statement.
- The expression in the test portions of the **if** and **while** statements must have **bool** type.
- A **break** statement can only appear within a **while** loop.

Appendix B. Report Specification

1. Background

Group members:

Name	Student Number	Work content

2. CMM Grammar

Describe the lexical and syntactical grammar of your CMM language formally by using EBNF. You may make some modification on the original grammars or add some new features.

3. Design and Implementation

Describe the structure of your CMM interpreter, kernel algorithms, and the tools you used for implementation, etc.

4. Testing

List your test scripts and the test results.

5. How to use (Optional)

Give a short description on the running environment of your CMM interpreter and how to use it.

6. Conclusion

Make your comments on your CMM interpreter.

7. Reference