CpS 450: Language Translation Systems

Dream Compiler Manual

C Level

David John Hayes
4-25-2024

Table of Contents

Introduction

This manual serves as a comprehensive guide to functionalities, features, and technical aspects of this compiler. This technical manual contains detailed instructions on how to operate the compiler, including its command line options and manual assembly and linking procedures. Additionally, it notes the language features supported by this compiler, shedding light on any limitations, extensions, and optimizations applied. Furthermore, this manual offers insight into the technical intricacies of the compiler's architecture, runtime memory management, and I/O mechanisms.

Usage

To run the compiler, follow these steps:

1. Building the Project:
    i. Open Visual Studio Code (VSCode) and navigate to **File > Open Folder**. Choose the folder containing the compiler project (the "dream" folder).
    ii. Open a new terminal in VSCode by selecting **Terminal > New Terminal**.
    iii. Build the project using Gradle by executing the command:

    ```
    gradlew build install
    ```

    - Note: Mac/Linux users may need to make gradlew executable and run:

    ```
    ./gradlew build install
    ```

    iv. This command compiles the project and runs unit tests, ensuring there are no errors.
2. Running the Compiler:
    i. Locate the compiled script at build\scripts\dream and use it to execute the compiler.
    ii. Optionally, you can use VSCode to run the compiler. Open the launch.json file containing run configurations by selecting **Run > Open Configurations**.
    iii. Copy and paste the example launch.json configuration provided below into your launch.json file.
    iv. Choose **Run > Run without debugging**. You'll be prompted to enter command line arguments.
    v. The compiler supports the following command line options:
        - -ds: Enables debug mode for the scanner (lexer).
        - -dp: Enables debug mode for the parser.

- -S: Generates assembly code only, without invoking the GCC compiler for compilation.
- Provide the Dream source code file(s) as arguments to the compiler.
- Example Command:

```
dream [-ds] [-dp] [-S] <filename> [filename...]
```

vi. To run the compiler with debug mode enabled for the scanner and parser, and to generate assembly code without compiling, use the following command:

vii. Provide the filename(s) of the Dream source code file you want to compile.

3. Using the -S Option:
   i. Assemble the Assembly Code and Link with Standard Library:
      - Use the GCC assembler (gcc) to assemble the generated assembly code (.s file) into an object file. The -m32 flag is used to specify 32-bit architecture.
      - Link the generated object file with the standard library (stdlib.o) or any other necessary libraries.

```
gcc -m32 <path-to-file>.s stdlib.o -o <path-to-file>.o
```

      - This command compiles the assembly code into an object file (`<path-to-file>.o`).

   ii. Execute the Program:
      - Once the executable file is created, you can run it by executing the following command:

```
./<path-to-file>
```

   By following these steps, you can manually assemble and link the output assembly code to create an executable file for your Dream program.

4. Testing:
   i. You can make changes to any Java file or the grammar and rerun the compiler. VSCode will automatically rebuild the project.
   ii. If automatic rebuilding doesn't occur, try deleting the build folder containing compiled artifacts. Then, press F1 and enter Java: Rebuild Projects to force a rebuild.

Example launch.json configuration:

```
{
    "version": "0.2.0",
    "configurations": [

        {
            "type": "java",
            "name": "Main",
            "request": "launch",
            "mainClass": "cps450.Main",
            "args": "${command:SpecifyProgramArgs}"
        }
    ]
}
```

Features

Variable Declarations and Assignments: Define and initialize variables of data types persisting of integers and bools.
Control Structures: Utilize if-else statements and while loops (while) to control the flow of program execution.
Functions: Define and call functions with parameters and return values, supporting both built-in (I/O) and user-defined functions; supports recursion as well.

Technical Notes

Compiler Tools and Versions

The tools used for this compiler are as follows:

1. Development Environment:
    - Visual Studio Code (VSCode)
        - Version: 1.88.1
        - Description: VSCode is used for editing/writing code for this compiler, as well as for debugging purposes
2. Parser Generator:
    - ANTLR
        - Version: 2.4.6
        - Description: ANTLR is used for generating lexical analyzers and parsers for the Dream programming language grammar.

3. Build Tool:
    - Gradle:
        - Version: 8.6
        - Description: Gradle is employed for automating the build process of our compiler.

Compiler Organization

1. Input Processing:
    - The compiler iterates over each input filename obtained from the command line arguments. For each file, it reads the Dream source code from the file and creates a CharStream object.
2. Lexical Analysis (Scanning):
    - The input source code is passed to the lexer (MyDreamLexer), which tokenizes the input stream, generating a stream of tokens representing the lexical elements of the program.
3. Parsing:
    - The token stream generated by the lexer is passed to the parser (DreamParser), which constructs a parse tree representing the syntactic structure of the program.
4. Error Handling:
    - Custom error handling is implemented to suppress default error messages and provide custom error reporting using MyDreamErrorListener.
5. Semantic Analysis:
    - After successful parsing, the compiler performs semantic analysis to enforce language constraints and verify the correctness of the program's semantics. Semantic checks are performed using a custom SemanticChecker.
6. Code Generation:
    - If the semantic analysis passes without errors, the compiler proceeds to code generation. It utilizes a CodeGen object to traverse the parse tree and generate target instructions.
7. Output Generation:
    - Target instructions generated during code generation are written to an assembly file (.s) using the writeAssemblyToFile method.
8. Optional Compilation:
    - If the -S option is not specified, the generated assembly file is compiled into an executable using the GCC compiler (gcc method).

9. Compilation Report:
    - The compiler provides feedback to the user at each stage of the compilation process, including syntax errors, semantic errors, and compilation success or failure.
10. Intermediate Representation: Abstract Syntax Tree (AST):
    - The AST represents the hierarchical structure of the input program after parsing. Each node in the AST corresponds to a syntactic element of the program, such as expressions, statements, and declarations.

Dream Runtime Memory Management

To understand the activation record, here's an example of how it works:

- Dream File:

```
class CReadint is

  start() is
    x: int
    y: int
  begin
    x := in.readint()
    y := in.readint()
    out.writeint(x)
    out.writeint(y)

  end start
end CReadint
```

- start Function:
    - When the start function is called, its activation record is created on the stack. The stack frame for the start function includes:
    - The old base pointer (%ebp) is pushed onto the stack to preserve the caller's base pointer.
    - The current stack pointer (%esp) is moved into the base pointer (%ebp) to establish the base of the new stack frame.
    - Space is allocated on the stack for local variables. In this case, no local variables are declared explicitly, but temporary storage is allocated for function parameters.
    - Function parameters are pushed onto the stack for the readint function calls.
    - The return address for the call instruction to readint is stored on the stack.
    - After each readint call, the returned value is stored in the appropriate location within the stack frame.

- main Function:
    - The main function is a wrapper function that simply calls the start function.
    - As with any function call, a new activation record is created on the stack for the main function.
    - The start function is called, and its activation record is created on top of the stack.
- Exit Call:
    - After the start function completes execution, the stack is cleaned up and restored to its previous state.
    - The exit system call is invoked to terminate the program, passing 0 as the exit status.
- Activation Record Organization:
    - Each stack frame begins with saving the old base pointer (%ebp) and updating it with the current stack pointer (%esp).
    - Local variables and function parameters are allocated space on the stack.
    - After the function executes, the stack frame is deallocated by restoring the previous base pointer and stack pointer.
    - Function parameters and return addresses are used to transfer control between functions.

I/O and Memory Management

Input and Output are handled through integrated Reader and Writer classes. The Reader class contains the function `readint()` and is called like this:

```
in.readint()
```

Whereas the Writer class has one function `writeint(int x)` which takes a parameter x that is an integer. `writeint()` is called like this:

```
out.writeint(x)
```

These two classes handle the I/O of the Dream language. These functions are called as my own home-grown functions. The functions were written in C and then compiled to an object file (`stdlib.o`). Then, when compiling the .exe files to run a Dream program, the object file is passed in so that the Dream file has access to both the `readint()` and `writeint()` functions, which in turn allows the assembly file created to access the functions as well.

Testing and Bug Report

Official Test Files

- D Tests:
  - dassign1.dream: Successful
    Output:
    ```
    1
    1
    0
    0
    1
    1
    0
    15
    2
    14
    -4
    0
    ```
  - dif.dream: Successful
    Output:
    ```
    2
    0
    2
    3
    ```
  - dwhile.dream: Successful
    Output:
    ```
    1
    2
    3
    4
    1
    1
    1
    2
    1
    1
    1
    2
    1
    1
    1
    2
    ```

- C Tests:
  - cbasics.dream: Successful

    Output:

    ```
    0
    10
    20
    0
    10
    -5
    0
    10
    20
    -5
    25
    ```

  - cchange.dream: Successful

    Input:

    ```
    164
    ```

    Output:

    ```
    6
    1
    0
    4
    ```

    Input:

    ```
    4067
    ```

    Output:

    ```
    162
    1
    1
    2
    ```

  - cfact.dream: Successful

    Input:

    ```
    7
    ```

    Output:

    ```
    7
    5040
    ```

  - cgcd.dream: Successful

    Input:

    ```
    4098
    432
    ```

    Output:

    ```
    6
    ```

  - citerfact.dream: Successful
    Input:
    ```
    5
    ```
    Output:
    ```
    0
    120
    ```
  - creadint.dream: Successful
    Input:
    ```
    83793
    -9191464
    ```
    Output:
    ```
    83793
    -9191464
    ```
- No bugs known.

Appendices

(Attached below)