Compiler Architecture

Mario Horacio Garrido Czacki

March 2020

Introduction

This compiler has been designed to be as flexible as possible. This means that once working, one should not mess with its code. To this end, I have abstracted it to a series of automata (actors) that merely operate over a series of specification files for the desired language. Any modifications or other languages are therefore compatible with this compiler's architecture, thus permitting wide flexibility and easy human modification of the language specification.

Actors

The compiler's basic architecture is composed of the following actors:

Interface

The Interface's task is receiving the user's commands. An example would be the compilation instruction or the path to the source code one wishes to compile.

Output

- A string with the pathfile to compile.

Reader

The Reader's task is to load the user's source code into a string. This actor should be able to handle all operations related to file reading.
In addition to the former, the Reader should also be able to load the language's specification files (c_tokens and c_structures) into a generic version of their canonical structures.

Output

- SCS: A string with the source code file's contents.

- GTL: A List of tuples which will contain the identifying atom of the token type and its RegEx.
- GAST: A List with all possible Abstract Syntax Tree structures.

Lexer

The Lexer will receive both the SCS and the GTL. This will allow it to tokenize the SCS by matching the RegEx of each token contained in the GTL. For each illegal token found, the user should be notified of the offending token and its position via the Interface and no OTL should be generated.

Output

- If all tokens are legal:
  - OTL: A Token List generated from the SCS.
- If there are illegal tokens:
  - TEL: A list of tuples containing three elements, the illegal token string, the token's row, and the token's column.

Parser

The Parser will receive both the OTL and GAST and will then generate an Output Abstract Syntax Tree (OAST) for the program.
Each Token's usage context must be respected. For this purpose, GAST contains all legal production rules for all possible substructures. If there are any context violations, the user should be notified of the offending Token and its position via the Interface and no OAST should be generated.

Output

- If all tokens are contextually legal:
  - OAST: An Abstract Syntax Tree with the entire program logic.
- If there are context violations:
  - ASTEL: A list of tuples containing three elements, the context violating token, the token's row, and the token's column.

Code Generator

The Code Generator will receive both the OAST and GAST in order to generate the Assembly code for the OAST. The GAST will contain the Assembly equivalents for every substructure, so by processing the leaves of the OAST and going up the tree the the AS will be generated.

Output

- AS: An Assembly string representation of the source code.

Writer

The Writer is in charge of receiving the AS and writing it into a text file for compilation. Naturally, this module should be able to handle exceptions and errors related with file writing.

Output

- AT: An Assembly text file ready to be turned into object code.

Invoker

The Invoker is the last step in the compilation process, as its task is to pass the pathfile of AT and necessary arguments to GCC in order to generate the final compiled object code.

Output

- A call to GCC with the necesary parameters to generate the executable.

Sequence Diagram

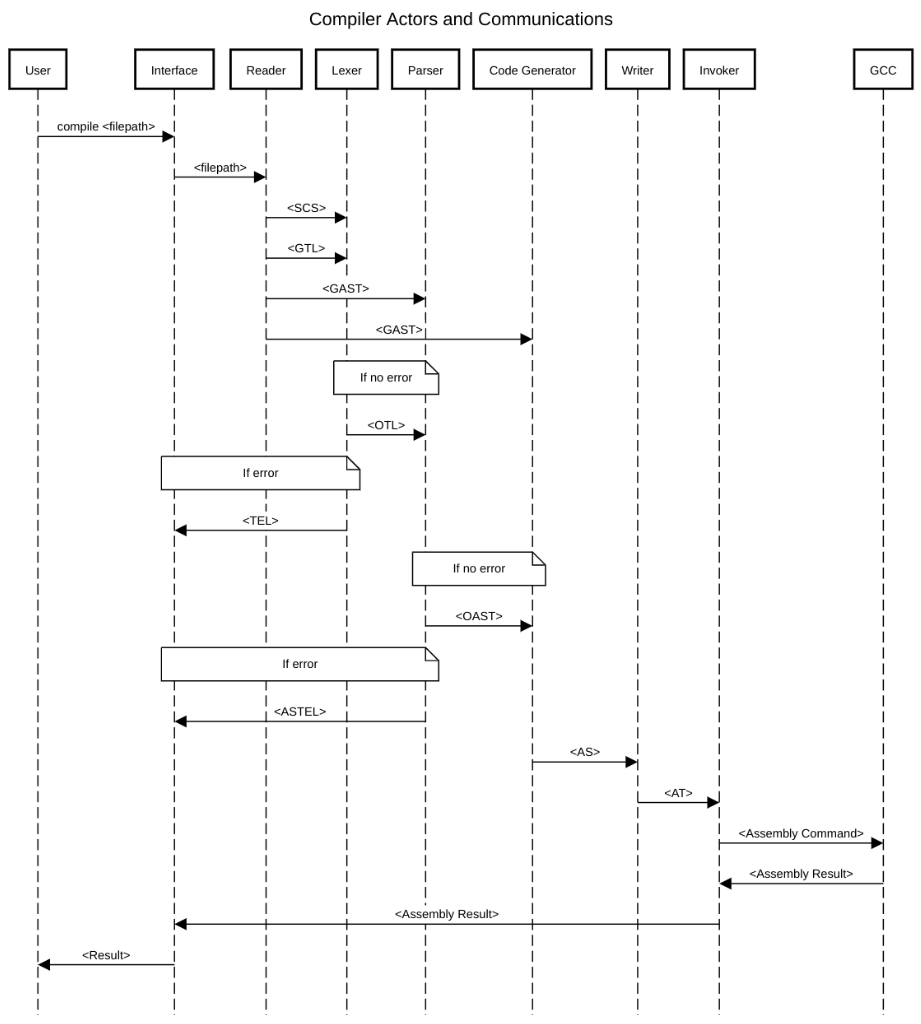

Figure 1: Sequence diagram of the Actors involved in the Compiler process and their communications.

Data Structures

The following data structures are integral to the communication between actors, and thus they must be well defined:

Token List

A List consisting of Token structs. This represents the tokenized strings and their positions in the document. Tracking the position of the token allows us to trace any errors to the source code and string that caused it.

A Token struct contains:

1. Tag: Token's canonical name. E.g. literal, sum, open-parenthesis, etc...
2. Expression: Matched string. String that has been matched to the RegEx of the token. This is mainly used to save the value of a literal.
3. PosY: Row number of the token in the source code.
4. PosX: Column number of where the token starts in the source code.

Note that in either the generic version of this list (GTL) or with tokens which posses no variance (if statements, for example) some of these values may very well be empty.

Abstract Syntax Tree List

A List made out of Node structs. Node refers to a Node within an Abstract Syntax Tree. A Node struct contains:

1. Tag: Structure's canonical name.
2. Token: The Token structure that references this part of the structure. This may be empty.
3. Children: List with the child substructures (also Abstract Syntax Trees) in the order specified in the GAST. Any non-valid or non-aplicable child will be assigned the nil value. If the structure has no children (for example a literal) Children will be empty.
4. Class: List with all applicable classes for this Node. This cannot be empty.
5. ASM: String file with the equivalent Assembly Language Code.

Specification Files

Specification Files are a crucial component of this compiler. They have the important task of providing all of the necessary specifications of the C language, and allow us to easily expand upon the compiler without changing any code whatsoever. If at any moment compiling another language was necessary, another set of Specification Files would suffice to implement it in its entirety.

c_tokens

XML File that contains <token> elements with two subelements:

- tag modifier: Name of the token in its generic shape. For example: plussign, if, else, literal, variable, etc...
- <pattern>: Regular Expression that matches all posible tokens of this kind.

c_structures

XML File that contains <structure> elements with various subelements:

- tag modifier: Name of the structure. For example: sum if-statement, evaluation, literal, variable, etc...
- <token>: Token that invokes this structure. If empty, no token is necessary as long as all substructures are matched (this is helpful for operations).
- Zero or more <substructure>:
  - tag modifier: Name of the substructure. For example in an if-statement: condition, main-body, else-statement. Must be unique within the <structure>.
  - <contents>: Contains the names of the <class> tags that apply to this substructure. For example, within the condition of an if-statement: evaluation, operation, comparison, etc...
- One or more <class>: Contain the possible classifications of this substructure. For example, a comparison would have the classes: value, operation, etc...
- <asm>: Contains the Assembly Language equivalent of this substructure. Some considerations:
  - You may refer to the <substructure> results via the format :number, where number is the zero-based index of the substructures of this structure. For example, a sum structure would have two substructures, element_a and element_b. You would refer to them as :0 and :1 respectively.

- Any temporary registers you require can be accessed by :tnumber where number is the zero-based index of the temporary register. Example: :t0, :t1, etc...
- The result register can be accessed via using :r.

A live example of this would look like the following:

```
mov :t0, [:0]
mov :r, [:1]
add :r, :t0
```