COMPILER

Paper

Ian An
QAN001@uchicago.edu

This paper discusses some of the critical achievements and obstacles that are encountered over the development process. They are generally grouped based on the phase of compilers.

1 Scanner

In scanner, we produced all the tokens by scanning the characters in the source file one by one.

We followed the basic rule that the longest possible string is to be located and matched to the corresponding tokens. We adopted a string buffer to save the accumulated characters that are yet to be matched with the tokens. Additional characters are appended to the end of the string buffer so long as the current string buffer is consistent with the legal patterns of the language. Once the addition of a new character disqualifies the string in the string buffer as a valid expression defined by the language, we roll back one character and match the string before the appending action to a valid token. There must be a valid match because additional character is only appended to a valid string. The matching of string to a token is only performed when a longest possible string is located (after rolling back from an invalid candidate) to ensure the longest possible patterns are adopted.

We made use of the regexp package from golang, to help us get a set of expressions that are defined by the language. The expressions are compiled once in the early phase and used as reference for comparing with the string buffer from the source file.

Apart from the expressions compiled by the package, we arranged all keywords into a few sets (maps in golang) as if they are dictionaries for us to look up during scanning. A valid string is tried to be matched with the keywords first due to their high priority. A variable or number token is issued only when a valid string cannot match with any keyword.

The process described above is repeated until the end of the file. Upon seeing the last character of the file, we examined and tried to match the string buffer for one last time, because the buffer cannot grow any longer.

There are several special conditions we need to consider. Firstly, we paid special attention to newline characters. They are ignored under most situations, but they are significant when signaling the end of a line on a line with comments. We adopted a flag to indicate if the current string being considered is part of a comment and ignored everything except for a new line character. As a result, the subsequent characters are no longer overlooked. Secondly, we dealt with the OR operand “||” as a special case. They are treated as a single operand but are actually two characters. Thus, we peeked ahead one character when encountering a single “|”. Finally, the different representations of a new line character on different operating systems also need attention.

2 Parser

Parsing, which is also known as syntactic analysis, aims to align the tokens produced from the previous step with the defined grammars. In our project, we implemented a Recursive Descent Parser (RDP) for Golite, which is responsible to produce an Abstract Syntax Tree (AST) based on the input of tokens from scanner.

RDP is easy to implement, but we made some mistakes at the beginning. Used to the practices learned from the previous homework on parser, we accidentally terminated the parser process prematurely. We did not clearly understand the meaning of “nil” as a return value. Instead of an error condition, we considered it as an empty node, which reported an error when the absence of an element is allowed.

We experienced some structural changes over the implementation. Initially, we read in tokens one at a time from scanner. But it displayed some difficulties when a backtrack is needed when multiple grammars fulfil the requirement of a nonterminal. We could not find a way to unread the tokens. There are methods provided in Golang to unread characters with a limit in length. Eventually, we had to adopt the suggested way in the previous homework, essentially to read all tokens from scanner when initializing parser and store them in a slice of tokens locally. Such data structure allows us to refer to different tokens with an index. This is not an ideal way for larger project, but it is the best that we can think of.

3 Semantic Analysis

Building symbol table and performing type checks are the two goals of this step. Symbol table is essential in the subsequent steps as we include information of each variable, function and structure within the respective scopes of existence.

We shall first discuss the overall structure of the symbol table. A separate symbol table is built for each scope, e.g., a new scope is necessary when entering a function. Each symbol table are independent but interrelated. They need to refer to the symbol table of the outer scope as the parent, as we need to locate the variables that are defined in the outer scope. Inside the symbol table, a scope name is defined for future reference. A hashtable, which is essentially a map that matches each variable within the scope with its datatype. Additional fields including parameter types and parameter names are defined for symbol tables of functions. These fields make it easy for us to verify the correctness of arguments in invocations based on the parameter lists, and similar tasks. With respect to data types, there are three of them in general, i.e., variable for basic types, functions for function routines and structure for pointer types. In specific, the variable types save the detailed data types (Boolean, or int) in terms of singleton, values and allocated register IDs for ILOC translation. The function types save the return type and a link to the symbol table of the inner scope of the function. Similarly, the structure types save the allocated register IDs for ILOC translation and a link to the symbol table of the inner scope of the structure. Certain fields are defined for a general symbol table or data type, which may be unnecessary. For example, parameter types are essential for functions but meaningless for normal variables. But these fields are defined in general to fulfill the requirements of the interface. Though additional fields are to be maintained, we take advantage of the ease brought by inheriting from the common interface.

We categorize all the grammars into two categories, expressions and statements. Statements play important roles in constructing the symbol table, while expressions are essential in the process of type check.

Constructing symbol tables is the first step of semantic analysis. The goal is to define the scope of each variable based on the declaration and check any duplicate declarations. Upon

encountering a new declaration, report error when the same variable has been declared in the past, include it in the symbol table otherwise.

With the symbol table, type checking follows as the second step. It is worth mentioning that the following items are carefully examined during the process.

- While the symbol table is being constructed, each variable is added to the table without specific information. The corresponding data type is being added to the table during static type checking.
- Matching of data types on both sides of an assignment: Obtain types of both sides with the help of locally defined "GetType()" function and perform a cross check. This also encompasses the case for assigning the return value of a function to a variable. There are basically 6 kinds of data types being defined in our project, i.e., integer, boolean, function, structure, void (an indication for void return type from a function) and unknown (an indication for an undefined or wrong data type). All of them are expressed in terms of singletons.
- Matching of data types on both sides of a binary operator: Obtain types of both sides and perform a cross check. Specific data types are required by certain binary operators, which leads to a further examination of the detailed data type.
- Verifying whether the item on the left of an assignment is assignable: Obtain type of the item on the left of an assignment operator and ensure it is a variable, a structure or a field of a structure.
- Matching of invocations and function definitions: A cross check is to be performed between the argument list of the invocation and the parameter list of the function, which is obtained from the symbol table.
- Matching of the actual and defined return types: This is accomplished when performing static type check on the return statement. The defined return type is obtained from the symbol table and cross checked with the actual data type included in the statement itself.

4 ILOC Intermediate Representation

While semantic analysis concludes the front end of the compiler, ILOC translates leads the process into the back end. The input to this step is the Abstract Syntax Tree that is constructed by parser and refined by semantic analysis process. A slice of function fragments is expected as the output, while each function fragment consists of a slice of ILOC instructions. These instructions are grouped into different function fragments according to the scope of the functions.

As the input to ILOC translation is the constructed AST, the translation is invoked upon each node of the tree. However, different types of function signatures are defined. This also explains the way each function fragment and ILOC instruction is stored in our project.

- Each program is composed of several functions, so the output on the level of program is supposed to be a list (slice) of function fragments.
- A function definition and all subsequent statements delineates a function. Upon translating a function to ILOC representation, a new function fragment is created. All the ILOC representations translated from the statements within the function are appended to the function fragment, which is eventually returned as the output of the function.

- Following from the previous step, statements within functions are components on a smaller scale. As the translated ILOC instruction is appended to the slice of instructions provided, the input and output on the level of individual statements are the slice of ILOC instructions.

We implement the interface being provided for all instructions. Basically, the interface encompasses the possible elements that may be exist in a variety of ILOC instructions, e.g., source register, target register, etc. We included some functions in the interface to fulfil our needs. For instance, a function used to access the source string in an instruction is included, which is necessary when extracting components like global variable names that are essentially of string data types from the instructions. The functions defined by the interface are implemented by all instructions.

In addition, we also expanded the operand type to cover more cases. Initially, there operand types include only registers and immediate constants. We considered and included the cases for using global variable as operands, using different number of operands within instructions (load is a typical example).

As mentioned in the section for semantic analysis, a specific register ID is declared and associated with each variable when constructing symbol table. This is accomplished to enforce that variable and register is a one-to-one mapping, so that the value of each variable is maintained properly. The number of registers used is not a big concern, because an unlimited number of registers is allowed in this step.

In general, this step is straightforward, but it requires the programming skills to manipulate different data stored in AST based on their specific structures.

5 Code Generation

This is the final step of a compiler. The target is to generate assembly instructions based on the ILOC instructions produced in the previous step. Given that the input to this step is a slice of function fragment, the process of generation is to loop through every single fragment, and every single inner ILOC instruction in turn.

In our project, the key linkage between code generation and the previous steps is the register IDs allocated for ILOC instructions. Due to the way we allocated, each register ID is on a one-to-one mapping with variables. On the other hand, the offset for each variable on the stack is critical in code generation as we adopted the naïve register allocator. When starting with each function fragment, we allocate the offsets for each variable based on the register IDs in ILOC instructions. In this way, we maintain the uniqueness of each variable, while interacting with the complex symbol table to a minimum extent.

Besides, we need to consider the register allocation for function parameters. We have to rely on the symbol table for gathering the information. Instead of passing symbol table over different functions, we extract the parameters information, basically the positions in the parameter list and their register IDs in ILOC instructions. We only invoke other functions with the limited necessary information.

6 Miscellaneous

There are a few obstacles that we encounter throughout the process. Luckily, we tackle them by either finding some optimal solutions from Ed or applying workaround methods. These obstacles are briefly discussed in this section.

6.1 Import cycles

We constantly encounter the error of import cycles in Golang. Basically, it arises when two or more source files import each other in the header. It may not essentially cause any error, but it is forbidden in Golang. This is hard to avoid as certain functions are needed considering the interactions between different modules.

Eventually, we extract the commonly used methods and group them into a separate folder named “utility”. In this way, each file that takes advantage of the frequently used functions need only include the utility package, which simplifies the overall structure, enforces the modularization of software development and avoids unnecessary import cycle errors.

For a similar reason of simplifying the project structure, we decide not to pass symbol table to every translatetoAssembly() function. Instead, we calculate the offset for each variable based on the symbol table ahead of time and pass the useful information only.

6.2 The structure of symbol table

The structure of symbol table becomes increasingly complex as the development proceeds. We included almost all relevant information, e.g., data types, variable values, variable register IDs in ILOC representations, scope name, function parameter list, function parameter types, function return types, etc. Not all of the information is necessary for all different types of variables, but they have to be implemented for all if they are needed for one specific type. This is enforced by the interface. It was hard to manipulate over the symbol table structure, but we also implemented many helper functions to navigate through it.

6.3 Backtracking in parser

As briefly discussed in the section of parser, we adapted the way to store tokens in parser for the purpose of backtracking. The previously adopted way is to read tokens from scanner one at a time, which is elegant but challenging when the token stream does not match with one of the many possible grammar patterns. Instead of continuing the parsing process directly after failing, we need to retrieve the tokens up until the most recently accepted AST nodes. So, we store all tokens in parser and refer to them with indices. Though not a perfect solution, it is flexible and easy to manipulate along the token stream.

7 Conclusion

As the project continues, the development activities became increasingly enjoyable. Neither my partner nor I have a Computer Science background, which makes the initial phase extremely challenging. It took us a lot of effort to tackle the semantic analysis step, and the entire Thanksgiving holiday on ILOC translation. Though it was a hard time, I appreciate that I can even witness my progress in not only the understanding in compilers, but also the general programming and problem analysis skills. Hope everybody enjoys and achieves what they want. Thanks.