Compiler Documentation

Thomas Maloney
net-id: tmaloney@iastate.edu

April 2021

Contents

1 Overview

The compiler has been rewritten in C++ and has gone through a complete "makeover" so to speak. The documentation on the old C code from previous versions is in the document `developers-old.pdf` which is very much outdated and exists purely for historical reasons. The documentation is now broken up into a section for each namespace in the source code to give an overview of the purpose of each module and the data structures used.

2 The Default Namespace

2.1 main.cpp

This file contains the entry point of the compiler. It passes the arguments from the command line to a method that parses them (described in section 2.2) and returns a `struct` that contains info about what flags were passed and what the possible output file might be (should that have been given). I then check which flag got passed in and whether the output should be written to a provided file or just `stdout`. Contains helper methods to handle manually running the lexer, or running the parser and handling the list of translation unit syntax tree nodes (either to be ran through the syntax checker for part 2 or passed to the binder for parts 3+).

2.2 arg_parser

This actually has not changed much, if at all from the original C variant, so to paraphrase what the C variant did:

> The header file contains a `struct`, an `enum`, and a method signature that gets implemented in `args_parser.c`. Here I use `getopt` from `unistd.h` to read/parse each flag and then record which flag and whether there was an output file to a `struct` that I return. The potential output file gets passed back as an out-parameter. If no flags were passed in, or one that doesn't exist, I print out the string that describes how to use the program to `stderr`.

2.3 driver

The `driver` class was written so that flex and bison can "communicate" and also gives me a single interface with which I can interact with the lexer and parser. The C++ documentation for bison was pretty underwhelming (at least for use cases where you wanted to treat the parser like you were using C++ and not just trying to interface with C) and the C++ documentation for flex was almost nonexistant so the `driver` class is the result of what I *think* is necessary for them to pass data between eachother in C++ land.

As for custom data I store in the driver, I keep a list of `TranslationUnitNode`s that get synthesized by the parser after it finishes its rounds so to speak. There's more on that in section 5. I also keep track of the current file (although multi-file code support is effectively dropped in part 3 onward, still technically works in parts 1 and 2 though). I also keep track of a list of `LexemeDataNode`s (see section 2.4 below) and a flag to indicate if any errors occured.

2.4 lexeme_data.h

This defines a class `LexemeDataNode` which contains token information generated by the lexer. The purpose of this class is to allow me to collect the token data neccessary for output in part 1 without having to print the data immediately to the console. Ultimately, the reason why I didn't just build a list of strings instead was so I could fix any output formatting issues in one spot instead of having to `grep` through the lexer.l file.

2.5 lexer

The biggest change that was made to the lexer from the previous version is that it now interacts with the `driver` class and passes all tokens through to bison as a `SyntaxToken` data type. More on that in section 5. Other than that, it still just handles the parsing of lexemes.

2.6 mycc.ypp

The bison grammar file, imports almost all (if not actually all) syntax node classes from the `Syntax` namespace described in section 5.

2.7 part_two_syntax_check

The `PartTwoSyntaxPrinter` class takes in the root node of the syntax tree keeping a (slightly less crude than before) "symbol" table of the variables, structs, and functions (along with their params, and local structs and variables). It then calls the instance of the `Logger` it gets passed and prints out the syntax information of the file that it parsed.

2.8 qsem

The "Quick Semantic Analyzer". It's quick since it actually isn't doing any analysis, rather it's just the code that logs the results from the analysis (which is actually done by the `binder` as described in section 6). Also technically in its own namespace since originally when I was porting to C++, `qsem` was going to actually perform some half-backed semantic analysis just as a sanity check while I was working on the binding code.

2.9 syntax_tree_printer

This is just an addition I made to help me debug parsing and to some degree binding as well, it can be invoked like `./mycc -6 [input file]`. Note that it only prints to standard out.

3 The Logging Namespace

3.1 Logger

A generic logging class that takes in an output stream on construction so to direct the output of the log calls.

3.2 Diagnostics

A class that essentially acts as a wrapper around a list of strings. The interface for this class is just a list of functions that report semantic errors to a private vector of strings. This way, if I want/need to change the message that gets reported, I just need to change a string in one method as opposed to searching for it throughout the codebase.

3.3 Part Three Info

Contains the structs: `PartThreeVariableInfo`, `PartThreeStructInfo`, `PartThreeStatementInfo`, `PartThreeFunctionInfo`, and `PartThreeInfoList`. These essentially act as the "part 3" equivalent of the `PartTwoSyntaxPrinter` class (see section 2.7). The biggest difference in this case (other than the fact that the structs above contain info about the bound tree and the class from section 2.7 deals with the syntax tree) is that the part three info structs are built during binding, instead of while traversing the bound tree afterwards. This is done since I don't attach `SyntaxToken` information to every `BoundNode` like I do with every `SyntaxNode`.

4 The Symbols Namespace

4.1 Symbol

Abstract class that resembles some symbol type that would be stored in a symbol table. Contains a protected field to store the name of the symbol (e.g. a function name, type name, variable name, etc.) and an abstract method for getting what kind of symbol it is.

4.2 Parameter Symbol

Represents a parameter to a function. Contains a `TypeSymbol` (see section 4.4) instance that records what C type the parameter is (e.g. int, const float, char[],

struct somedata, etc). Also records whether the parameter is a constant and if it is an array type.

4.3 Function Symbol

Represents a function in code. Like the `ParameterSymbol`, this also contains a `TypeSymbol` instance to keep track of the type of value the function returns. It also contains a list of `ParameterSymbol`s that keep track of the function's parameter info. Since there can exist multiple function prototypes for a single function, I keep track of the line that the function is defined on so when looking defining the function, I can check whether it already has been defined or if it was just declared.

4.4 Type Symbol

The `TypeSymbol` class sort of ended up as the "mac daddy" of the other symbol types. It contains an instance of a struct `TypeAttribute`s which keep track of whether the type is a struct, an integer type, a numeric type, an array, and/or a constant. The `TypeAttribute`s struct has a partial ordering (`std::partial_ordering` in C++20) defined on it that says for any two given `TypeAttributes` α_0 and α_1, we have

$$\Leftrightarrow(\alpha_0,\alpha_1)=\begin{cases}\text{undordered} & \text{only one of } \alpha_0 \text{ or } \alpha_1 \text{ is either an array or a struct}\\ \text{less} & \alpha_0\in\mathbb{Z} \text{ and } \alpha_1\in\mathbb{R}\setminus\mathbb{Z}\\ \text{greater} & \alpha_1\in\mathbb{Z} \text{ and } \alpha_0\in\mathbb{R}\setminus\mathbb{Z}\\ \text{equivalent} & \text{all of } \alpha_0 \text{ and } \alpha_1 \text{ attributes are equal}\\ \text{unordered} & \text{otherwise}\end{cases}$$

(I know technically $x\in\mathbb{R}\setminus\mathbb{Z}$ isn't the best (or even remotely accurate) way to represent x is a `float` and not an `int`, but I felt it still gets the message across). (Also, the reason for choosing the symbol $\Leftrightarrow$ is because C++20's three way comparitor is written `<=>` which in my case returns the partial order). I then defined a partial order on the `TypeSymbol` class itself that says for any two `TypeSymbol`s τ_0 and τ_1, we have

$$\Leftrightarrow(\tau_0,\tau_1)=\begin{cases}\alpha(\tau_0)\Leftrightarrow\alpha(\tau_1) & \alpha(\tau_0)\Leftrightarrow\alpha(\tau_1) \text{ does not evaluate as equivalent}\\ \text{id}(\tau_0)\Leftrightarrow\text{id}(\tau_1) & \text{otherwise}\end{cases}$$

Where $\alpha(\tau)$ represents the `TypeAttribute`s of some type τ and $\text{id}(\tau)$ returns the name of some type τ.

This partial order can be intuited as representing whether a given type can be "widened" to another given type. For example, $\Leftrightarrow(\text{char},\text{float})=\text{Less}$ which means that `char` can be widened to a `float`. Types that are "unordered" are unable to be cast to eachother, both implicitly and explicitly.

One of the convinient parts about this is that it makes it relatively easy to determine the wider type of two types in a binary expression (which also makes it easy to determine if the operator is valid on two types).

4.5 Struct Symbol

The `StructSymbol` class represents information about a user defined struct. It contains a list of `VariableSymbol`s which are used to keep track of the type information about the struct's members.

4.6 Variable Symbol

The `VariableSymbol` class represents information about a variable, be it global or local (or in the case described in section 4.5, represents members of a user defined type). It contains information the type of the variable, whether or not it is an array type (and if so, what its size is), and whether or not it is a constant.

5 The Syntax Namespace

5.1 Overview

While there are a metric ton of files and classes in the `Syntax` namespace, they are all relatively light. Each class essentially just represents a production on the parse tree, with some information added or discarded as needed. The `Syntax` namespace and corresponding classes were added to provide a more typesafe way to interact with the syntax tree. One of the exceptions to this is the `SyntaxToken` class, as it isn't really a node on the syntax tree, but rather an attribute of the non-terminals, containing info like the token type, the line number, and the text.

6 The Binding Namespace

6.1 Overview

Like the `Syntax` namespace (see section 5), the `Binding` namespace *also* has a metric ton of files and classes. Fortunately, they are also relatively light. The bound nodes contain similar info to the syntax nodes, however, they don't keep track of the `SyntaxToken`s like the syntax nodes do, and instead of storing the evaluated type as just a string, they store bound `TypeSymbol` references.

6.2 Binder

The `Binder` class acts as the driver of the semantic analysis and binding process and is (unlike the other classes in the namespace) a pretty heavy class. The `Binder` class essentially performs a recursive descent on the syntax tree built from the parser and then synthesizes the attributes of each `BoundNode` from the bottom up (save for some inherited things like scope).