Ccash Compiler Documentation

Architecture Overview

The compiler is split into four projects, `Ccash`, `Ccash.Antlr`, `Ccash.CodeGeneration`, and `Ccash.SemanticAnalysis`. The `Ccash` project is the one that generates the final executable, while `Ccash.Antlr` contains the grammar file for C$, `Ccash.g4`. Whenever `Ccash.Antlr` is built, Antlr is used to generate the lexer and parser from the grammar file. It is linked as a library by `Ccash`, which should always force a rebuild of `Ccash.Antlr`, ensuring that the generated code is always up to date.

The entry point of the compiler is located in the `Ccash.cs` file, which calls the rest of the code to handle parsing, semantic analysis, and code generation.

Finally, `Ccash.SemanticAnalysis.Tests` is where the code to run the SNCAG algorithm is located, in the form of a unit test.

Lexical and Syntactic Analysis

This is entirely handled by Antlr, which takes care of generating a basic Abstract Syntax Tree for us.

Semantic Analysis

The AST obtained through Antlr is then traversed twice. The first traversal uses the `GlobalScopeListener` and generates function headers so they can be referred to later during the second traversal, which will take care of compiling the function bodies.

Since every function needs to have a unique symbol, both for our Symbol Table and for the LLVM IR, function overloading is realized by mangling the name of the function with the type information of its parameters. This is handled by the `FunctionNameMangler` class. It is mainly used when adding the functions and methods of primitive types to the global Symbol Table.

Overload Resolution and Type Coercion

As its name suggests, the `FunctionOverloadResolver` figures out which overload of a function (or operator, which are just functions) to call given a set of arguments. It does this by first searching all the overloads of the function, then filtering out those that have the wrong number of parameters. From there, it looks for constructors marked as `implicit` between the types of the parameters and the types of the arguments, which are defined by the different `CcashType` descendants. The overload that requires the fewest implicit conversions is selected as the one that will be called, and the arguments are coerced into the correct type by calling the implicit constructors.

One thing to bear in mind is that not all type coercions between numeric types are safe to perform. For instance, implicitly casting `float64` into `float32` incurs a risk of losing precision, and should be avoided. Similarly, implicitly casting a signed integer to an unsigned integer is unsafe due to the fact the value might not be preserved if it was negative. Implicitly casting an integer to another integer of a smaller size is problematic because the value might not fit. However, an unsigned integer may be implicitly converted

into any larger integer type of any sign, since its value will always be preserved.

Code Generation

The generation of LLVM IR for each function happens right after the semantic analysis of the function and all its statements in `CompilationListener`. The code generation is facilitated by the `IRBuilder` class, which acts as a convenient wrapper around LLVMSharp's builder.

The classes in the Intrinsics folder handle the generation of code that LLVM natively understands, such as most operations on primitive types. The rest of the heavy lifting is done by `Generate()` in `StatementGenerator` and `Expression()` in `ExpressionBuilderExtensions`.

Output

The compiler will output two files for a given `.ccash` file: a `.bc` file and `.ll` file. The `.bc` file is LLVM bitcode that can be executed by `lli` or compiled into an executable with `clang`, while the `.ll` file simply contains the generated LLVM IR in readable text and can be viewed with any text editor.

In the case of an error in the generated code, the compiler will output a `.ll` file containing the LLVM IR of the function containing the error.

Adding Functionality to C$

Putting it all together, adding new functionality to the language will generally involve these steps:

- Adding the syntax to `Ccash.g4`
- Adding the required new AST node classes and handling them in `IExpression.Create()` or `IStatement.Create()`
- Performing the semantic analysis, handling errors, etc
- Adding functions to the `StatementGenerator` or to `ExpressionBuilderExtensions` to generate the appropriate LLVM IR