Project 5: Code Generation

Tyler Burkett, Dr. Jerzy Jaromczyk

November 28th, 2023

For this project, you will be finalizing your compiler by creating one more visitor to walk the AST and generate LLVM IR code along the way. Even though we are not generating architecture-specific assembly, our tool is still a compiler; it translates from one language to another. By generating LLVM IR, you will have access to a wide range of tools which you can use to do more than just continue onto machine code. These include interpreters, optimizers, and various other analysis tools, all of which work with LLVM IR.

1 B-Sharp Language

The B-Sharp language is a superset of the B-Minor language in the text book. In other words, all aspects of the B-Minor language as described in the textbook also apply to B-Sharp, with some additions. The following are the list of features introduced with B-Sharp. For this project, we will only focus on the syntax needed by the language.

- An additional `float` type. There are also associated float literals. These literals are of the form "*d.d*", where *d* is one or more digits.
- An additional operator `as` for explicit casting. The casting operator has precedence between multiplication/division/modulo and exponentiation.

```
1   var: integer = 1;
2   var2: float = var as float;
```

- A `read` statement, which will be used to read in input from `stdin` into a variable. Similar to the `print` statement, it can take a list of expressions and it will infer what to do based on the types within, but they must all be a variable/array index where a value can be stored. For each type in our language, the input format for each will follow the conventions of the C-function `scanf` for the analogus types in C.

```
1   entry_value: integer = 0;
2   read entry_value;
```

- (Bonus) A new looping statement; `loop-while-repeat`.

```
1   bool_val: boolean;
2   ...
3   loop
4       ...
5   while ( bool_val )
6       ...
7   repeat
```

The `while` part is optional, resulting in a `loop-repeat` statement that creates an infinite loop.

```
1   loop
2       ...
3   repeat
```

2 Background Information

For the AST itself, the most important thing to recognize is the implementation of the visitor pattern. A `Visitor` interface in declared with a `visit` method for each of the different types of AST nodes. An object that we want to traverse over the AST nodes with will inherit and implement this interface. Each AST node inherits from a `Visitable` interface. This interface defines the `accept` method, which each nodes implements as follows:

```
void Class::accept(Visitor *v) {
  v->visit(this);
}
```

Because `this` is always a pointer to the exact `Class` object in question, the `Visitor` is always able to invoke the correct version of `visit` for each node. Inside of `visit`, a `Visitor` will perform some set of actions and, if necessary, call the `accept` method on any children of the node. This back and forth enables us to separate any algorithms we want to implement on the AST from the AST itself. It is worth mentioning that this technique does not require us to use method overloading for the `visit` method, but it can be employed to simplify the `Visitor` interface.

LLVM is a collection of compiler and toolchain technologies that will allow us to generate executable code. Clang, a widely-used compiler for

C/C++, is a part of this collection and uses the LLVM libraries we will use in this practicum. Learning how to use the LLVM libraries is a major step in learning how to create production-grade compilers. We have discussed the IR at length in class. The following are reference materials that you may wish to use while creating this project.

- LLVM-12 Docs
- LLVM-12 Programmer's Manual. Details important data structures and layout of the LLVM library.
- LLVM-12 Language Reference Manual. A complete reference to the LLVM IR.
- LLVM Doxygen. This is essentially a complete detail for every object in the LLVM library. It is always updated for the most recent version of LLVM (at the time of writing version 16), so there may be some methods/objects that don't exist in our version of the library. However, it is still very similar, so it's good to refer to when you need to look up the methods of an object.

3 Code Generator Requirements

For our language, our atomic types correspond to the following LLVM types.

- int → `i32`
- float → `float`
- char → `i8`
- bool → `i1`
- string → `i8*`

Using the LLVM library, implement the following actions for each node. Note that in essentially all cases, you will also need to visit a node's children. You should be following the jigsaw tile pattern discussed in the last practicum when designing the visitor action's for each node. The term "buffer" in these actions refers to any method you can use to transfer information across nodes in the AST; a private member variable of the code generator, a private member collection (vector, stack, etc.) of the code generator, public members of the AST nodes, etc.

- ☐ TranslationUnit
  - ☐ Initialize the LLVM objects used generate the IR.
- ☐ GlobalVarDecl
  - ☐ Do nothing, visit variable declaration.
- ☐ FuncDecl
  - ☐ Insert the corresponding function declaration into the LLVM module.
- ☐ FuncDef
  - ☐ Insert the corresponding function declaration into the LLVM module.
  - ☐ Create a block and associate it with the prior function declaration.

- ☐ For each parameter of the function, insert `alloca` instructions for each one. Insert a `store` instruction from the parameter into the newly allocated local registers; those registers will be used in instructions that reference a function parameter.

☐ ArgDecl

- ☐ Create an `Argument` class as a temporary holder for the declaration' name and type.
- ☐ Store that value in a buffer that the higher up FuncDef/FuncDecl node can use.

☐ ForStmt

- ☐ Create a block for the following
  - The check expression.
  - The update expression.
  - The for loop body.
  - An exit point for the loop.
- ☐ Insert instructions and breaks for each part of the for loop into their appropriate blocks, ending on the exit block.

☐ IfStmt

- ☐ Create a block for the following
  - The if body.
  - An exit point for the if statement.
- ☐ Insert instructions and breaks for each part of the if statement into their appropriate blocks, ending on the exit block.

☐ IfElseStmt

- ☐ Create a block for the following
  - The if body.
  - The else body.
  - An exit point for the if statement.
- ☐ Insert instructions and breaks for each part of the if statement into their appropriate blocks, ending on the exit block.

- ☐ PrintStmt
  - ☐ For each expression in the print list, generate a call to `printf` to format each value. You can do this as one call, where you build up the format string based on the types of the arguments used, or have a separate call for each expression.
- ☐ ReadStmt
  - ☐ For each expression in the read list, generate a call to `scanf` to read a value from stdin into said expression. For the LLVM IR, the function name is `__isoc99_scanf` but has the same signature as `printf`. You can do this as one call, where you build up the format string based on the types of the arguments used, or have a separate call for each expression.
- ☐ BlockStmt
  - ☐ Do nothing; visit the statements.
- ☐ DeclStmt
  - ☐ Do nothing; visit the declaration.
- ☐ ExprStmt
  - ☐ Do nothing; visit the expression statement.
- ☐ RetVoidStmt
  - ☐ Insert a `ret` instruction.
- ☐ RetExprStmt
  - ☐ Insert a `ret` instruction that returns the included expression value.
- ☐ LoopWhileStmt
  - ☐ Create a block for the following
    - The first part of the loop.
    - The check expression.

- The second part of the loop.
- An exit point for the loop.

☐ Insert instructions and breaks for each part of the loop into their appropriate blocks, ending on the exit block.

☐ InfiniteLoopStmt

☐ Create a block for the following

- The loop body.
- An exit point for the loop.

☐ Insert instructions and breaks for each part of the loop into their appropriate blocks, ending on the exit block.

☐ NonEmptyExprStmt

☐ Do nothing; visit the expression.

☐ EmptyExprStmt

☐ Do nothing.

☐ VarDecl

☐ If this is a local variable, insert an `alloca` at the top of the function body.

☐ Otherwise, create a global variable with `LLVMModule->getOrInsertGlobal()`.

☐ InitDecl

☐ If this is a local variable, Insert an `alloca` at the top of the function body. Then, insert a `store` instruction for scalar types or a `memcpy` intrinsic function call for array types. Note that for the later, you will need to cast both the variable and initializer values to `i8*` and calculate the size of the RHS (in bytes) to use the intrinsic function.

☐ Otherwise, create a global variable with `LLVMModule->getOrInsertGlobal()`. Then, use `globalVar->setInitializer(initializer)` to set the global variable to the initializer value.

☐ ExprInit

- ☐ Do nothing; visit the expression.

☐ ListInit

- ☐ Create a `ConstantArray` object from the values.
- ☐ Store the resulting value in an buffer to be accessed by a higher initializer or variable declaration.

☐ IdentExpr

- ☐ Lookup the `Value` corresponding to this identifier.
- ☐ If the identifier is being used for its r-value; insert a `load` instruction and buffer the result for higher expressions to use.
- ☐ If the identifier is being used for its l-value; buffer the `Value` itself.

☐ Most Expression Nodes

- ☐ Insert an instruction or intrinsic function call that corresponds to the expression.
- ☐ Store the resulting `Value` in a buffer to be used by higher expressions.
- ☐ For AssignExpr, the `Value` is the right-hand side.
- ☐ For IncExpr and DecExpr, you will have to insert an additional `store` instruction to store the modified value back into the variable.
- ☐ For IndexExpr, the instruction to use is `getelementptr`. If it is being used as an l-value (i.e. on the LHS of an assign), the `Value` to buffer is the result of `getelementptr`. For its r-value, you will need to insert an additonal `load` instruction and buffer that result instead.

☐ ParensExpr

- ☐ Do nothing; visit the expression.

☐ Literal Nodes

- ☐ Generate the corresponding LLVM IR constants for each kind of literal.

- ☐ Type Nodes
  - ☐ If you need to revisit these nodes, generate the LLVM `Type` object corresponding to this type.
  - ☐ Store the `Type` object in the code generator in a buffer that can be used by higher up declarations or types.
- ☐ List nodes
  - ☐ Iterate over each item in the node.

It is **strongly** suggested for development to focus on generating code for one test case at a time (see Submission), rather than simply trying to implement each node down the list. Implement the minimum of what you need to do to achieve it, then build on top of what you have for the next one.

4 Submission

The submission should be a zip/tar file containing at minimum the following files.

- Makefile
- CodeGenerator.cpp/.hpp
- README
- tests/: a directory containing test input
- Any of the other code provided as part of the project that you used

You are free to modify any of the code provided for this project that you aren't required to modify for this project.

The Makefile should be able to perform at minimum the following commands.

- `make run IN=FILE_IN`. Generate the Flex scanner and Bison parser, compile it along with the main program, and execute it giving it a text file `IN` as the arguments. Make sure to make the rule dependent on the executable so that way `make` will build the executable if needed.

- `make clean`. Clean up all files that are generated from running `make`.

For this project, compile with C++17. Some test programs in B-Sharp will be provided, but you are expected to write additionally tests to ensure that the compiler is operating correctly. For this project, you may assume that the programs used to test will be syntactically and semantically correct. Below is the list of programs you are expected to compile, with a breakdown of the maximum points you can earn for doing so. Note that this list is cumulative; you **MUST** implement the first program to be applicable for the maximum for the second, and so on.

- Hello, World! (85 points)
- FizzBuzz: (90 points)
- Ackermann: (95 points)
- A program that contains all of the new features added by B-Sharp (100 points). You are not required to implement any bonus features, unless you are required to do so for some other reason (e.x. taking the class as a graduate). Bonus features will be counted as +10 points.
- One program of your choice that requires you to generate code for at least one feature that isn't covered by the prior tests (declaring an array variable, using a function in the C stdlib by declaring it, etc.) (110 points).

For checking readability and programming style, the grader will be using the `clang-tidy` tool on your provided C++ code (excluding the Bison file). A `.clang-tidy` is included with the provided materials that consist of all the checks that will be done for your code. It would be a good idea to install `clang-tidy`, put the `.clang-tidy` file in the base of your project, and run `clang-tidy` on your code prior to submission to ensure you don't loose points. You can run `clang-tidy` on a file as follows:

```
clang-tidy file --
```

At the top of each source code file you create/modify, you must include a header comment with at minimum the author (you), class, the program filename, and date. Below are some of the recommended fields you may wish to include:

```
       SYNOPSIS : Basic usage information about the
program (i.e. information about how to run the program).
       COMPILATION : Basic compilation information for
program (e.g. "g++ -std=c++11 ...").
       PROGRAMS/APPS USED : Name/describe all programs (e.g., flex)
          ↪ used
       ALGORITHM : Name/describe the algorithm used in the
program as needed
       RUNNING TIME : Running time complexity of the algorithm used
in the program as needed
       BUGS : Known bugs in the program.
       REFERENCE : Acknowledge, refer/cite all sources
and help (including tutors, students, classmates, etc.)
```

Depending on the assignment, additional fields may be required. Fields other than the suggested fields above are allowed, and, in fact, should be added if there is important information about the program that is not covered by the recommended or required fields. This sort of information may be especially useful if the source file must be examined manually during grading. However, only the fields specified for the assignment will be graded.

Documentation comments must be the first comments in the source file. Thus, it may be useful to keep the comments towards the beginning of the file. For an individual field, the field (including both the field label and the portion of the field value that will be graded) should be included in a comment on a single line. The field label name should be given at the beginning of the line (possibly preceded by whitespace or characters on the line indicating that the line is a part of the comment) and followed by a colon (":"). Following the colon should be the value of the field. Whitespace before or after the value of the field will be ignored. For instance, a line for the NAME field in a C++ program might look like the following:

```
* PROGRAM: pa1-000.cpp
```

Additional information about the field may be included in lines below the initial line. This may be useful, for instance, to give credit to sources you used for your assignment. For example, to give credit to the instructor for sample code, the AUTHOR field could read

```
* AUTHOR: Your Name
                Sample code provided by course instructors
```

However, only the first line of the field will be checked, unless stated otherwise in the assignment. Documentation comments may be provided using any comment style for the source language. The following, however, are some suggestions for documentation comment style in different languages:

C, C++ - Single C-style multi-line comment at the beginning of the file or after includes.

Example:

```
/*
 * CLASS: CS441G Fall 2022
 * PROGRAM: my-source.cpp -- this text is optional
 * AUTHOR: Firstname M. Lastname
 * Sample code provided by course instructors
 * EXTRAFIELD: This is some additional information.
 * More than one line is okay.
 */
```

Java - Single C-style multi-line comment at the beginning of the file or after imports.

Example:

```
/*
 * CLASS: CS441G Fall 2022
 * PROGRAM: MySource.java -- this text is optional
 * AUTHOR: Firstname M. Lastname
 * Sample provided by course instructors
 * EXTRAFIELD: This is some additional information.
 * More than one line is okay.
 */
```

Python - Multiple comments at the beginning of the file or after imports.

Example:

```
# CLASS: cs441G Fall 2022
# PROGRAM: my_source.py -- this text is optional
# AUTHOR: Firstname M. Lastname
# Sample code provided by course instructors
# EXTRAFIELD: This is some additional information
# More than one line is okay
```