CMPU 331 · Compilers · Fall 2019

Project - Intermediate Code Generator

Due Friday, December 20, 2019 by 11:59pm

1 Overview

The four project milestones will direct you to design and build a compiler for the DL language, which is a simple subset of the C language. Each assignment will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment will ultimately result in a working compiler phase which can interface with other phases. You will be implementing the projects in Python.

For this assignment, you will write a code generator that outputs LLVM intermediate representation.

You may work either individually or in pairs for this project.

2 Getting Started

You will submit your code through GitHub Classroom, so the first step is to clone a copy of the skeleton files from GitHub. In Moodle, under the topic ”Code Generation”, you'll find a link to ”Generator on GitHub”. Follow that link, select your team (or create a new team), and then click the ”Accept this assignment” button to create your own private copy of the skeleton files. Your copy will be a GitHub repository located at `https://github.com/vassar-cs331-2019b/generator-yourusername` (substitute your GitHub username). From that page, you can copy the URL for cloning by clicking the green button ”Clone or download”, and then paste the URL onto the command-line as an argument to the `git clone` command.

```
$ git clone https://github.com/vassar-cs331-2019b/generator-yourusername.git
```

The clone command will create a directory named `generator-yourusername` on your machine, and in that directory you'll find the following files:

- `LICENSE` - This file contains the open source license for the code. Below the copyright line with my name on it, add another copyright line with your name. (Or, just copy this file from your lexer project repository.)
- `setup.py` - This file contains metadata about the code. Replace my name and email with yours. (Or, just copy this file from your lexer project repository.)
- `runtests.py` - This script is a simple test runner. You can call it with `python3 runtests.py` to run all the tests.
- `generator.py` - This script is a command-line interface for the code generator, which you might find useful while you're developing it. You can run it it on the command-line, passing it one argument for the name of a DL file to parse, like `python3 generator.py filename.dl`. The script will read the file, parse it, run your generator library, and write out the LLVM code returned by the generator to a file named `filename.ll`.

- `sly/lex.py` and `sly/yacc.py` - These files are libraries used in the lexer and parser. You won't make any changes to these files, they are only included in the skeleton so you don't need to install them.
- `dl/ast.py` - This library contains the class definitions for the AST node types you will use in your code generator.
- `dl/lexer.py` and `tests/test_lex.py` - These are a complete implementation and tests for the lexer stage of the compiler. You can (optionally) replace them with your own implementation from the lexer project submission, together with any tests you added.
- `dl/parser.py` and `tests/test_parser.py` - These are a complete implementation and tests for the parser stage of the compiler. You can (optionally) replace them with your own implementation from the parser project submission, together with any tests you added.
- `dl/generator.py` - This file contains a skeleton for a DL code generator. There are comments indicating where you need to fill in code, but you are welcome to make any modifications to the skeleton. The skeleton is functional, and will pass two tests, but it doesn't do much.
- `tests/test_generator.py` - This file contains tests for the code generator, including at least one test for each type AST node, and one test of a longer code example. The provided tests are not exaustive, you may want to add more as you work. You won't be graded on the tests you add, but testing more thoroughly can increase your confidence that your semantic analysis is working as it should.
- `tests/simple.dl` - This file is a code example in the DL language, which is used by one of the tests.

The CS lab workstations have all the software you need installed already. If you want to work on your own laptop, you might need to install LLVM.

PyCharm (`https://www.jetbrains.com/pycharm/`) is one IDE for working with Python, and is already installed on the CS lab workstations (though, apparently only with a trial license). But, you can use any IDE or text editor you prefer.

3 Code Generator Implementation

In this assignment you will write a code generator for DL. The assignment makes use of two resources: a tree traversal library and a collection of class definitions for abstract syntax tree (AST) nodes. The output of your code generator will be a string containing a complete program, with the same functionality as the original DL source program, but translated into LLVM intermediate representation.

You might find it helpful to refer to these sources as you work:

- Appendix A - "The DL Language" starting on page 247 of the textbook "A Practical Approach to Compiler Construction" by Des Watson
- Table 5.1 - "Parse tree nodes" on page 120 of the textbook "A Practical Approach to Compiler Construction" by Des Watson
- A small working example of a code generator for the Calc language, from class: `https://github.com/vassar-cs331-2019b/calc-examples/blob/master/calc/generator.py`

3.1 Visitor Methods for AST Nodes

The skeleton includes class definitions for a collection of AST node types that you used in the parser and semantic analyzer, and will use again in the code generator. These are inspired by the node types defined in the textbook (on page 120), but not exactly the same. A description of the AST node types was provided in the instructions for the parser project assignment, so you will likely find it useful to refer back to that description.

For each AST node type, you will need to write a single method in the code generator, that knows how to generate the LLVM code for that specific type of AST node. For example, for `BinOp` AST nodes, you will need to implement a method in the code generator named `visit_BinOp`, which receives a single argument named `node`, and generates the appropriate text for the LLVM opcode that corresponds to whatever operator the `BinOp` node contains (that is, addition, subtraction, multiplication, or division).

The skeleton code generator includes a visitor method for every AST node type, but most of them haven't been implemented yet. You will need to delete the single line `pass` from the body of the method (which just tells Python "do nothing for this method, but also don't complain that the method isn't implemented yet"), and replace it with your own code. The following table lists the visitor methods included with the skeleton, a description of what they should do, and the value each method should return.

3.1.1 visit_Integer

Already implemented.

What it does: Not much. Since LLVM works with integers directly, this simply uses the integer value from `node.value`.

Note, this method doesn't directly add any generated code, it only returns a value to be used by other visit methods.

Returns: The integer value.

3.1.2 visit_Variable

Partially implemented.

What it does: Checks if the `node.symbol` (from the semantic analyzer) was a `VariableSymbol` or an `ArgumentSymbol`, and calls a different helper method for each. Arguments are accessed using a temporary register with the same name as the DL argument, so the helper method `access_Argument` simply returns `node.name` with the string "%" added in front.

What you need to implement: For `VariableSymbol`, you will need to implement the helper method `access_Variable`, which will write out a single line of LLVM code, something like this:

```
%tmp.5 = load i32, i32* %x
```

In this example, `%tmp.5` is a temporary register, and `%x` is a register which holds (a pointer to) the DL variable named `x`.

The generator class includes a helper method for generating unique temporary register names, which you can call in Python:

```
temp_name = self.new_temporary()
```

This exact line will be the first line of your access_Variable helper method. The second line will simply create a variable called local_name, by appending "%" onto the front of the DL variable name, to make a LLVM local name.

```
    local_name = "%" + node.name
```

The rest of the helper method will be a few lines of code that you will use over and over again throughout the generator:

```
        template = """
    ; some LLVM code here
        """
        output_code = template % (temp_name, local_name)
        self.add_code(output_code)
        return temp_name
```

First it makes a template variable, which is just literal lines of LLVM code, but with %s in place of any text that needs to be substituted. In this helper method, the LLVM code will be just the load instruction above, but replacing both %tmp.5 and %x with %s so we can substitute generated register names. The next line processes the template, substituting each %s with a string value from the list, and then stores the result in the variable output_code. The next line calls the method add_code, telling the generator to save the generated lines of code. And, the last line returns the name of the temporary register, so that other instructions can use the value that was fetched from the variable.

Returns: A string containing the name of the temporary register that holds the value fetched from the variable or argument.

3.1.3 visit_ArrayIndex

Already implemented.

What it does: Accessing a value at an array index is similar to accessing a value in a named variable. What makes the method more complex is that arrays contain multiple elements, so the LLVM code needs to first fetch a pointer to the correct element in the array (using the index), and then fetch the value from that element. To do this, the method generates two lines of LLVM code, the first is a getelementptr instruction, and the second is a load instruction (just like we generated for variable accesses).

```
    %tmp.7 = getelementptr [10 x i32], [10 x i32]* %y, i32 0, i32 5
    %tmp.8 = load i32, i32* %tmp.7
```

In this example, %tmp.7 is a temporary register holding a pointer to one element of the array, [10 x i32] is the type of the array (the array holds 10 elements, and each element is an i32 integer), %y is a register which holds (a pointer to) the DL array named y, and 5 is the index of the specific element we're accessing from the array. Finally, %tmp.8 is a temporary register holding the value fetched from the element at index 5 of the array.

Returns: A string containing the name of the temporary register that holds the value fetched from the array at the index position.

3.1.4 visit_BinOp

Not implemented yet.

What you need to implement: Take a look at the `visit_BinOp` method in the Calc example, it is basically identical to what you need here. This method will generate a single LLVM instruction.

```
%tmp.1 = add i32 %tmp.2, %tmp.3
```

This example is an `add` instruction, `%tmp.2` and `%tmp.3` are temporary registers holding the values to be added, and `%tmp.1` is a temporary register holding the result of the addition. In your method, you'll need to create a new temporary register to hold the result of the operation, by calling the `new_temporary` method. Then, you'll recursively visit the left and right children of the `BinOp` node, so they can generate the neccessary LLVM code to access the two arguments to the operaton.

```
left_reg = self.visit(node.left)
right_reg = self.visit(node.right)
```

Then look at `node.op` to decide which LLVM instruction you need to generate for this `BinOp`. The possible values for `node.op` are `PLUSOP`, `MINUSOP`, `MULTIPLYOP`, and `DIVIDEOP`, and the corresponding LLVM instruction will be one of these four:

```
%result = add i32 %arg1, %arg2
%result = sub i32 %arg1, %arg2
%result = mul i32 %arg1, %arg2
%result = udiv i32 %arg1, %arg2
```

When you make your `template` string, use `%s` in place of all three temporary register names, and also in place of the operator name `add`. Finally, process the template substitutions, save the generated code by calling the `add_code` method, and return the temporary register name for the result of the operation, so that other instructions can use the result.

Returns: A string containing the name of the temporary register that holds the result of the operation.

3.1.5 visit_RelOp

Already implemented.

What it does: Generating code for a relational operator is similar to generating code for a binary operator. The possible values for `node.op` are `EQOP`, `NEOP`, `LTOP`, `LEOP`, `GTOP`, and `GEOP`, and the corresponding LLVM instructions look like:

```
%tmp.1 = icmp eq i32 %tmp.2, %tmp.3
```

This example is an `icmp eq` instruction (compare equal), `%tmp.2` and `%tmp.3` are temporary registers holding the values to be compared, and `%tmp.1` is a temporary register holding the result of the comparison.

Returns: A string containing the name of the temporary register that holds the result of the operation.

3.1.6 visit_Assign

Partially implemented.

What it does: In DL, assignment is the only way to store a value in a variable or array element. The left side of the assignment operation needs to be handled differently than accessing variables or array elements, specifically it needs to generate LLVM `store` instructions, instead of `load` instructions.

The `visit_Assign` method checks if `node.left` is a `Variable` or an `ArrayIndex`, and calls a different helper method for each. The `assign_ArrayIndex` helper method generates the LLVM instructions to assign a value to an array element, and the `assign_Variable` helper method is left for you to implement.

What you need to implement: You will need to fill in the `assign_Variable` helper method, to generate the LLVM code to assign a value to a variable. Take a look at the `assign_ArrayIndex` helper method, since it is similar to what you need here (though, assigning to an array element is more complex than assigning to a variable). The `assign_Variable` method will generate a single LLVM instruction:

```
    store i32 %tmp.4, i32* %x
```

In this example, `%tmp.4` is a temporary register containing the value to store in the variable, and `%x` is a register which holds (a pointer to) the DL variable named `x`.

The name of the temporary register containing the value to store has already been provided for you, in the argument named `right_reg`. You will need to create the local name of the DL variable, which will just be `node.name` with "%" added in front. Finally, create the `template` string, process the template substitutions, and save the generated code by calling the `add_code` method.

Returns: nothing

3.1.7 visit_Print

Already implemented.

What it does: This method generates the LLVM instructions to print a value. In LLVM, output is handled by built-in functions, so the single line of generated LLVM code is a call to the `printf` function.

Returns: nothing

3.1.8 visit_Read

Already implemented.

What it does: This method generates the LLVM instructions to read a value from "standard input" (typically this means reading from a prompt on the command-line, but for testing purposes you can trick it into reading from some other source). In LLVM, input is handled by built-in functions, so the single line of generated LLVM code is a call to the `scanf` function.

Returns: nothing

3.1.9 visit_Block

Already implemented.

What it does: This method iterates through `node.statements`, and visits each statement AST node. The method doesn't directly generate any code, all code generation for statements is done by the visit methods for the statement AST nodes.

Returns: nothing

3.1.10 visit_If

Not implemented yet.

What you need to implement: You will need to generate the labels and branch instructions to implement the control flow for an `if` statement. Take a look at the `visit_While` method, which is already implemented, since it's very similar to what you need to do for `visit_If`. Just remember that the control flow for `if` doesn't repeat, so the labels and branch instructions will be a little different than `while`. Ultimately, the code you generate will look something like this:

```
  ; compare instruction generated by visiting node.condition
  br i1 %tmp.2, label %if.true.7, label %if.false.8
if.true.7:
  ; true body generated by visiting node.body_true
  br label %if.end.9
if.false.8:
  ; else body generated by visiting node.body_else
  br label %if.end.9
if.end.9:
```

In this example, `if.true.7`, `if.false.8`, and `if.end.9` are all labels, generated by calling a Python helper method in the generator, which creates a new unique label:

```
true_label = self.new_label("if.true")
```

(Note, you don't have to name your labels exactly like this, they really only need to be some unique string name.)

The `%tmp.2` temporary register in the example holds the result of the comparison operation. The string name of this temporary register is returned by the call to visit `node.condition`:

```
cond_reg = self.visit(node.condition)
```

You will also need to visit `node.body_true` and `node.body_else` to generate the instructions for the code inside those two blocks. One important point to keep in mind is that the visits to the condition and body AST nodes will *immediately* add their generated code, so you need to break up the lines of code generated directly in the `visit_If` method into 3 separate template strings and process each one in the right order. The `visit_While` method is a good example of how this is done.

3.1.11 visit_While

Already implemented.

What it does: This method generates the labels and branch instructions to implement the control flow for a `while` loop. The code it generates looks something like this:

```
while.loop.2:
  ; compare instruction generated by visiting node.condition
  br i1 %tmp.6, label %while.body.3, label %while.end.4
while.body.3:
  ; body generated by visiting node.body
  br label %while.end.4
while.end.4:
```

In this example, `while.loop.2`, `while.body.3`, and `while.end.4` are all labels, generated by calling a Python helper method in the generator, which creates a new unique label:

```
    loop_label = self.new_label("while.loop")
```

The temporary register `%tmp.6` in the example holds the result of the comparison operation. The string name of this temporary register is returned by the call to visit `node.condition`.

If you take a look at how `visit_While` is implemented, you'll see that it generates a few lines of LLVM code before it visits `node.condition`, then generates a few more lines of LLVM code between visiting `node.condition` and `node.body`, and finally generates a few lines of LLVM code after it visits `node.body`. It is necessary to break up the code generation for control flow this way because the visits to the condition and body AST nodes *immediately* add their generated code.

3.1.12 visit_Declarations

Already implemented.

What it does: This method iterates through `node.declarations`, and visits each `FunctionDeclaration` AST node (to generate the LLVM code for function declarations), but saves the `VariableDeclarations` AST nodes to process later. The `visit_Declarations` method doesn't directly generate any code, all code generation for declarations is done by the `visit_FunctionDeclaration` and `visit_VariableDeclarations` methods.

Returns: A list of `VariableDeclarations` AST nodes extracted from the declarations list, so `visit_Program` can generate the code for variable declarations in the body of the `main` function.

3.1.13 visit_VariableDeclarations

Already implemented.

What it does: This method iterates through `node.variables`, checks if each declaration is a `Variable` or an `ArrayIndex`, and calls a different helper method for each. The `declare_Variable` helper method generates the LLVM instructions to allocate memory for an integer variable, and then initializes the variable value to 0.

```
    %x = alloca i32
    store i32 0, i32* %x
```

In this example, `%x` is the register holding (a pointer to) the DL variable named `x`, and the variable is allocated enough space to hold a single integer of type `i32`.

The `declare_ArrayIndex` helper method generates the LLVM instructions to allocate memory for an array.

```
    %y = alloca [10 x i32]
```

In this example, `%y` is the register holding (a pointer to) the DL array named `y`, and the array is allocated enough space to hold 10 integers of type `i32`.

Returns: nothing

3.1.14 visit_FunctionDeclaration

Already implemented.

What it does: This method generates the LLVM code for a function declaration. The code it generates looks something like this:

```
define i32 @alpha(i32 %b, i32 %c) {
    ; variable declarations generated by visiting node.vars

    ; body generated by visiting node.body
}
```

In this example, `@alpha` is the name of the function, `%b` is the first argument to the function, and `%c` is the second argument.

The name of the function is just `node.name` with "@" added in front, since functions are declared as globals in LLVM.

The list of arguments is generated by visiting `node.args`, and will be an empty string if there are no arguments. The Python code for handling arguments checks if `node.args` is defined, and only visits the `Arguments` AST node if it exists.

```
    args_string = ""
    if node.args:
        args_string = self.visit(node.args)
```

The list of variables to declare for the function is also optional, so the Python code checks if `node.vars` is defined before visiting the `VariableDeclarations` AST node.

If you take a look at how `visit_FunctionDeclaration` is implemented, you'll see that it generates the beginning of the function declaration before it visits the variable declarations and `node.body`, and finally generates the closing bracket for the function declaration after it visits `node.body`. It is necessary to break up the code generation for the function declaration this way because the visits to the variable declaration and body AST nodes *immediately* add their generated code.

Returns: nothing

3.1.15 visit_FunctionCall

Not implemented yet.

What you need to implement: You will need to implement the code to call a function, which means generating a single line of LLVM code that looks something like this:

```
    %tmp.7 = call i32 @alpha(i32 %tmp.8, i32 %tmp.9)
```

In this example, `%tmp.7` is a temporary register holding the result of the function call, `@alpha` is the name of the function, `%tmp.8` is a temporary register holding the first value passed as an argument to the function, and `%tmp.9` is a temporary register holding the second value passed as an argument to the function.

The name of the function is just `node.name` with "@" added in front, since functions are declared as globals in LLVM.

The list of arguments is generated by visiting `node.args`, and will be an empty string if there are no arguments. Take a look at how the `visit_FunctionCall` method checked whether `node.args` exists before visiting it, you'll need to do exactly the same in `visit_FunctionDefinition`.

When you make your `template` string, use `%s` in place of all three temporary register names, and also in place of the function name `@alpha`. Finally, process the template substitutions, save the generated code by calling the `add_code` method, and return the temporary register name for the result of the function call, so that other instructions can use the result.

Returns: A string containing the name of the temporary register that holds the result of the function call.

3.1.16 visit_Arguments

Already implemented.

What it does: This method generates a string sequence of arguments, with types for each argument. The string it generates looks something like this:

```
    i32 %b, i32 %c
```

Note, this method doesn't directly add any generated code, it only returns a string to be used by other visit methods.

Returns: a string sequence of arguments for a function definiton or function call.

3.1.17 visit_Return

Already implemented.

What it does: This method generates the LLVM instruction to return from a function, which looks like:

```
    ret i32 %tmp.9
```

Returns: nothing

3.1.18 visit_Program

Already implemented.

What it does: This method generates the LLVM code for the beginning and end of the entire program.

The method first visits `node.declarations` to generate any function declarations, and capture a list of variable declarations to process later. Then the method generates the LLVM code necessary to import the `@printf` and `@scanf` functions, which are used by `visit_Print` and `visit_Read`. Finally, the method generates the LLVM code for the `@main` function, which looks something like this:

```
define i32 @main() {
    ; variable declarations generated by visiting each element in var_decs

    ; body generated by visiting node.body
    ret i32 0
}
```

If you take a look at how `visit_Program` is implemented, you'll see that it generates the beginning of the `@main` function declaration before it visits the variable declarations and `node.body`, and finally generates the `ret` instruction and closing bracket for the function declaration after it visits `node.body`. It is necessary to break up the code generation for the function declaration this way because the visits to the variable declaration and body AST nodes *immediately* add their generated code.

Returns: nothing

3.2 Skipped Tests

In the skeleton, many of the tests are marked as `skip`, because we don't expect these tests to pass yet. Marking the tests as skipped makes them display a clean message when you run `runtests.py`, like this:

```
... skipped 'Arrays not implemented yet'
```

When you start working on the productions related to a particular test, first delete the line that marks that test to be skipped, from the test file `tests/test_generator.py`. The line will look like this:

```
@unittest.skip("Arrays not implemented yet")
```

After you delete the skip marker, the test will fail, and display an ugly error message. The error message will give you some clues about what's missing from your code generator. Once you've implemented all the productions needed to pass a particular test, the status report at the end of the test run will change from something like:

```
FAILED (errors=1, skipped=4)
```

To something like:

```
OK (skipped=4)
```

Passing tests is how you know you've implemented your code generator correctly. When all the tests are passing with no skips, you're done.

In general, you'll find it easier to start working on the tests towards the top of the `tests/test_generator.py` file, since they test features that are easier to implement. The tests grow more complex towards the end of the file, and the very last test parses the DL code example in `tests/simple.dl`.

4 Submitting

To submit your work, `git add` any files you changed, then `git commit` them with a sensible commit message about what you changed, and finally `git push` the changes back to GitHub.

```
$ git add dl/generator.py
$ git commit -m "Implemented code generation for arrays"
$ git push origin master
```

It's a good idea to commit your work frequently as you go, so you have a backup if anything goes wrong. Only the last version you submit before the deadline will be graded.