vcalc

cmput415

May 17, 2023

CONTENTS

This assignment expands the simple calculator, *SCalc*, to build a vector calculator called *VCalc*. For *VCalc* you need only build an *LLVM IR* back end. None of the assembly back ends that you built for *SCalc* need to be supported for *VCalc*. An interpreter is not necessary but can be a good way to ensure that your grammar works as expected.

VCalc is a superset of *SCalc*. **All operations supported by SCalc must be fully also supported by VCalc. All valid SCalc programs must run in VCalc without modification.** (The only exceptions are variable names in a *SCalc* that are now reserved Keywords.) *VCalc* has the additional features discussed in subsequent section.

CHAPTER

ONE

CHANGE LOG

- **2020/09/01 3:00PM**
  - Initial release for Fall 2020

CHAPTER

TWO

KEYWORDS

The following keywords are now also reserved in *VCalc*:

- `in`
- `vector`

CHAPTER

THREE

VECTORS

VCalc has a new type, `vector`, that is a vector of integer values. Vectors are restricted to the length that can be represented by the *largest possible index*. Indices are integers and integers are signed 32 bit integers. Because the largest possible integer is $2^{31} - 1$ or 2147483647, a vector can have a length in the range $[0, 2^{31}]$.

Assertion: All vectors will have length l such that $0 \leq l \leq 2^{32}$. (*vector-length*)

There is no way to specify a vector literal, they must be created through ranges, generators, filters, or index expressions with a vector index.

The only way to create an empty vector is through the use of a filter whose predicate is evaluated to false at each index of the domain or a range whose first bound is greater than the second bound. Once you have an empty vector, other operations may also produce empty vectors. That is, binary operations between an empty vector and another empty vector or a scalar, indexing by an empty vector, or using an empty vector as a generator or filter domain will also result in an empty result vectors.

CHAPTER

FOUR

RANGE

In *VCalc* the operator `..` is used to generate a vector holding a range of integers. This operator must have an expression resulting in an integer on both sides of it. These integers mark the *inclusive* upper and lower bounds of the range.

For example:

```
print(1..10);
print((10-8)..(9+2));
```

prints the following:

```
[1 2 3 4 5 6 7 8 9 10]
[2 3 4 5 6 7 8 9 10 11]
```

The number of integers in a range may not be known at compile time when the integer expressions use variables. In another example, assuming at runtime that `i` is computed as -4:

```
print(i..5);
```

prints the following:

```
[-4 -3 -2 -1 0 1 2 3 4 5]
```

Therefore, it is *valid* to have bounds that will produce an empty vector because the difference between them is negative. For example:

```
int i = 3;
int j = 0;
print(i..j);
```

prints the following:

```
[]
```

CHAPTER

FIVE

GENERATORS

A generator is another way to create a vector in *VCalc*. Generators work the same as they did in the generator assignment and have the following form:

```
[<domain variable> in <domain> | <expression>]
```

The identifier is referred to as the domain variable, the vector is the domain or domain vector, and the expression is the right-hand-side expression. The domain variable is an integer typed variable defined only in the scope of the generator.

The domain may be any vector-valued expression which includes identifiers (that are vector typed), ranges, generators, filters, and index expressions with a vector index. The expression must evaluate to an integer. This means that if the result of the expression is a boolean it will be implicitly promoted to an integer, but a vector result is an *error*.

Generators are identical to list comprehensions from other languages. For instance, to generate a vector of the first 100 perfect squares, one may write the following generator:

```
vector sqrs = [i in 1..100 | i * i];
```

The expression on the right yields the value for a single element of the generated vector, which corresponds to the element `i` of the domain vector.

The right-hand-side expression does not need to depend upon the domain variable. For instance:

```
print([i in 1..10 | 0]);
```

prints the following:

```
[0 0 0 0 0 0 0 0 0 0]
```

As another example, the following generator produces the square value of all positive, even integers up to 20.

```
print([i in [ j in 1..10 | j * 2] | i * i]);
```

prints the following:

```
[4 16 36 64 100 144 196 256 324 400]
```

CHAPTER

SIX

FILTERS

A filter has similar syntax to a generator, but instead has a & instead of a | as shown here:

```
[<domain variable> in <domain> & <predicate>]
```

The identifier and vector are still called the domain variable and domain vector, however, the right-hand-side expression is now called the *predicate*. The domain variable is an integer typed variable defined only in the scope of the generator.

As in a generator, the domain may be any vector-valued expression which includes identifiers (that are vector typed), ranges, generators, filters, and index expressions with avector index. The expression must evaluate to a boolean. This means that if the result of the expression is an integer it will be implicitly demoted to a boolean, but a vector result is an *error*.

A filter will create a new vector containing only the elements of the domain where the predicate evaluates to a true value. The domain values that satisfy the predicate are appended to the result vector in their original order. For instance, to select all of values greater than 5 in a vector you might do:

```
print([i in 1..10 & 5 < i ]);
```

prints the following:

```
[6 7 8 9 10]
```

CHAPTER

SEVEN

EXPRESSIONS

7.1 Operators

Because we've added a new binary operator, we need to update our precedence table. Operators without a horizontal line dividing them have equal precedence. For example, addition and subtraction have an equal level of precedence.

Class	Operation	Symbol	Usage	Associativity
Vector	index	[]	`expr[expr]`	left
	range	..	`expr .. expr`	left
Arithmetic	multiplication	*	`expr * expr`	left
	division	/	`expr / expr`	left
	addition	+	`expr + expr`	left
	subtraction	-	`expr - expr`	left
Comparison	less than	<	`expr < expr`	left
	greater than	>	`expr > expr`	left
	is equal	==	`expr == expr`	left
	is not equal	!=	`expr != expr`	left

7.2 Binary Operations on Vectors

Binary oprations between vectors require extra specification.

1. All binary operations are performed element-wise. This means that the specified operation is applied to elements at the same index in both vectors with the result then being placed into the same index in the result vector. For example:

```
vector v = 1..5 + 1..5;
print(v);
```

   prints the following:

```
[2 4 6 8 10]
```

2. Binary operations *can* be performed between vectors of different sizes. For most operations the smaller vector is padded with zeroes to match the larger vectors size and then the operation is applied. For example:

```
print(6..10 + 1..3);
print(1..3 + 6..10);
```

prints the following:

```
[7 9 11 9 10]
[7 9 11 9 10]
```

The only exception is when the smaller vector is a *divisor*. A divisor must be extended with ones to prevent division by zero errors. For example:

```
print(6..10 / 1..3);
print(6..8 / 1..5);
```

prints the following:

```
[6 3 2 9 10]
[6 3 2 0 0]
```

3. Boolean operators between vectors are still applied element-wise, but the result will be converted to an integer as decribed in *SCalc* before being saved into the result. For example:

```
vector a = [i in 0..5 | i / 2];
vector b = [i in 1..6 | i / 2];
print(a);
print(b);
print(a == b);
```

prints the following:

```
[0 0 1 1 2 2]
[0 1 1 2 2 3]
[1 0 1 0 1 0]
```

7.3 Integer to Vector Promotion

Integers used in expressions with vectors will be promoted to vectors. The scalar value will be copied into *each index* of a new vector the same size as the other operand before applying the operator in the regular vector fashion. For example:

```
print(1..5 + 5);
print(2 * 3..6);
print(5 < 3..7);
```

prints the following:

```
[6 7 8 9 10]
[6 8 10 12]
[0 0 0 1 1]
```

A more complicated example:

```
print(5 + [i in 1..3 | 0] + 1..5);
```

prints the following:

```
[6 7 8 4 5]
```

One might expect:

```
[6 7 8 9 10]
```

but recall that addition is left associative. Therefore the order of the operations in the print statement is:

```
print((5 + [i in 1..3 | 0]) + 1..5);
```

The five will be promoted to a vector of length three to match the generator, resulting in `[5 5 5]`, which will be added to the generator for no change. Then it will be *extended* to match the length five range as `[5 5 5 0 0]` before being added to create the final result of `[6 7 8 4 5]`.

7.4 Vector Indexing

Vectors can be indexed by a scalar to produce the integer value at a specified index. Vectors in *VCalc* are *zero indexed*. As well, indexing outside of the bounds of a vector (e.g. `v[i]` where $0 <= |v| < l$ and $i < 0$ or $i >= l$) is *not an error*. An index out of bounds *always returns zero*.

Index domains must be vectors:

- Domain can be an identifier for a vector.
- Domain can be the result of a range, generator, filter, or another index expression with a vector index (see below).
- Domain cannot be an integer. For example, this is invalid:

  ```
  print(1[1]);
  ```

Examples of valid index expressions:

```
vector v = 1..5;
print(v[0 - 1]);
print(v[2]);
print(v[5]);
print([i in v | i * 2][3]);
print([i in v & i > 2][0]);
```

prints the following:

```
0
3
0
8
3
```

Domain vectors can also be indexed by a domain indexing vector to produce a new result vector. This new vector will contain the values of the domain vector as if each of the values in the domain indexing vector had individually indexed the domain vector and then been appended to the result vector. For example:

```
vector v = 1..7;
vector i = 2..4;
print(v[i]);
print(v[i * 2]);
```

prints the following:

```
[3 4 5]
[5 7 0]
```

Each value in `i` serves as an index into `v`. Each value indexed from v is appended to the result and then printed.

CHAPTER

EIGHT

STATEMENTS

8.1 Declaration

VCalc adds vectors as an assignable type. To declare a vector variable, you declare a variable as you would an integer, but replace `int` with `vector`. Vectors may be initialized with any expression that returns a vector. For example, assigning a range to a vector `v`:

```
vector v = 1..10;
print(v);
```

prints the following:

```
[1 2 3 4 5 6 7 8 9 10]
```

8.2 Assignment

There are a few new important points when dealing with assignments.

1. The size of a vector may change while the program is executing if a vector variable is assigned another value. For instance, the following sequence of statements *is* valid:

   ```
   vector v = 1..10;
   v = 1..1000;
   ```

   You will have to allocate more memory to store the result of the assignment.

2. The type of the expression of the assignment must match the destination variable's type. This is apparent for trying to assign vectors to a scalar. In the case of scalars being assigned to vectors, one might expect that we can use our extension policy to copy our scalar to every index of a newly created vector but the question is, how large is the new vector. Because that is indeterminable, this is not allowed. For example, the following sequence of statements *is not* valid:

   ```
   int i = 1..3;
   vector v = 1;
   ```

3. Many languages allow you to assign to vector indices, *VCalc does not*. For example, the following sequence of statments *is not* valid:

   ```
   ;
         vector v = 1..3;
         v[0] = 99;
   ```

8.3 Conditional

Conditional conditions must evaluate to booleans, this means that vectors are not a valid condition. Remember, however, that integers can be implicitly downcast to booleans.

8.4 Loops

Loop conditions must evaluate to booleans, this means that vectors are not a valid condition. Remember, however, that integers can be implicitly downcast to booleans.

CHAPTER

NINE

TYPE CHECKING AND ASTS

With the addition of another type that can be mixed in, type checking becomes a necessity in *Vcalc*. This means ensuring that vectors and scalars are where they belong. Most expressions allow the interchange of vectors and scalars, but there's a few cases where it is necessary to have one or the other.

Note that these rules are already in their respective sections, this list just serves to bring further attention to where type checking is important.

- Ranges: lower and upper bounds must be integers.
- Conditional Statements: must be booleans (remember that integers can be implicitly downcast to booleans).
- Domains: in a domain expression (generator, filter, index) the domain must be a vector.
- Generators: the expression must be an integer (remember that booleans can be implicitly upcast integers).
- Filters: the predicate must be a boolean (remember that integers can be implicitly downcast to booleans).

A good way to handle this now and plan ahead for *gazprea* is to start building an abstract syntax tree (AST) from your parse tree as well as a class that knows how to traverse it.

An AST will allow you to attach information in ways that make sense to you. It allows you to strip away unecessary tokens from the parse tree as well as allowing you to convert the parse tree into a new form that also makes sense to you. This could mean normalising parts of the tree to reduce code generation efforts, attaching information through fields or entirely new nodes, and more.

You can find more advice in the AST Tips and Tricks section.

CHAPTER

TEN

SCOPING

Loops and conditionals are scoped in *VCalc*, unlike *SCalc*. As well, generators and filters both have internal scopes for their domain variable.

A reference to a variable will resolve to the *definition* in the innermost possible scope. This matches the scoping rules found in *C*. For example:

```
int i = 1;
print(i);

if (1 == 1)
  int i = 3;
  print(i);

  i = i * 2;
  print(i);
fi;

print(i);

loop (i < 20)
  print(i);
  i = i + 10;
pool;

print(i);
```

prints the following:

```
1
3
6
1
1
11
21
```

Generator and filter scopes only exist during the evaluation of the expression or predicate. The scope will only contain the domain variable.

Be careful in what order you evaluate things. For example:

```
int i = 0;
if (1 == 1)
  int i = i + 1;
  print(i);
fi;

vector v = 0..3;
print([i in i..3 | i]);
print([v in v & v < 2]);

int j = 5;
print([i in v | i * j]);
```

prints the following:

```
1
[0 1 2 3]
[0 1]
[0 5 10 15]
```

If you define a variable in a scope before evaluating the expression, you may mis-resolve a value. If you enter the new scope in your filter or generator before resolving the domain, you may mis-resolve the domain.

CHAPTER

ELEVEN

INPUT

The input processed by your compiler will be in a file specified on the command line. Your compiler will be invoked with the following command:

```
vcalc <input_file_path> <output_file_path>
```

You should open the file `input_file_path` and parse it. The input file will be a valid vcalc file.

CHAPTER

TWELVE

OUTPUT

Output is to be written to a file specified on the command line. Your compiler will be invoked with the following command:

```
vcalc <input_file_path> <output_file_path>
```

You should open the file `output_file_path` and write to it. The output file should be overwritten if it already exists.

Output content is standardized to ensure everyone can pass everyone's tests. Follow these specifications:

- There *must* be a new line after each `print` statement's printed value.
- There *must not* be any trailing space after printed value and before the newline.
- There *must* be an empty line at the end of your output.
- There *must not* be spaces between the first and last number and the accompanying brackets in a vector.
- There *must* be spaces between the numbers in a vector.
- There *must not* be anything except spaces between the numbers in a vector.

Clarification: Empty input should result in empty output. (*empty-input*)
Clarification: Empty vectors print only brackets. (*empty-vector*)
Clarification: A vector with one value is only the brackets and the value. (*single-value-vector*)

CHAPTER

THIRTEEN

ASSERTIONS

ALL input test cases will be valid. It can be a good idea to do error checking for your own testing and debugging, but it is *not necessary*. If you encounter what you think is undefined behaviour or think something is ambiguous then *do* make a forum post about it to clarify.

What does it mean to be valid input? The input must adhere to the specification. The rules below give more in-depth explanation of specification particulars.

1. **vector-length**:

   All vectors will have length l such that $0 \leq l \leq 2^{32}$. Trying to create an index greater than $2^{32} - 1$ will cause overflow and result in a negative number. Indexing with a negative number returns 0. Therefore, vector locations greater than $2^{32} - 1$ would be inaccessible. For example, the following tests would be considered invalid:

   ```
   print((0-1)..2147483647);
   ```

   But the following test is valid because the vector length is still within range:

   ```
   print((0-2)..2147483645);
   ```

CHAPTER

FOURTEEN

CLARIFICATIONS

These clarifications are meant to add more information to the specification without cluttering it.

1. **empty-input**:

   Empty input should result in empty output. This is in keeping with all of the output rules defined. There are no `print` statements so there would be no numbers, newlines or output of any kind. All that you are left with is a single empty line, which matches "*should* be an empty line at the end of your output".

2. **empty-vector**:

   Empty vectors print only brackets. This is in keeping with all of the output rules defined. There are no integers to print between the brackets, so there is no values nor spaces to print. For example:

   ```
   []
   ```

3. **single-value-vector**:

   A vector with one value is only the brackets and the value. This is in keeping with all of the output rules defined. There is only one integer. There is no space between the first bracket and the integer and no space between the integer and the second bracket. For example:

   ```
   [1]
   ```

CHAPTER

FIFTEEN

DELIVERABLES

Your submission will be **the latest commit before the deadline** to your github repository. Your submission will be automatically snapshotted by the GitHub classroom at the submission time.

Do no submit your binaries, they will be built just before being tested. The solutions will be built using the lab machines. You should make sure your solution builds in a lab environment prior to the submission time.

Your tests also should be committed to your github repository. We will pull both your submission and tests directly from your repository.

You do not need to submit anything on eclass or anywhere else.

CHAPTER

SIXTEEN

TIPS AND HINTS

1. The learning curve for *LLVM* is not trivial. Thus **START EARLY**. There will be a lot of things to learn. If you can't figure out how to do something don't be afraid to ask. Someone else will know or someone else will also want to know.

   As well, you should check the *LLVM* Tips and Hints section for a good starting place.

2. You should definitely consider making an AST in this assignment. While it's not strictly necessary it can be a great help, and you'll be much better equipped moving into *gazprea*.

   You should check the AST Tips and Hints section for a good starting place here as well.

3. Write tests **BEFORE** you implement the things they will test.

4. Reuse your tests from *SCalc*.

5. There are times when you do not want to visit the tree in order (e.g. filters/generators), this makes using a listener difficult. We suggest using a visitor.

6. Try to plan ahead of time to avoid having to rewrite things. Occasionally you will need to rewrite portions of your code if you find a new complication. You can mitigate some of this with modular design.

7. Your files in this project can get quite large. Don't be afraid to split them up. Remember that you can have multiple implementation files per header file but you should still try to keep similar things together.

8. Remember that your style should be consistent. Now that you're in a team you should discuss some probably points of code contention to make sure you're on the same page.

9. This is the biggest assignment, thus **START EARLY**. **DO NOT USE LLVM IR VECTOR TYPES**. These types are designed for Single Instruction Multiple Data (SIMD) processing. They are also architecture specific on implementation. Using the LLVM IR vector types will result in a segmentation fault in architectures that do not support them. Not all lab machines support the *LLVM IR* vector.

10. The LLVM language implementation tutorial can help with some ideas on how to begin codegen. In particular, sections three and five are of particular interest. Section 7 is worth looking through for more discussion of the use of `alloca` over `phi` nodes. The other sections can be read at your own discrection.

11. The demo that was presented in lab can be found here. Remember to set it up in CLion just like you did with your regular project (environment variables).

CHAPTER

SEVENTEEN

LLVM TIPS AND HINTS

This section is likely to be constantly updated as new questions are asked or useful things are found. You will be notified as appropriate.

- It may be helpful to find out how *clang* translates equivalent *C* programs into *LLVM IR*. You can ask *clang* to output its generated *LLVM IR* via this command:

  ```
  clang -emit-llvm -S -c test.c
  ```

  Clang will sometimes optimise unused code away when we would really like to see what it's doing. Consider changing to a different optimsation level or disabling it completely (`-O0`). As well, try printing intermediate values, the program will be forced to evaluate them.

  While you can't use the text directly, it can give you an idea of what instructions are being created. The *LLVM* documentation is quite good. If you don't immediately understand an instruction, the *LLVM* language reference is a great resource, as is our class forums. The instruction generation function is often found under the same name in the IR builder.

  LLVM IR is not static and can change between versions. Often, things are not too different so using a different version will not affect you. If things are not working out and you would like to be absolutely sure, you can build *clang* yourself or use the executable that has been already built for you on the CSC lab machines (need to be sourcing our setup files). We operate from the release_60 branch in the c415 repository. The version command thus produces:

  ```
  clang version 6.0.1 (https://github.com/cmput415/clang.git␣
  ↪2f27999df400d17b33cdd412fdd606a88208dfcc) (https://github.com/cmput415/llvm.git␣
  ↪2c9cf4f65f36fe91710c4b1bfd2f8d9533ac01b5)
  Target: x86_64-unknown-linux-gnu
  Thread model: posix
  InstalledDir: /cshome/c415/415-resources/llvmi/bin
  ```

- The *LLVM* interface has its own small optimisations built in. Often this is only instruction reduction. Be aware that you may find code that you emitted to be missing. The easiest to see case is constant folding. If two constants are operated on, the operation will be completed before emitting any code and instead you will see only their result as a constant.

  You can disable code folding by adding `NoFolder` to your IRBuilder.

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/NoFolder.h"
llvm::IRBuilder<llvm::NoFolder> noFoldBuilder;
```

- Sometimes *LLVM* generates unexpected (but correct) code. For example, requesting an integer cast can generate a multitude of instructions based on size of operands and signedness.

For example, an unsigned cast from `i1` to `i32` will produce a `zext` (zero extend) instruction while a signed cast with the same types will produce a `sext` (sign extend) instruction, which is probably not what you want.

Be careful of downcasting. Asking for a cast from a larger type integer type to a smaller integer type will only ever produce a `trunc` (truncate) instruction. This is *correct* but it's not always what you want.

- In order to perform some operations, it is a good idea to create a library of functions yourself to use internally. These functions compose what is called your runtime. These can be especially useful when dealing with vectors and may help to keep your generated code more easily readable.

  For example, rather than generating the necessary guards and final load used in vector indices, it may make more sense to create an indexing function to handle this for you, replacing all codegen with a simple function call.

  You can make sure there is no naming conflicts by either suffixing or prefixing your internal runtime variables or all of the program variables. *LLVM IR* allows `.` characters in variable names while *VCalc* does not. This allows for easily guaranteed conflict-free names.

- Most instructions generated give you the opportunty to name the result if you want. While you don't need to, it can help you debug when things are going wrong.

- *LLVM* has an automatic way to verify modules for you. It can be a good idea to use it just before you output your code to make sure everything makes sense. This can be extremely helpful for noticing small errors. Here's a basic invocation for your output using the verifier.

```
#include "llvm/IR/Verifier.h"
#include "llvm/Support/raw_os_ostream.h"
#include <iostream>
...
llvm::raw_os_ostream llOut(outStream);
llvm::raw_os_ostream llErr(std::cerr);
llvm::verifyModule(mymodule, &llErr);
myModule.print(llOut, nullptr);
```

- **DO NOT USE LLVM IR VECTOR TYPES**. These types are designed for Single Instruction Multiple Data (SIMD) processing which require specific version of processors. Using the LLVM IR vector types will result in a segmentation fault in architectures that do not support them. Not all lab machines support the *LLVM IR* vector.

- You will need to divise your own vector type and method of storing data. One way is to use a linked list of structs with predefined integer arrays. Another is to `malloc`/`free` memory. Each has their own unique pros and cons.

- You need to make a `main` function to insert code into to begin with. Here's some boilerplate to get you rolling:

```
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/TypeBuilder.h"
#include "llvm/Support/Cast.h"
...
// Create main function, returns int, takes no args.
llvm::FunctionType *mainTy = llvm::TypeBuilder<int(), false>::get(ctx);
auto *mainFunc = llvm::cast<llvm::Function>(mod.getOrInsertFunction("main",␣
↪mainTy));

// Create an entry block and set the inserter.
llvm::BasicBlock *entry = llvm::BasicBlock::Create(ctx, "entry", mainFunc);
ir.SetInsertPoint(entry);
```

- When you run `lli` many common *c* functions are available, in particular you want `printf` to do your printing. To get `printf`, you need to add it to your module similarly to adding your `main`, but you do *not* define it. This

corresponds to your *c*-style forward declare and will make sure that llvm links `printf` into you executable. Here's your boilerplate code where `module` is your `llvm::Module`:

```
#include "llvm/IR/Attributes.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/TypeBuilder.h"
#include "llvm/Support/Cast.h"
...
// Declare printf. Returns int, takes string and variadic args.
llvm::FunctionType *printfTy = llvm::TypeBuilder<int(char *, ...), false>::get(ctx);
auto *printfFunc = llvm::cast<llvm::Function>(module.getOrInsertFunction("printf",␣
↪fTy));

// Add the suggested argument attributes.
printfFunc->addAttribute(1u, llvm::Attribute::NoAlias);
printfFunc->addAttribute(1u, llvm::Attribute::NoCapture);
```

- You may need to declare global constants in your module. The method for integers is similar to strings, but we show strings here because you will need it for use with `printf`. For example, if I wanted to create a `printf` format string for integers (`module` is `llvm::Module` and `context` is `llvm::Context`):

```
#include "llvm/IR/Constant.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/Support/Cast.h"
...
// Create the constant data array of characters.
llvm::Constant *intFormatStr = llvm::ConstantDataArray::getString(context, "%d");

// Create the global space we will use. The string "intFormatStr" is the name you␣
↪will need to
// to use to ask for this value later to get it from the module.
auto *intFormatStrLoc =
  llvm::cast<llvm::GlobalVariable>(
    module.getOrInsertGlobal("intFormatStr", intFormatStr->getType())
  );

// Set the location to be initialised by the constant.
intFormatStrLoc->setInitializer(intFormatStr);
```

- Calling functions is roughly the same in all places, but `printf` can be a little annoying to begin with because of the way it is defined, so here is some more boilerplate code for calling that as well (`module` is `llvm::Module`):

```
#include "llvm/IR/Function.h"
#include "llvm/Support/Cast.h"
...
// Note that we use getFunction not getOrInsertFunction. This will blow up if you␣
↪haven't
// previously defined printf in your module. See above.
llvm::Function *printfFunc = module.getFunction("printf");

// Get your string to print.
auto *formatStrGlobal = llvm::cast<llvm::Value>(mod.getGlobalVariable("my string␣
↪name"));
```

(continues on next page)

(continued from previous page)

```
// The type of your string will be [n x i8], it needs to be i8*, so we cast here. We
// explicitly use the type of printf's first arg to guarantee we are always right.
llvm::Value *formatStr =
  ir.CreatePointerCast(formatStrGlobal, printfFunc->arg_begin()->getType(),
↪"formatStr");

// Get our value.
llvm::Value *value = <appropriate code to get your value to print>;

// Call printf. Printing multiple values is easy: just add to the {}.
ir.CreateCall(printfF, {formatStr, value});
```

- In case you wanted calloc (or malloc) as well:

```
#include "llvm/IR/Attributes.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/TypeBuilder.h"
#include "llvm/Support/Cast.h"
...
// Declare calloc. Returns char *, takes array size, element size.
llvm::FunctionType *fTy = llvm::TypeBuilder<char *(size_t, size_t), false>
↪::get(ctx);
auto *callocFunc = llvm::cast<llvm::Function>(mod.getOrInsertFunction("calloc",␣
↪fTy));

// Add the suggested function attributes.
callocFunc->addFnAttr(llvm::Attribute::NoUnwind);
callocFunc->addAttribute(0, llvm::Attribute::NoAlias);
```

CHAPTER

EIGHTEEN

AST TIPS AND HINTS

This section is likely to be constantly updated as new questions are asked or useful things are found. You will be notified as appropriate.

- At first glance, ASTs may not seem to provide much value. Much of your *VCalc* AST will be identical to your parse tree. There are, however, good reasons to make use of them now. A motivating example:

  ```
  print(a + b);
  ```

  What are `a` and `b`? Integers? Vectors? One of each? How does your code generator know? Your AST can help you. You have a few options:

  1. Make your code generator figure it out.
  2. Attach type information to your AST at this node denoting each operands type during a type inference pass on your tree. Still need to check for extension.
  3. Do a type inference pass and replace the integer operand with an extension node so we only need to check the type of one operand to know the result type.
  4. Swap the operator node for a vector operator node and have the code generator assume that the non-vector version has integer operands while the vector one needs to check if there's an integer to extend.
  5. Add type information to the above solution so we only need to check that.
  6. Don't add type information and instead swap the operand node for an explict extension node and have both operator nodes assume the operands are of the right type.
  7. Anything else. It's up to you.

  This is just one example of where you could possibly use an AST to make your life easier along the way.

- You should create a tree traversal class in the same vein as ANTLR and its BaseVisitors. This way you can use the same mechanism for manipulating the tree, type checking, or final code generation.
- An AST is not a "scope tree". You can maintain a stack of tables that tell you what is currently in scope as you *traverse the tree* but scoping is not inherently part of the ast.

CHAPTER

NINETEEN

PROJECT LAYOUT

For the tools provided to work your project should be in the specified layout.

```
+-- cmake
|   +-- antlr_generate.cmake
|   +-- get_antlr.cmake
|   +-- get_antlr_manual.cmake
|   +-- get_llvm.cmake
|   +-- symlink_to_bin.cmake
+-- CMakeLists.txt
+-- grammar
|   +-- VCalc.g4
+-- include
|   +-- placeholder.h
+-- LICENSE.md
+-- README.md
+-- scripts
|   +-- configureLLVM.sh
+-- src
|   +-- CMakeLists.txt
|   +-- main.cpp
+-- tests
    +-- input
    |   +-- ...
    +-- output
        +-- ...
    +-- VCalcConfig.json
```

CHAPTER

TWENTY

SETTING UP CLION

CLion requires a little bit of setup. Much of it is the same, but we need to add LLVM now.

1. Open up CLion. From the welcome screen select `Import Project from Sources` or, if you've been using CLion and it opens to previous project, from the `File` menu select `Import Project...`. Navigate to where your project is located an choose it. Choose `Open Project` *not* `Overwrite CMakeLists.txt`. If you already have a project open, you can choose to use your current window or create another one.
2. CLion doesn't make use of your command line environment, it has its own storage place. Therefore we need to add `ANTLR_INS` and `LLVM_DIR` to CLion's environment.
    1. Open your settings. On Linux this is `File` → `Settings...`, while on MacOS this is `CLion` → `Preferences...`.
    2. From the left menu, expand `Build, Execution, Deployment`.
    3. Select `CMake` from the newly expanded options.
    4. In the right pane, select the `...` to the right of the empty text field to the right of `Environment`.
    5. In the new pane, select the + symbol to add a new entry in the environment. On Linux this is in the top right of the pane, while on MacOS this is in the bottom left of the pane.
    6. In the new text field under `Name` enter `ANTLR_INS`. In the field under `Value` enter the path to your `antlr_install` directory. If you've forgotten it but your terminal is set up correctly, or if you're using the lab machines, then you can enter the following command to print it:

        ```
        echo $ANTLR_INS
        ```

    7. Select the + symbol to add another symbol to your environment. In the field under `Name` enter `LLVM_DIR`. Since the value depends on how you have set up your environment (or how we have if you're using the lab machines) you will need to enter this command in your terminal to find the value.

        ```
        echo $LLVM_DIR
        ```

        Add this value in the `Value` field.
    8. Apply all of your changes while closing the settings.
3. Make sure you're building the `all` target, not just the `scalc` target. From the drop down menu in the top right of the IDE you can choose your build target. Change this to `Build All`.
4. CLion may not automatically pick up ANTLR's generated sources as your project's. We can fix this by telling CLion where the files are. Build once to have the `gen` directory appear in the project manager pane. Right click on the directory, near the bottom of the menu find `Mark directory as`, within that menu select `Project Sources and Headers`.

CHAPTER

TWENTYONE

TESTING

21.1 Testing Tool

Inside the `tests` directory, a testing configuration file, `VCalcConfig.json`, is provided. You need to edit `inDir` with the absolute path of `.../tests/input`, `outDir` with the absolute path of `.../tests/output`, `testedExecutablePaths` with your ccid and the absolute path of `.../bin/vcalc`, and finally `runtimes` with your ccid and the absolute path of `.../bin/libvcalcrt.so`

Running the tester should now run your tests with your solution. Note that the output files will not be cleaned up so it might be wise run your tests in a directory that is not your repository. If you would still like to run them in the same directory, just know that the output files do not need to be tracked by git and can be safely deleted.

```
tester <path_to_config>
```

For more information about the testing tool and how it works, see the Tester README.

21.2 Generating Test Cases

A Python script `fuzzer.py` is available for automatic generation of random test cases.

More information is available in the README file included with the script, but an example usage of the script is as follows:

```
python fuzzer.py config.json test
```

The above command will generate two files: `test.in` and `test.out`, where `test.in` contains the VCalc source code of the test case and `test.out` contains the expected output. These files can be placed in your `tests` directory for use with the *Testing Tool*.

On the CSC Lab Computers, the fuzzer is located in the directory `/cshome/cmput415/415-resources/fuzzers/VCalc`. Be sure to use `python3` when running fuzzers on the CSC Lab Computers.