vcalc

cmput415

Jun 11, 2025

CONTENTS

1	Keywords	3		
2	Vectors			
3	Range	7		
4	Generators	9		
5	Filters	11		
6	Expressions 6.1 Operators 6.2 Binary Operations on Vectors 6.3 Integer to Vector Promotion 6.4 Vector Indexing	13 13 13 14 15		
7	Statements 7.1 Declaration 7.2 Assignment 7.3 Conditional 7.4 Loops 7.5 Print	17 17 17 18 18 18		
8	Comments 19			
9	Type Checking	21		
10	Scoping	23		
11	Input	25		
12	Output	27		
13	Assertions	29		
14	Clarifications	31		
15	Deliverables	33		
16	Tips and Hints	35		
17	MLIR Tips and Hints	37		

18 AST Tips and Hints

This assignment expands the simple calculator, *SCalc*, to build a vector calculator called *VCalc*. For *VCalc* you will build a compiler that generates MLIR. As *MLIR* is an IR infrastructure it supports many special purpose intermediate representations called *dialects*. You will target the LLVM Dialect in this assignment. All *MLIR* dialects must evenually be lowered to *LLVM IR*, which is the common IR that the LLVM back-end uses to generate machine specific object code. None of the assembly back ends that you built for *SCalc* need to be supported for *VCalc*, because the LLVM back-end can support them all. 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 *SCalc* that are now reserved Keywords.) *VCalc* has the additional features discussed in subsequent section.

ONE

KEYWORDS

The following keywords are now also reserved in VCalc:

- in
- vector

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 $[0, 2^{31} - 1]$.

Assertion: All vectors will have length l such that $0 \le l \le 2^{31} - 1$. (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 vector.

THREE

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:

 $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}$ $\begin{bmatrix} 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{bmatrix}$

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:

[]

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]

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 a vector index. The predicate 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 write:

print([i in 1..10 & 5 < i]);</pre>

prints the following:

[6 7 8 9 10]

EXPRESSIONS

6.1 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
Grouping	parentheses	0	(expr)	N/A
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

6.2 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:

[79]	11 9	10]
[79]	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]

6.3 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);</pre>

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 $[6 \ 7 \ 8 \ 4 \ 5]$.

6.4 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 $0 \le |v| < l$ and i < 0 or $i \ge l$) 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:

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.

SEVEN

STATEMENTS

7.1 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]
```

7.2 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;

7.3 Conditional

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

7.4 Loops

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

7.5 Print

The print statement in *VCalc* behaves the same as *SCalc* for integers, but must be extended to print vectors. All the elements of the vector are printed on a single line between the opening and closing brackets.

For example:

```
print(1..10);
```

prints the following:

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

The output of print 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*)

EIGHT

COMMENTS

VCalc supports a subset of C99 comments.

Single line comments are made using //. Anything on the line after the two adjacent forward slashes is ignored. For example:

// A comment on its own line
print(v); // This is ignored

NINE

TYPE CHECKING

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 are 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 to integers).
- Filters: the predicate must be a boolean (remember that integers can be implicitly downcast to booleans).

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 1 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.

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.

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.

The output of your compiler is the *LLVM IR* that corresponds to the given input program. The output file can be used as input to 11c, for example, which compiles *LLVM IR* input into assembly language for a specified machine. From there the tools you used in *SCalc* can be used to create an executable.

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 $0 \le l \le 2^{31} - 1$. Trying to create an index greater than $2^{31} - 1$ will cause overflow and result in a negative number. Indexing with a negative number returns 0. Therefore, vector locations greater than $2^{31} - 1$ 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)..2147483644);

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:

r	٦
L	
L	-

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]

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.

SIXTEEN

TIPS AND HINTS

1. The learning curve for *LLVM* and *MLIR* is not trivial: **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 *MLIR/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 probable 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 Getting Started with the LLVM System can help introduce you to LLVM. For MLIR, see Getting Started. In particular, sections three and five of the Kaleidoscope tutorial are of particular interest for generating LLVM IR. Generating MLIR is similar. 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 the lab can be found here. Remember to set it up in CLion just like you did with your regular project (environment variables).

CHAPTER SEVENTEEN

MLIR 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 optimisation level or disabling it completely (-00). 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.

- It can be a little harder to find programs that can emit MLIR. Tensorflow and flang (Fortran compiler) are two or the more well-known compilers that use MLIR, but both are admittedly niche. As an alternative, you can use the mlir::dump() method, which works on all MLIR operations including modules and functions. The MLIR framework provides several tools that can parse and work with files containing MLIR. In particular, mlir-opt can be used to run almost every optimization or transformation pass that exists on your MLIR output. For more information, execute mlir-opt --help after building and setting up MLIR.
- While there appears to be copious quantities of *MLIR/LLVM* documentation, it can be frustatingly difficult to find documentation or examples of something you care about. One of the more effective tools is grep, especially when used on the MLIR repository. git grep is automatically recursive, but because a lot of MLIR is generated at build time, grep -R will search files in the build tree but outside the repository.
- 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. *MLIR* and *LLVM* allow . characters in variable names while *VCalc* does not. This allows for easily guaranteed conflict-free names.

• *MLIR* 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 "mlir/IR/Verifier.h"
...
if (mlir::failed(mlir::verify(module)))
   std::cerr << "verification failed :-(" << std::endl;</pre>
```

- Stick to the ensorsed dialects found in the base repository.
- 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 (note builder is type mlir::OpBuilder):

```
#include "mlir/Dialect/LLVMIR/LLVMDialect.h"
#include "mlir/IR/BuiltinAttributes.h"
#include "mlir/IR/Builders.h"
#include "mlir/IR/BuiltinOps.h"
#include "mlir/IR/Value.h"
...
// For our purposes, the prototype for main can be "int main()"
mlir::Type intType = builder->getI32Type();
auto mainType = mlir::LLVM::LLVMFunctionType::get(intType, {}, false);
mlir::LLVM::LLVMFuncOp mainFunc = builder->create<mlir::LLVM::LLVMFuncOp>(builder->
_getUnknownLoc(), "main", mainType);
// Create an entry block and set the inserter.
mlir::Block *entry = mainFunc.addEntryBlock();
builder->setInsertionPointToStart(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 declaration and will make sure that llvm links printf into you executable. Here's your boilerplate code where builder is type mlir::OpBuilder:

(continues on next page)

(continued from previous page)

• 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 newline (builder is type mlir::OpBuilder, context is type mlir::MLIRContext, and loc is type mlir::Location):

• 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 (builder is type mlir::OpBuilder, module is type mlir::ModuleOp, context is type mlir::MLIRContext, and loc is type mlir::Location):

```
#include "mlir/Dialect/LLVMIR/LLVMDialect.h"
#include "mlir/IR/Builders.h"
#include "mlir/IR/BuiltinOps.h"
#include "mlir/IR/Value.h"
. . .
mlir::LLVM::GlobalOp global;
if (!(global = module.lookupSymbol<mlir::LLVM::GlobalOp>("newline"))) {
    llvm::errs() << "missing format string!\n";</pre>
    return;
}
// Get the pointer to the first character in the global string.
mlir::Value globalPtr = builder->create<mlir::LLVM::AddressOfOp>(loc, global);
mlir::Value cst0 = builder->create<mlir::LLVM::ConstantOp>(loc,
                                           builder->getI64Type(),
                                           builder->getIndexAttr(0));
mlir::Type charType = mlir::IntegerType::get(&context, 8);
mlir::Value newLine = builder->create<mlir::LLVM::GEPOp>(loc,
              mlir::LLVM::LLVMPointerType::get(charType),
              globalPtr, mlir::ArrayRef<mlir::Value>({cst0, cst0}));
auto printfFunc = module.lookupSymbol<mlir::LLVM::LLVMFuncOp>("printf");
builder->create<mlir::LLVM::CallOp>(loc, printfFunc, newLine);
```

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 operand's 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 explicit 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.