Prof. Monica Shah
Compiler Construction

How To Make A Programming Language

With LLVM

Sachi Chaudhary 19BCE230
Aayush Shah 19BCE245

Index

INTRODUCTION

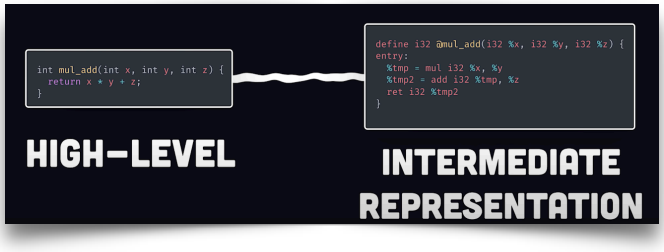
Every programmer might have thought about building own programming language at some point in their life. We wanted to do the same.

As it seems, It will not be a cup of cake but we'll do it for sure. Also it depends on how complex you want your language be. With this guide, hopefully you'll have a better concept of how to implement your language. At the time of writing, We'll be employing newer technologies such as LLVM, which thankfully do a lot of the grunt work for us. So, with everything out the way, let's get started!

What is LLVM ?

LLVM a toolkit used to build and optimise compilers. Building a programming language from scratch is hard. You as humans who want to write code in a nice simple syntax than machines that need to run it on all sorts of architectures, LLVM standardises the extremely complex process of turning source code into machine code. It was created in 2003 by grad student Chris Lattner at the University of Illinois and today it's the magic behind clang for C and C++ as well as languages like Rust, Swift, Julia and many others.

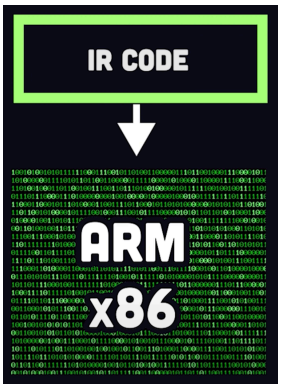
Most importantly it represents high level source code in a language agnostic code called intermediate representation or IR. This means vastly different languages like Cuda and Ruby produce the same IR allowing them to share tools for analysis and optimisation before they're converted to machine code for a specific chip architecture.



Front End

Middle End

Back End

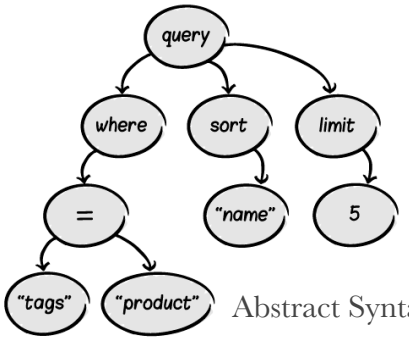


A compiler can be broken down into three parts: The front end parses the source code text and converts it into IR. The middle end analyses and optimises this generated code and finally the backend converts the IR into native machine code.

To build your own programming language from scratch right now :

1. Install LLVM then create a .c file now. Envision the programming language syntax of your dreams to make that high level code work.
2. **LEXER** *(turn raw text into tokens)* : You'll first need to write a Lexer to scan the raw source code and break it into a collection of tokens like literals identifiers keywords operators and so on next we'll need to define an Abstract Syntax Tree to represent the actual structure of the code and how different tokens relate to each other which is accomplished by giving each node its own class.

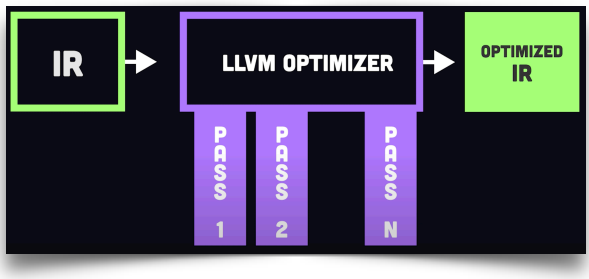
Token name	Sample token values
identifier	x, color, UP
keyword	if, while, return
separator	}, (, ;
operator	+, <, =
literal	true, 6.02e23, "music"



Abstract Syntax tree

3. **Parser** *(construct the AST)* : Third, we need a parser to loop over each token and build out the abstract syntax tree if you made it this far, congratulations! because the hard part is over.
4. **IR** *(generate intermediate representation code)* : Now we can import a bunch of LLVM primitives to generate the intermediate representation each type in the abstract syntax tree is given a method called *codegen* which always returns an LLVM value object used to represent a single assignment register which is a variable for the compiler that can only be assigned once. what's interesting about these IR primitives is that, unlike assembly they're independent of any particular machine architecture and that dramatically simplifies

things for language developers who no longer need to match the output to a processor's instruction set. Now that the front end can generate IR the op tool is used to analyse and optimise the generated code it makes multiple passes over the IR and does things like *dead code elimination* and *scalar replacement of aggregates* and finally that brings us to the back end where we write a module that takes IR as an input that emits object code that can run on any architecture. Congratulations, you just built your own custom programming language and compiler. Anyways, let's dive deeper!



Welcome to the manual. Here, we walk through the creation of a simple language, demonstrating how enjoyable and straightforward it can be. This lesson will get you up and running quickly and show you an actual example of something that generates code using LLVM.

This tutorial presents the simple "Kaleidoscope" which is a toy language, which is created iteratively over multiple chapters to demonstrate how it evolves over time. This allows us to cover a variety of language design and LLVM-specific ideas while showing and explaining the code along the way, reducing the overwhelming amount of information up front. We strongly encourage you to experiment with this code - make a copy, hack it up, and play with it.

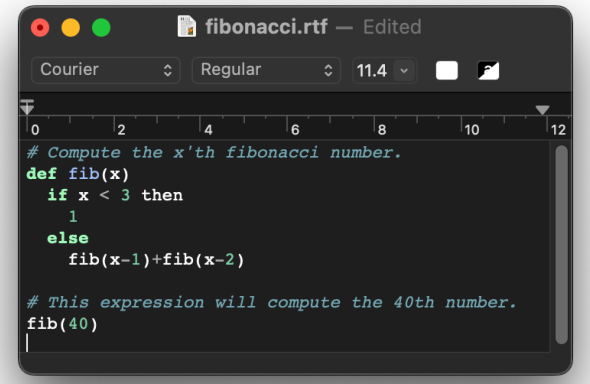
CHAPTER 1
KALEIDOSCOPE LANGUAGE AND LEXER

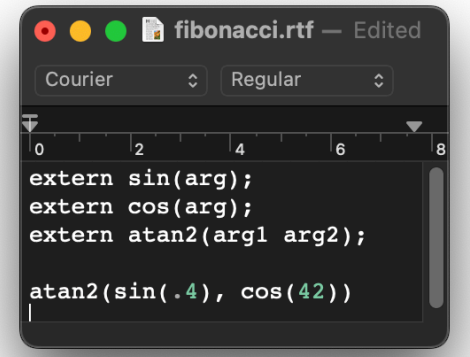
This section depicts our intended path and the fundamental functionality that we intend to implement. A lexer is also the first step in creating a language parser, and we utilise a simple C++ lexer that is straightforward to grasp.

1.1 The Language

This tutorial uses a toy language called "Kaleidoscope" (which means "beautiful, form, and vision"). Kaleidoscope is a procedural programming language that allows you to construct functions, utilise conditionals, perform math, and so on. After the tutorial, You can also add support for the if/then/else construct, a for loop, user defined operators, JIT compilation with a simple command line interface, debug information, and more. And for that, we have also given examined precise resources.

To keep things simple, Kaleidoscope's only datatype is a 64-bit floating point type (called 'double' in C terminology). As a result, all values are inherently double precision, and type declarations are not required. This results in a really nice and simple grammar for the language. The following basic example, for example, computes Fibonacci numbers:

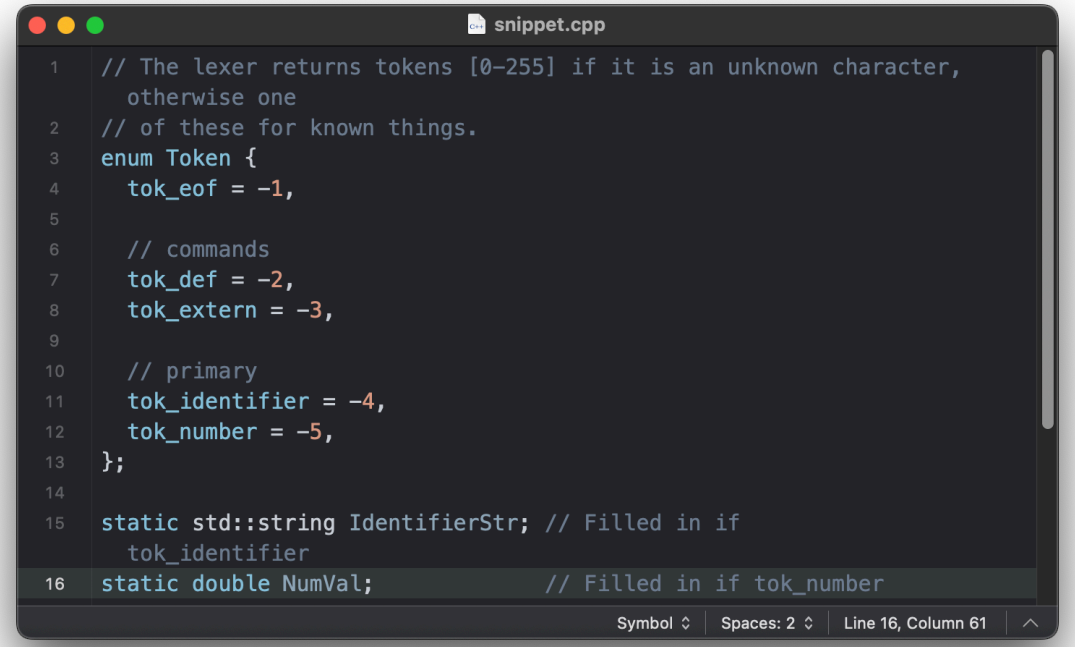




We also allow Kaleidoscope to call standard library functions, which the LLVM JIT makes very simple. This means that you can define a function before using it by using the 'extern' keyword. As an example shown above.

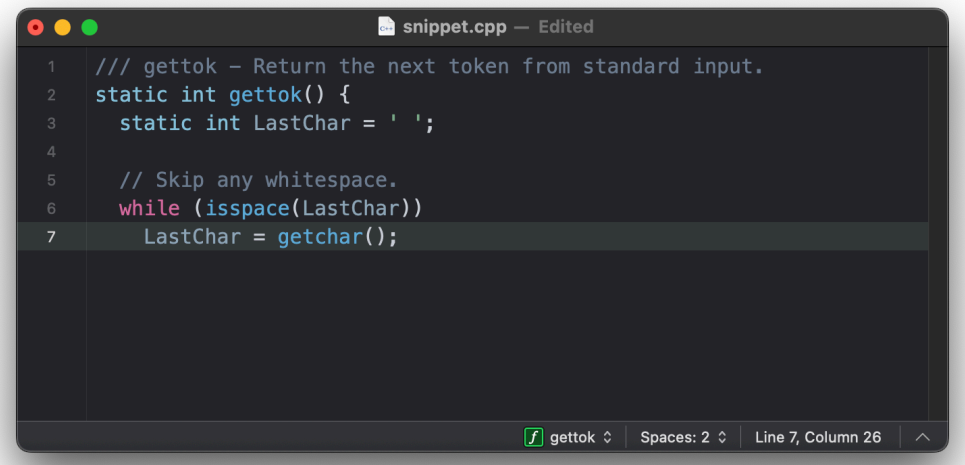
1.2 The Lexer

When it comes to implementing a language, the ability to process a text file and recognise what it says is essential. The traditional method is to use a "lexer" (also known as a "scanner") to divide the input into "tokens." Each token returned by the lexer contains a token code as well as possibly some metadata. First, we define the options:



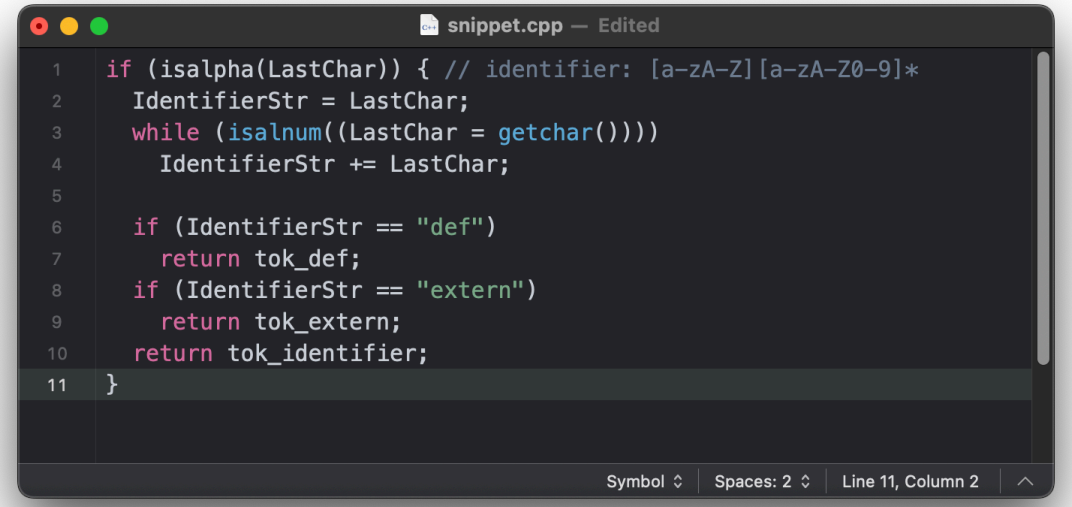
Each token returned by our lexer will be one of the Token enum entries or a 'unknown' character such as '+', which is returned as its ASCII value. If the current token is an identifier, the name of the identifier is stored in the IdentifierStr global variable. NumVal retains the value of the current token if it is a numeric literal (such as 1.0). We use global variables for convenience, but this is not the ideal option for a real-world implementation:).

The lexer is actually implemented as a single function called gettok. To return the next token from standard input, the gettok function is called. Its definition begins with:

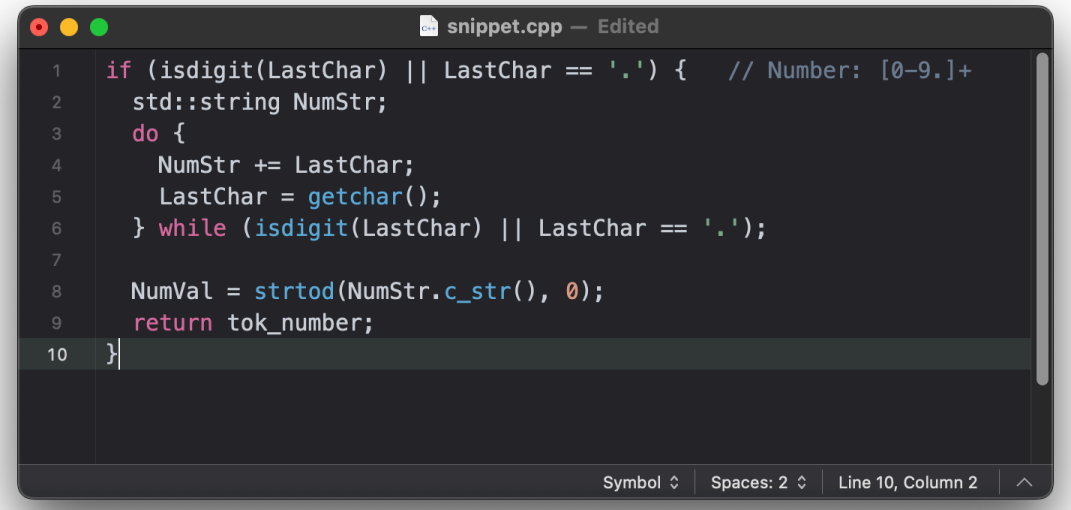


gettok reads characters from standard input one at a time using the C *getchar()* method. It consumes them as they are recognised and stores the last character read but not processed in LastChar. The first thing it must do is disregard whitespace between tokens. The loop mentioned above accomplishes this.

The next step for *gettok* is to recognise identifiers and specific keywords like "*def*." Kaleidoscope accomplishes this with a simple loop:



When this code lexes an identifier, it sets the global variable '*IdentifierStr*'. Also, because language keywords are matched by the same loop, we handle them inline here. The following numerical values are similar:

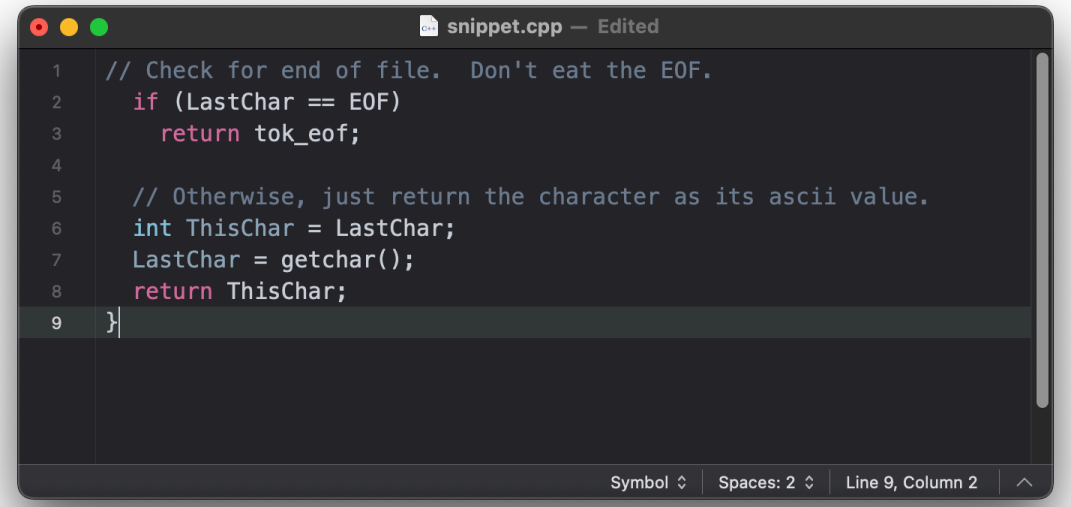


This is all relatively standard input processing code. We utilise the C strtod function to transform a numeric number read from input into a numeric value that we store in NumVal.

This isn't doing enough error checking: it will incorrectly read "1.23.45.67" and treat it as if you typed “1.23.”. *Please feel free to extend it though!*

Let’s Handle comments now :

When dealing with comments, we skip to the end of the line and then return the next token. Finally, if the input does not match one of the preceding cases, it is either an operator character such as '+' or the file's end. This code handles the following:



This completes the lexer for the fundamental Kaleidoscope language (the full code listing for the Lexer is available in the next chapter of the manaul and *github gist* is given algonwith). We'll then create a simple parser that leverages this to generate an AST.

CHAPTER 2
IMPLEMENTING A PARSER AND AST

We can now discuss parsing approaches and basic AST creation with the lexer in place. This course will go over recursive descent and operator precedence parsing.

2.1 Introduction

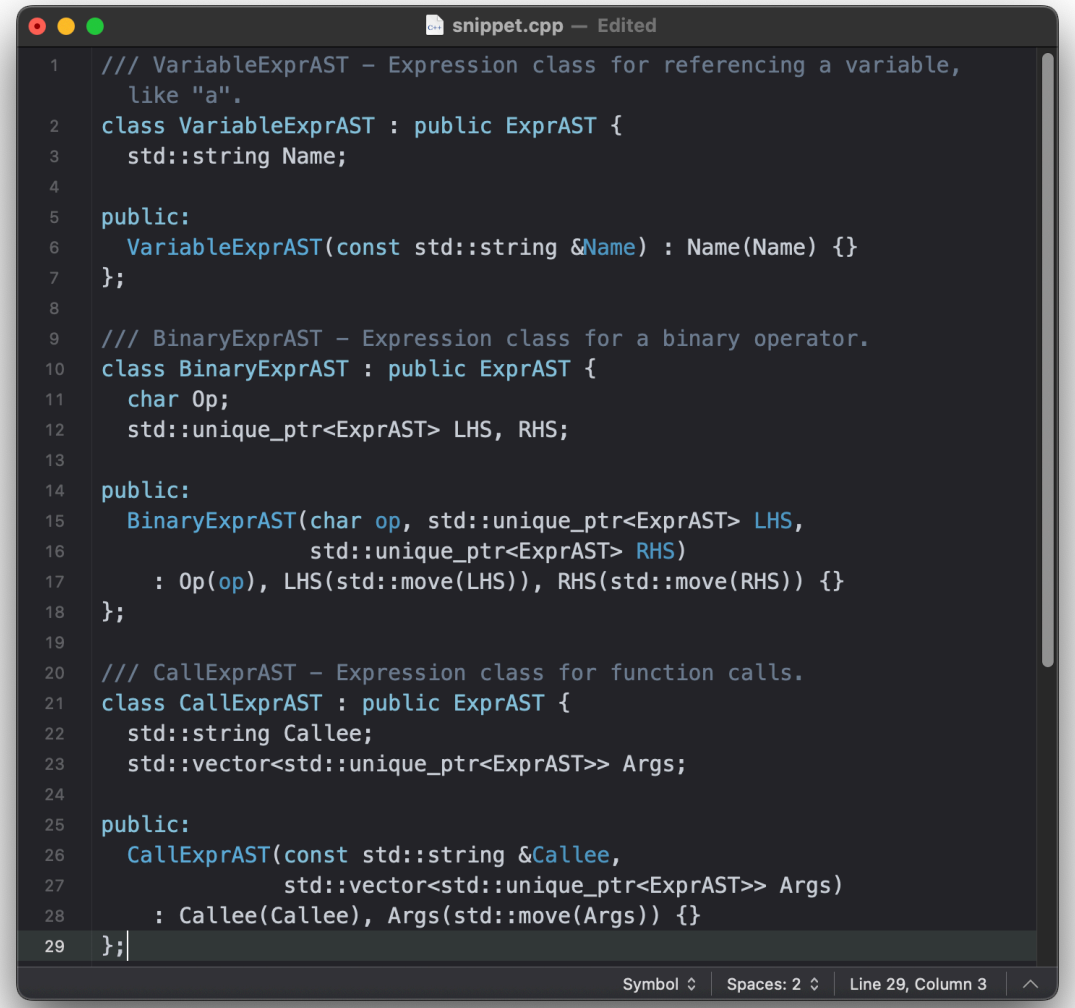
To parse the Kaleidoscope language, we will utilise a combination of Recursive Descent Parsing and Operator-Precedence Parsing (the latter for binary expressions and the former for everything else). However, before we begin parsing, we should discuss the parser's output: the Abstract Syntax Tree.

2.2 The Abstract Syntax Tree

The AST for a programme encodes its behaviour in such a way that following stages of the compiler (e.g., code generation) may easily read it. We essentially want one object for each language construct, and the AST should closely mimic the language. We have expressions, a prototype, and a function object in Kaleidoscope. We'll begin with expressions:

The code above defines the base ExprAST class as well as one subclass that we use for numeric literals. The crucial thing to remember about this code is that the NumberExprAST class stores the literal's numeric value as an instance variable. This allows the compiler to determine the stored numeric value in later stages.

We are currently only creating the AST, so there are no useful accessor methods on them. It would be very simple to add a virtual method, for example, to pretty print the code. Here are the other expression AST node definitions that will be used in the Kaleidoscope language's basic form:



Variables record the variable name, binary operators capture their opcode (e.g. '+'), and calls capture a function name as well as a list of any parameter expressions. One advantage of our AST is that it captures linguistic aspects without discussing the language's syntax. There is no mention of binary operator precedence, lexical organisation, or anything else.

These are all of the expression nodes we'll define for our basic language. It isn't Turing-complete because it lacks conditional control flow; you can fix that by having a look at the mentioned references in the later chapters.

Next, we need a way to talk about the interface to a function, as well as a way to talk about functions themselves:

snippet.cpp

```cpp
/// PrototypeAST - This class represents the "prototype" for a
  function,
/// which captures its name, and its argument names (thus implicitly
  the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;

public:
  PrototypeAST(const std::string &name, std::vector<std::string> Args)
    : Name(name), Args(std::move(Args)) {}

  const std::string &getName() const { return Name; }
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto;
  std::unique_ptr<ExprAST> Body;

public:
  FunctionAST(std::unique_ptr<PrototypeAST> Proto,
              std::unique_ptr<ExprAST> Body)
    : Proto(std::move(Proto)), Body(std::move(Body)) {}
};
```

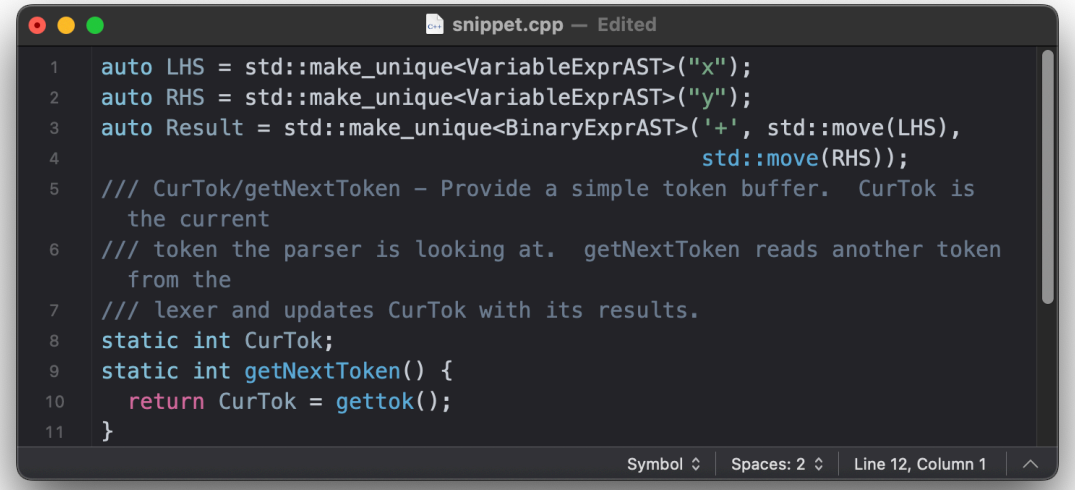
Symbol | Spaces: 2 | Line 24, Column 3

Functions in Kaleidoscope are typed using only the number of parameters. Because all values are double precision floating point, the type of each parameter is not required to be kept. The "ExprAST" class would almost certainly have a type field in a more aggressive and realistic language.

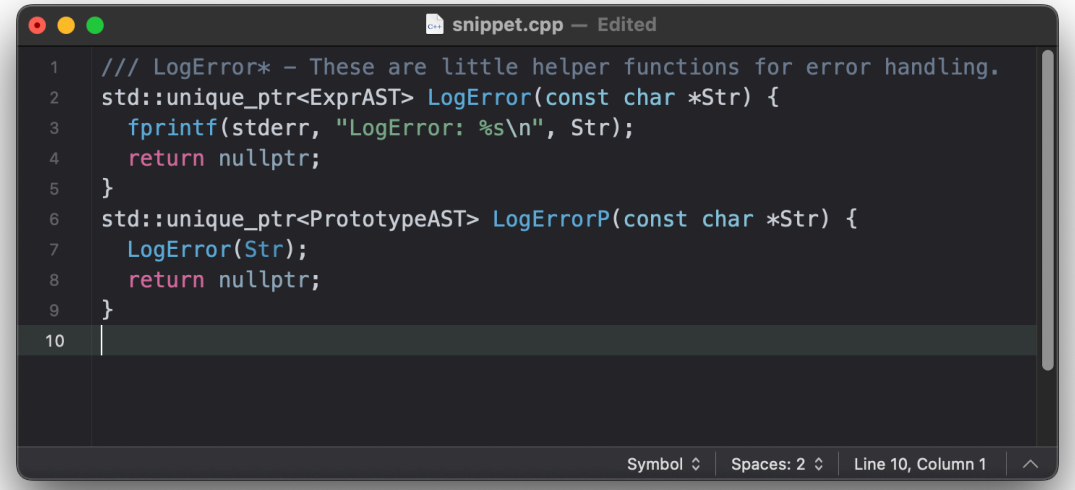
We can now discuss parsing expressions and function bodies in Kaleidoscope using this scaffolding.

2.3 Parser Basics

Now that we have an AST to work with, we must describe the parser code that will be used to build it. The aim here is to parse anything like "x+y" (which the lexer returns as three tokens) into an AST that can be created with calls like given below. Moreover, In order to do this, we'll start by defining some basic helper routines:



This wraps the lexer in a simple token buffer. This allows us to see what the lexer will return one token in advance. Every function in our parser will assume that *CurTok* is the currently parsed token.



Our parser will handle problems using the *LogError* procedures, which are simple assistance routines. Our parser's error recovery will not be the best and will not be particularly user-friendly, but it will suffice for our manual. These procedures make handling mistakes in routines with different return types easy because they always return null.

We can use these simple helper functions to implement the first part of our grammar: numerical literals.

2.4 Basics Expression Parsing

We begin with numeric literals because they are the easiest to understand. For each production in our grammar, we'll construct a function that parses it. We have the following numerical literals:

snippet.cpp

```
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken(); // consume the number
  return std::move(Result);
}
/// parenexpr ::= '(' expression ')'
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // eat (.
  auto V = ParseExpression();
  if (!V)
    return nullptr;

  if (CurTok != ')')
    return LogError("expected ')'");
  getNextToken(); // eat ).
  return V;
}
```

ParseParenExpr Spaces: 2 Line 18, Column 2

This method is pretty straightforward: it anticipates being called when the current token is a tok number token. It starts with the current number value, then generates a *NumberExprAST* node, moves the lexer to the next token, and finally returns.

This has some intriguing aspects to it. The most important is that this routine consumes all tokens associated with the production and returns the lexer buffer with the next token (which is not part of the grammar production) ready to go. This is a fairly common approach for recursive descent parsers. For a better illustration, consider the parenthesis operator.

This function demonstrates several interesting aspects of the parser:

1) It demonstrates how the LogError methods are used. When called, this function expects the current token to be a '(' token, but after parsing the subexpression, there may be no ')' waiting. For example, if a user enters "(4 x")" instead of "(4)," the parser should generate an error. Because errors can occur, the parser must provide a means of indicating that they have occurred: in our parser, we return null on an error.

2) Another intriguing feature of this function is that it employs recursion by invoking ParseExpression. This is useful since it allows us to handle recursive grammars while keeping each production as basic as possible. It should be noted that parentheses do not cause the formation of AST nodes. While we could implement it this way, the main function of parentheses is to assist the parser and give grouping. Parentheses are not required after the parser has constructed the AST.

The following simple production handles variable references and function calls:

snippet.cpp — Edited

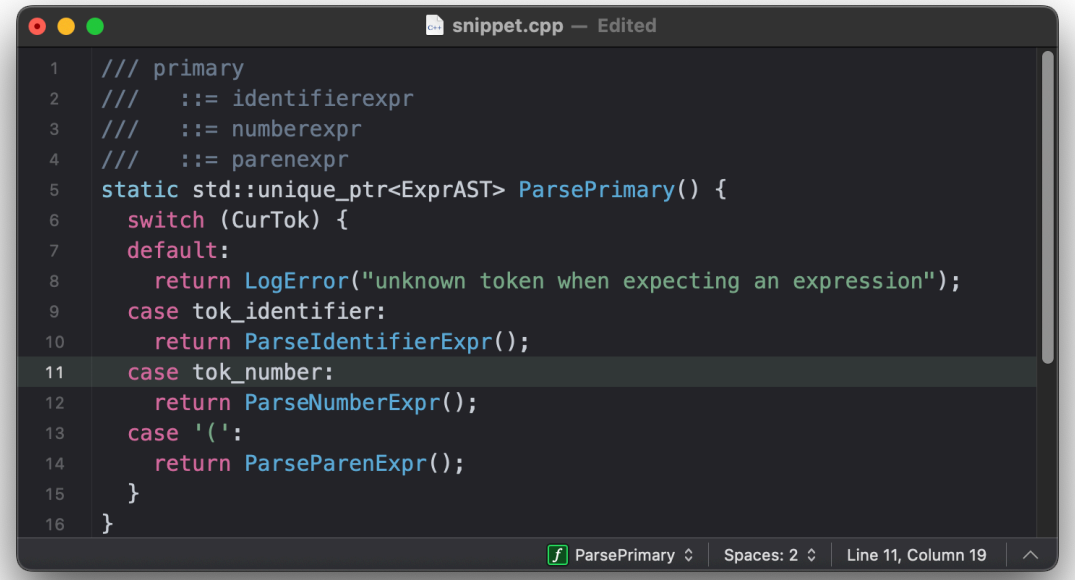
```cpp
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return std::make_unique<VariableExprAST>(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
    while (1) {
      if (auto Arg = ParseExpression())
        Args.push_back(std::move(Arg));
      else
        return nullptr;

      if (CurTok == ')')
        break;

      if (CurTok != ',')
        return LogError("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return std::make_unique<CallExprAST>(IdName, std::move(Args));
}
```

Symbol | Spaces: 2 | Line 36, Column 1

This routine is written in the same manner as the others. (If the current token is a tok identifier token, it expects to be called.) It also has error handling and recursion. One intriguing feature is that it use look-ahead to detect whether the current identifier is a stand-alone variable reference or a function call expression. It handles this by determining whether

the token following the identifier is a '(' token and then generating a *VariableExprAST* or *CallExprAST* node as needed.

We can build a helper function to package all of our simple expression-parsing logic into a single entry point now that we have it all in place. For reasons that will become clear later in the tutorial, we call this class of expressions "primary" expressions. To parse an arbitrary primary expression, we must first determine what type of expression it is:

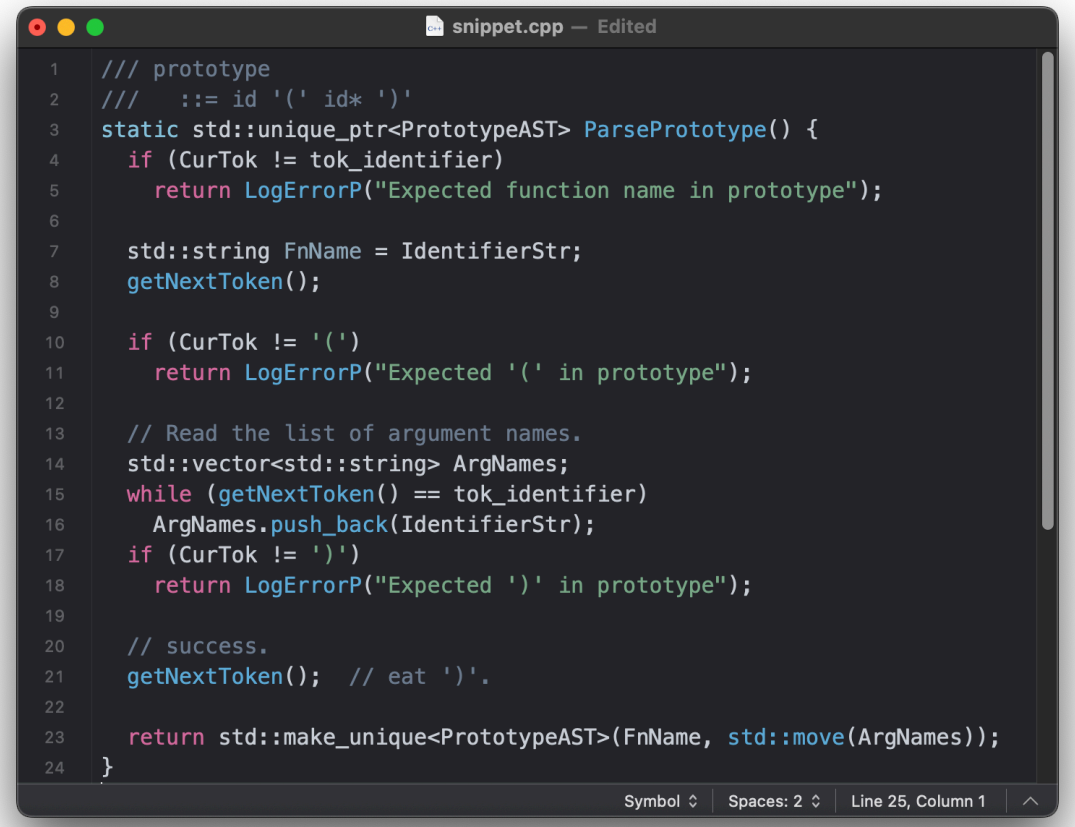


Now that you've seen the definition of this function, you can see why we can assume CurTok's state in the various functions. Look-ahead is used to determine which type of expression is being inspected, and then a function call is used to parse it.

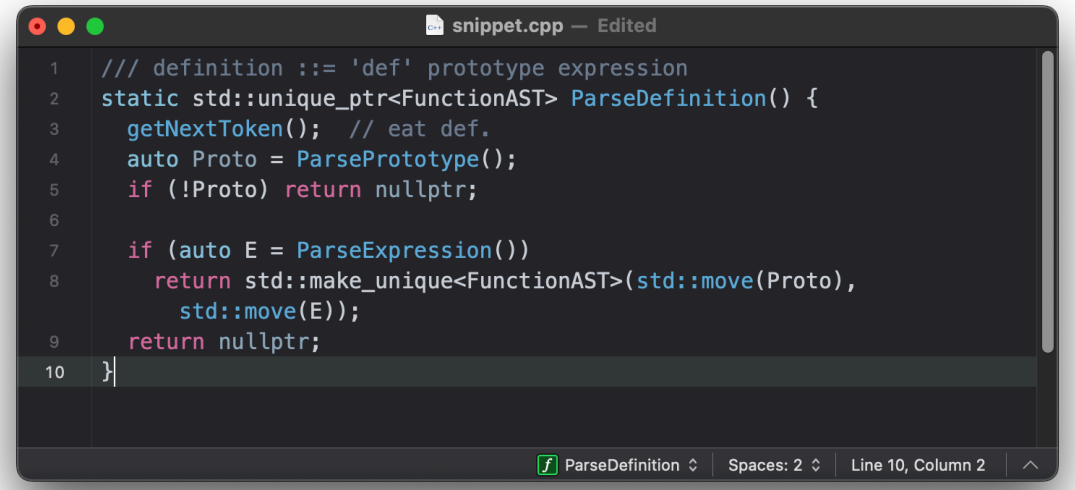
Now that basic expressions have been handled, we can move on to binary expressions. As they are a little more complicated, we can leave them but you can refer them on : Binary expression parsing

2.5 Parsing the rest

The handling of function prototypes is the next thing that is missing. These are used in Kaleidoscope for both 'extern' function declarations and function body definitions. The code to accomplish this is simple and uninteresting (once you've gotten beyond the expressions):

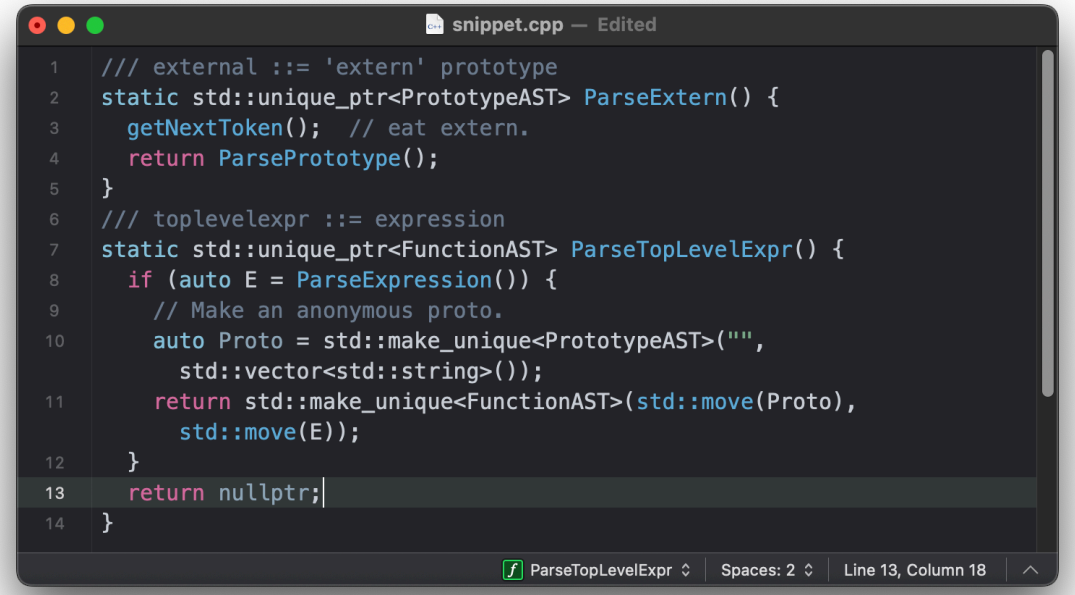


Given this, defining a function is fairly straightforward, consisting of only a prototype and an expression to implement the body:



Furthermore, we support 'extern' to declare functions such as '*sin*' and '*cos*,' as well as forward declaration of user functions. These 'externs' are merely prototypes with no bodies:

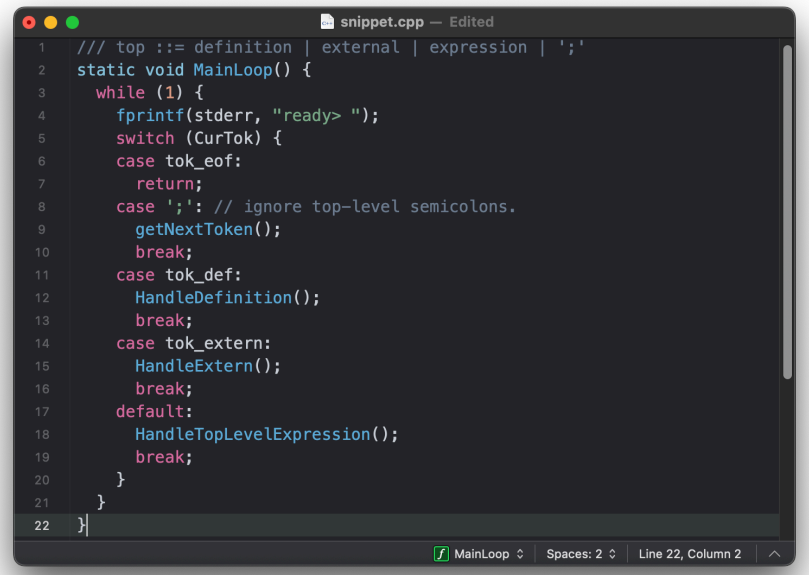
Along with that, we'll allow the user to enter arbitrary top-level expressions and have them evaluated on the fly. We'll deal with this by creating anonymous nullary (zero argument) functions for them:



Now that we have all of the pieces, let's create a small driver that will allow us to run the code we've written!

2.6 The Driver

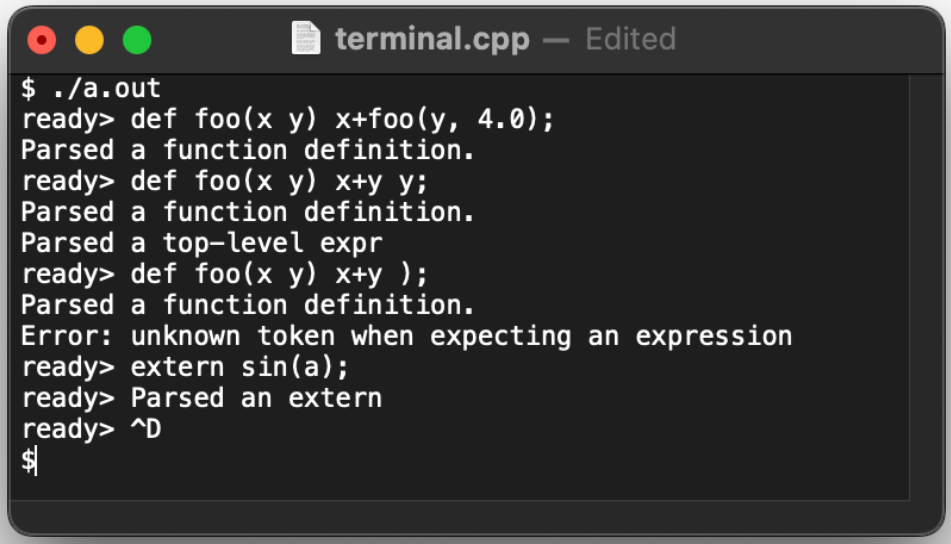
The driver just calls all of the parser components via a top-level dispatch loop. Because there isn't much to see here, We'll just include the top-level loop. See the "Top-Level Parsing" section for the complete code.



The most intriguing aspect of this is that we ignore top-level semicolons. Why is this, you may wonder? The basic reason is that if you type "4 + 5" at the command line, the parser does not know whether that is the end of what you will type or not. For example, on the next line, you could type "def foo...", in which case 4+5 represents the end of a top-level expression. You could also type "* 6" to continue the expression. With top-level semicolons, you can type "4+5;" and the parser will know you're done.

2.7 Conclusions

We fully defined our minimal language, including a lexer, parser, and AST builder, in just under 400 lines of commented code (240 lines of non-comment, non-blank code). After that, the executable will validate the Kaleidoscope code and notify us if it is grammatically incorrect. Here's an example of an interaction:



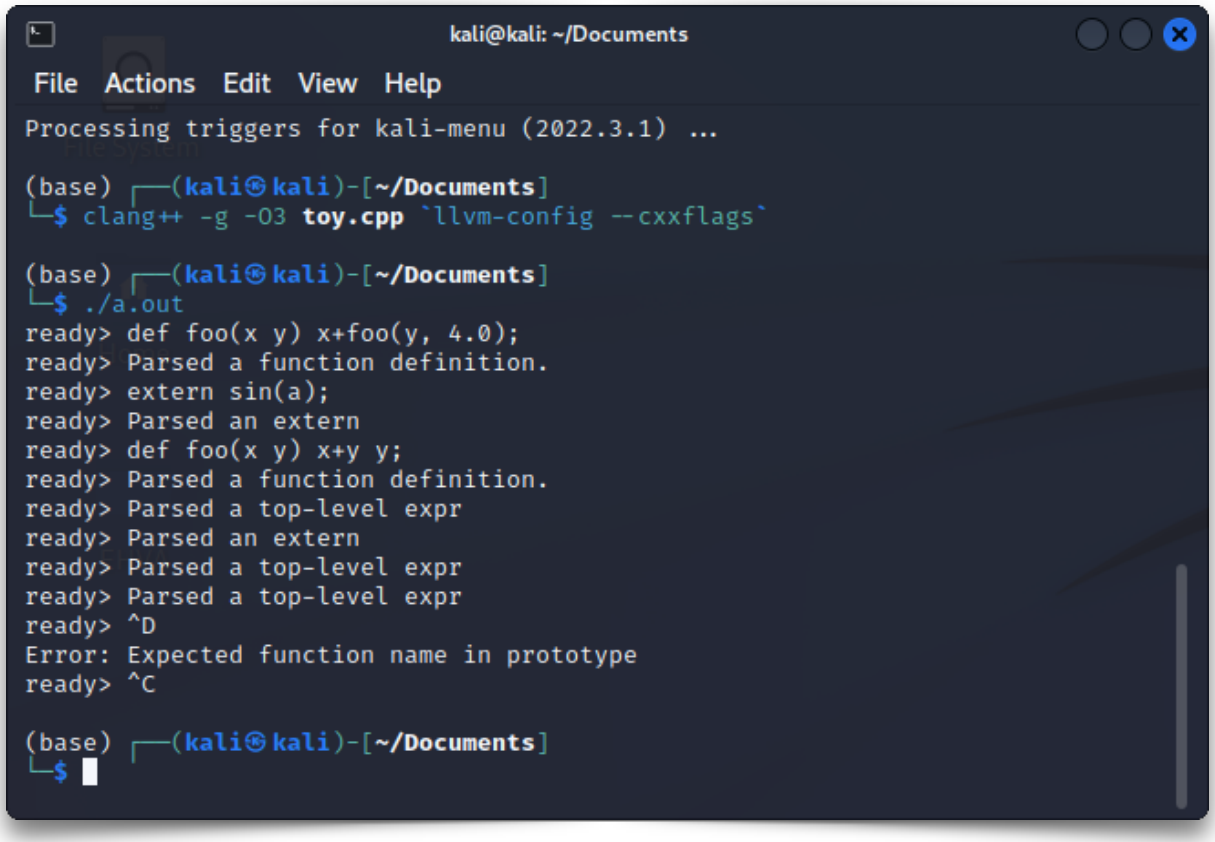
There is a lot of room for growth here. You can define new AST nodes, extend the language in a variety of ways, and so on. In the following section, we will show you how to generate LLVM Intermediate Representation (IR) from the AST.

2.7 Full Code Listing

The complete code for our running example is shown below. We need to link in the LLVM libraries because this uses them. To accomplish this, we use the llvm-config tool to tell our makefile/command line which options to use:

```
# Compile
clang++ -g -O3 toy.cpp `llvm-config --cxxflags`
# Run
./a.out
```

Here is the code : gist on github

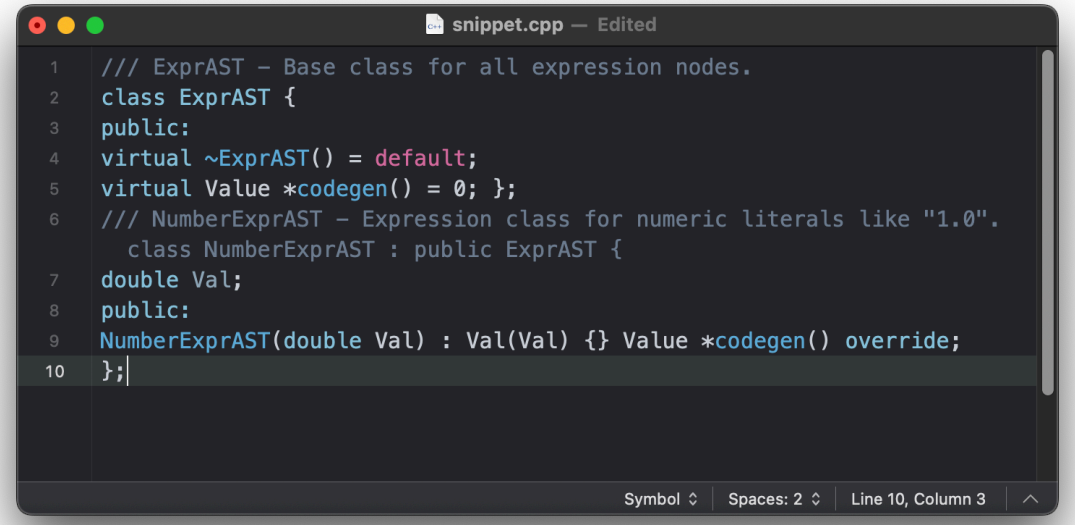


CHAPTER 3
CODE GENERATION TO LLVM IR

With the AST available, we demonstrate how simple it is to build LLVM IR and how to include LLVM into your project.

3.1 Code Generation Setup

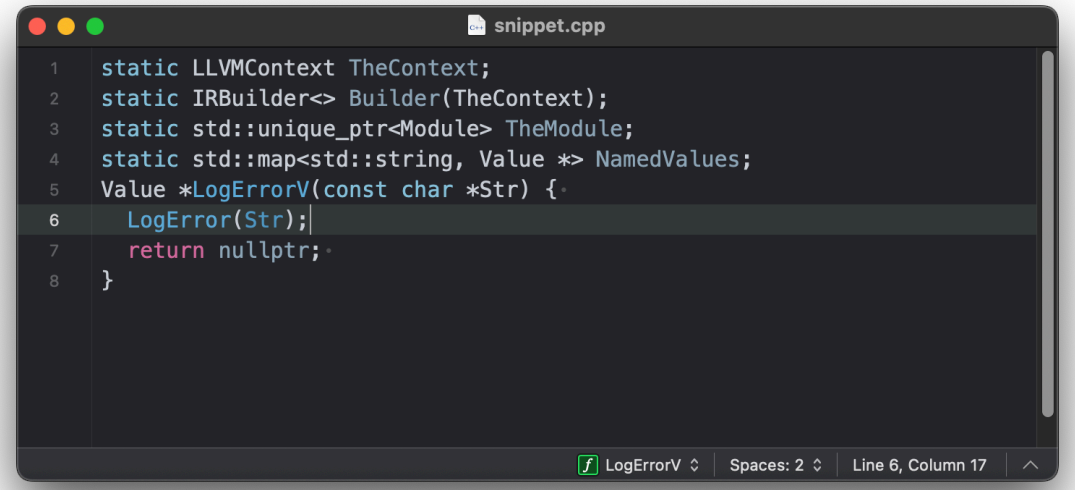
TO generate LLVM IR, we need to setup to get started. So, we define virtual code generation methods in each AST class as:



Here, codegen() method emits IR for AST along with all thing it depends on, and they all return an LLVM Value object. "Value" is the class used to represent a "Static Single Assignment (SSA) register" or "SSA value" in LLVM.

The second thing we want is a "LogError" method like we used for the parser, which will be

used to report errors found during code generation (for example, use of an undeclared parameter):

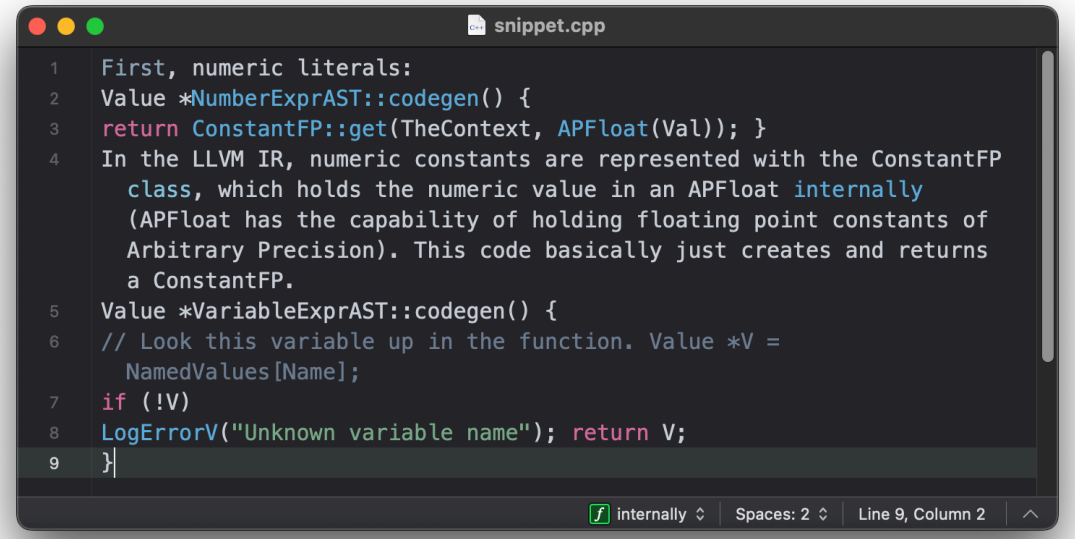


The static variables will be used to generate code. TheContext is an opaque object that owns many core LLVM data structures, including type and constant value tables. The Builder object is a utility object that simplifies the generation of LLVM instructions. TheModule is an LLVM component with functions and global variables. Because it will own the memory for all of the IR that we produce, the codegen() method returns a raw Value* rather than a unique ptr.

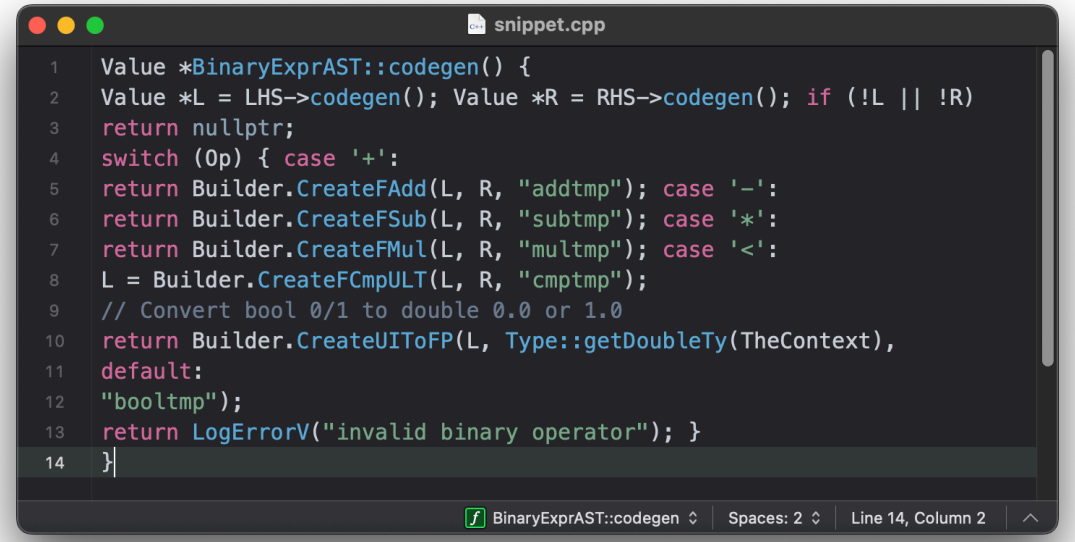
The NamedValues map records which values are defined in the current scope and their LLVM representation. (In other words, it is a code symbol table.) The only things that can be referenced in this form of Kaleidoscope are function parameters. As a result, when generating code for their function body, function parameters will be included in this map.

3.2 Expression Code Generation

Generating LLVM code for expression nodes is very straightforward: less than 45 lines of commented code.



In the simple version of Kaleidoscope, we assume that the variable has already been emitted somewhere and its value is available. In practice, the only values that can be in the NamedValues map are function arguments. This code simply checks to see that the specified name is in the map (if not, an unknown variable is being referenced) and returns the value for it.



The basic idea here is that we recursively emit code for the left-hand side of the expression, then the right-hand side, then we compute the result of the binary expression. IRBuilder knows where to insert the newly created instruction, all you have to do is specify what instruction to create (e.g. with CreateFAdd), which operands to use (L and R here) and optionally provide a name for the generated instruction. For example, if the code above emits

multiple “addtmp” variables, LLVM will automatically provide each one with an increasing, unique numeric suffix. Local value names for instructions are purely optional, but it makes it much easier to read the IR dumps.

LLVM instructions are constrained by strict rules: for example, the Left and Right operands of an add instruction must have the same type, and the result type of the add must match the operand types.

LLVM specifies that the fcmp instruction always returns an ‘i1’ value (a one bit integer). The problem with this is that Kaleidoscope wants the value to be a 0.0 or 1.0 value. In order to get these semantics, we combine the fcmp instruction with a uitofp instruction. This instruction converts its input integer into a floating point value by treating the input as an unsigned value. In contrast, if we used the sitofp instruction, the Kaleidoscope ‘<’ operator would return 0.0 and -1.0, depending on the input value.

snippet.cpp

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table. Function *CalleeF =
    TheModule->getFunction(Callee); if (!CalleeF)
  return LogErrorV("Unknown function referenced");
  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");
  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen()); if (!ArgsV.back())
    return nullptr;
  }
  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

CallExprAST::codegen Spaces: 2 Line 7, Column 30

The code above initially does a function name lookup in the LLVM Module’s symbol table. By giving each function the same name as what the user specifies, we can use the LLVM symbol table to resolve function names for us.

Once we have the function to call, we recursively codegen each argument that is to be passed in, and create an LLVM call instruction. LLVM uses the native C calling conventions by default, allowing these calls to also call into standard library functions like “sin” and “cos”, with no additional effort.

3.3 Function Code Generation

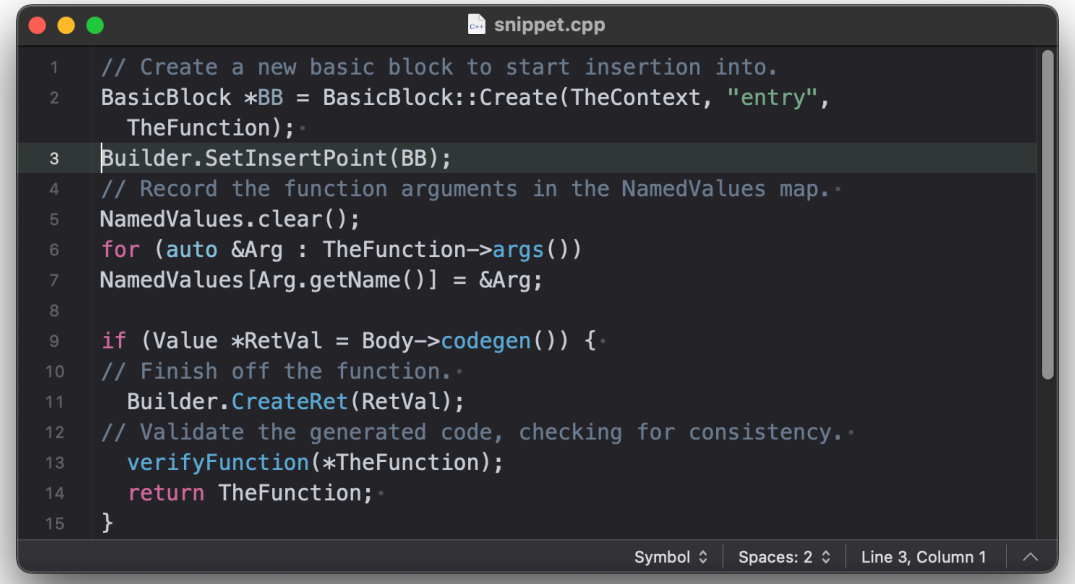
Code generation for prototypes and functions must handle a number of details. First, code generation for prototypes: they are used both for function bodies and external function declarations. The code is:

First that this function returns a "Function*" instead of a "Value*". Because a "prototype" is about the external interface for a function (not the value computed by an expression), it makes sense for it to return the LLVM Function it corresponds to when codegen'd.

The call to FunctionType::get creates the FunctionType that should be used for a given Prototype. Since all function arguments in Kaleidoscope are of type double, the first line creates a vector of "N" LLVM double types. It then uses the Functiontype::get method to create a function type that takes "N" doubles as arguments, returns one double as a result, and that is not vararg (the false parameter indicates this). The line creates the IR Function corresponding to the Prototype. This indicates the type, linkage and name to use, as well as which module to insert into. "external linkage" means that the function may be defined outside the current module and/or that it is callable by functions outside the module. The Name passed in is the name the user specified: since "TheModule" is specified, this name is registered in "TheModule"s symbol table.

Finally, we set the name of each of the function's arguments according to the names given in the Prototype, this step isn't strictly necessary.

For extern statements in Kaleidoscope, ne need to do this much. For function definitions however, we need to codegen and attach a function body.
For function definitions, we start by searching TheModule's symbol table for an existing version of this function, in case one has already been created using an 'extern' statement. If Module::getFunction returns null then no previous version exists, so we'll codegen one from the Prototype. In either case, we want to assert that the function is empty (i.e. has no body yet) before we start.



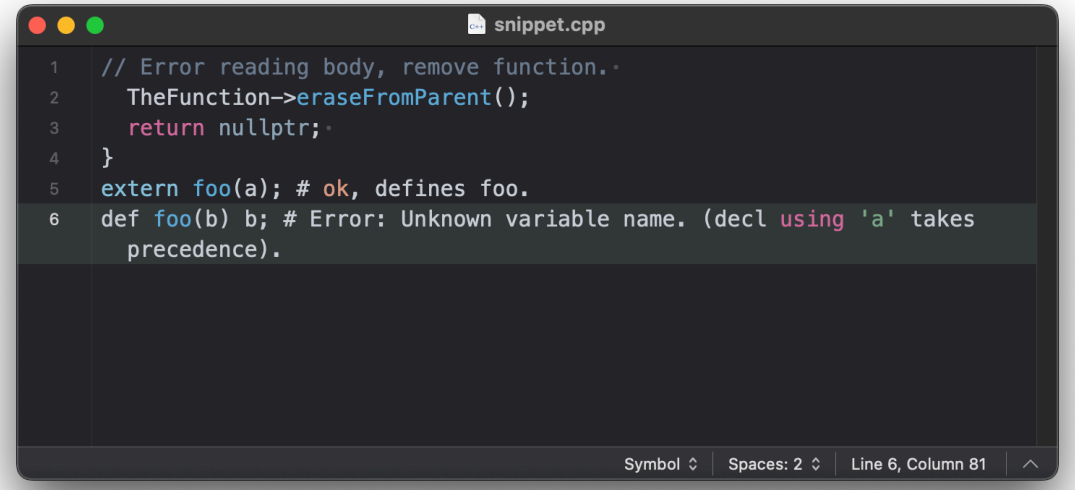
Now, the Builder is set up. The first line creates a new basic block (named "entry"), which is inserted into TheFunction. The second line then tells the builder that new

instructions should be inserted into the end of the new basic block. Basic blocks in LLVM are an important part of functions that define the Control Flow Graph.
Next we add the function arguments to the NamedValues map so that they're accessible to VariableExprAST nodes.

Once the insertion point has been set up and the NamedValues map populated, we call the codegen() method for the root expression of the function. If no error happens, this emits code to compute the expression into the entry block and returns the value that was computed. Assuming no error, we then create an LLVM ret instruction, which completes the function. Once the function is built, we call verifyFunction, which is provided by LLVM. This function does variety of checks on the generated code, to determine if our compiler is doing right.

Now, handling of the error case. We handle this by deleting the function we produced with the eraseFromParent method.

If the FunctionAST::codegen() method finds an existing IR Function, it does not validate its signature against the definition's own prototype. This means that an earlier 'extern' declaration will take precedence over the function definition's signature, which can cause codegen to fail, for instance if the function arguments are named differently. There are a number of ways to fix this bug. Here is a testcase:



3.4 Driver changes and closing thoughts

The sample code inserts calls to codegen into the "HandleDefinition", "HandleExtern" etc functions, and then dumps out the LLVM IR.

ready> 4+5;
Read top-level expression:
define double @0() {
entry:
ret double 9.000000e+00
}

The parser turns the top-level expression into anonymous functions for us. This will be handy when we add JIT support.

ready> def foo(a b) a*a + 2*a*b + b*b;

Read function definition:

define double @foo(double %a, double %b) { entry:

%multmp = fmul double %a, %a

%multmp1 = fmul double 2.000000e+00, %a %multmp2 = fmul double %multmp1, %b %addtmp = fadd double %multmp, %multmp2 %multmp3 = fmul double %b, %b

%addtmp4 = fadd double %addtmp, %multmp3

ret double %addtmp4 }

Further, we list the full code together. And add the required libraries.

References :

- LLVM official guide for language frontend building