LLVM:

Implementing a Language

Table of Contents

Tutorial Introduction

Welcome to the “Implementing a language with LLVM” tutorial. This tutorial runs through the implementation of a simple language, showing how fun and easy it can be. This tutorial will get you up and started as well as help to build a framework you can extend to other languages. The code in this tutorial can also be used as a playground to hack on other LLVM specific things.

The goal of this tutorial is to progressively unveil our language, describing how it is built up over time. This will let us cover a fairly broad range of language design and LLVM-specific usage issues, showing and explaining the code for it all along the way, without overwhelming you with tons of details up front.

It is useful to point out ahead of time that this tutorial is really about teaching compiler techniques and LLVM specifically, not about teaching modern and sane software engineering principles. In practice, this means that we’ll take a number of shortcuts to simplify the exposition. For example, the code leaks memory, uses global variables all over the place, doesn’t use nice design patterns like visitors, etc... but it is very simple. If you dig in and use the code as a basis for future projects, fixing these deficiencies shouldn’t be hard.

I’ve tried to put this tutorial together in a way that makes chapters easy to skip over if you are already familiar with or are uninterested in the various pieces. The structure of the tutorial is:

- **Chapter #1**: Introduction to the Kaleidoscope language, and the definition of its Lexer - This shows where we are going and the basic functionality that we want it to do. In order to make this tutorial maximally understandable and hackable, we choose to implement everything in C++ instead of using lexer and parser generators. LLVM obviously works just fine with such tools, feel free to use one if you prefer.
- **Chapter #2**: Implementing a Parser and AST - With the lexer in place, we can talk about parsing techniques and basic AST construction. This tutorial describes recursive descent parsing and operator precedence parsing. Nothing in Chapters 1 or 2 is LLVM-specific, the code doesn’t even link in LLVM at this point. :)
- **Chapter #3**: Code generation to LLVM IR - With the AST ready, we can show off how easy generation of LLVM IR really is.
- **Chapter #4**: Adding JIT and Optimizer Support - Because a lot of people are interested in using LLVM as a JIT, we’ll dive right into it and show you the 3 lines it takes to add JIT support. LLVM is also useful in many other ways, but this is one simple and “sexy” way to show off its power. :)
- **Chapter #5**: Extending the Language: Control Flow - With the language up and running, we show how to extend it with control flow operations (if/then/else and a ‘for’ loop). This gives us a chance to talk about simple SSA construction and control flow.
- **Chapter #6**: Extending the Language: User-defined Operators - This is a silly but fun chapter that talks about extending the language to let the user program define their own arbitrary unary and binary operators (with assignable precedence!). This lets us build a significant piece of the “language” as library routines.
- **Chapter #7**: Extending the Language: Mutable Variables - This chapter talks about adding user-defined local variables along with an assignment operator. The interesting part about this is how easy and trivial it is to construct SSA form in LLVM: no, LLVM does not require your front-end to construct SSA form!
- **Chapter #8**: Conclusion and other useful LLVM tidbits - This chapter wraps up the series by talking about potential ways to extend the language, but also includes a bunch of pointers to info about “special topics” like adding garbage collection support, exceptions, debugging, support for “spaghetti stacks”, and a bunch of other tips and tricks.

By the end of the tutorial, we’ll have written a bit less than 700 lines of non-comment, non-blank, lines of code. With this small amount of code, we’ll have built up a very reasonable compiler for a non-trivial language including a hand-written lexer, parser, AST, as well as code generation support with a JIT compiler. While other systems may have interesting “hello world” tutorials, I think the breadth of this tutorial is a great testament to the strengths of LLVM and why you should consider it if you’re interested in language or compiler design.

A note about this tutorial: we expect you to extend the language and play with it on your own. Take the code and go crazy hacking away at it, compilers don’t need to be scary creatures - it can be a lot of fun to play with languages!

Kaleidoscope Language and its Lexer

1.1 The Basic Language

This tutorial will be illustrated with a toy language that we'll call "Kaleidoscope" (derived from "meaning beautiful, form, and view"). Kaleidoscope is a procedural language that allows you to define functions, use conditionals, math, etc. Over the course of the tutorial, we'll extend Kaleidoscope to support the if/then/else construct, a for loop, user defined operators, JIT compilation with a simple command line interface, etc.

Because we want to keep things simple, the only datatype in Kaleidoscope is a 64-bit floating point type (aka `double` in C parlance). As such, all values are implicitly double precision and the language doesn't require type declarations. This gives the language a very nice and simple syntax. For example, the following simple example computes Fibonacci numbers:

```
# Compute the x'th fibonacci number.
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
```

We also allow Kaleidoscope to call into standard library functions (the LLVM JIT makes this completely trivial). This means that you can use the `extern` keyword to define a function before you use it (this is also useful for mutually recursive functions). For example:

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(.4), cos(42))
```

A more interesting example is included in Chapter 6 where we write a little Kaleidoscope application that displays a Mandelbrot Set at various levels of magnification.

Lets dive into the implementation of this language!

1.2 The Lexer

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a "lexer" (aka 'scanner') to break the input up into "tokens". Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities:

```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number
```

Each token returned by our lexer will either be one of the Token enum values or it will be an 'unknown' character like `+`, which is returned as its ASCII value. If the current token is an identifier, the IdentifierStr global variable holds the name of the identifier. If the current token is a numeric literal (like 1.0), NumVal holds its value. Note that we use global variables for simplicity, this is not the best choice for a real language implementation :).

The actual implementation of the lexer is a single function named gettok. The gettok function is called to return the next token from standard input. Its definition starts as:

```
/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();
```

gettok works by calling the C getchar() function to read characters one at a time from standard input. It eats them as it recognizes them and stores the last character read, but not processed, in LastChar. The first thing that it has to do is ignore whitespace between tokens. This is accomplished with the loop above.

The next thing gettok needs to do is recognize identifiers and specific keywords like `def`. Kaleidoscope does this with this simple loop:

```
if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
  IdentifierStr = LastChar;
  while (isalnum((LastChar = getchar())))
    IdentifierStr += LastChar;

  if (IdentifierStr == "def") return tok_def;
  if (IdentifierStr == "extern") return tok_extern;
  return tok_identifier;
}
```

Note that this code sets the `IdentifierStr` global whenever it lexes an identifier. Also, since language keywords are matched by the same loop, we handle them here inline. Numeric values are similar:

```
if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}
```

This is all pretty straight-forward code for processing input. When reading a numeric value from input, we use the C strtod