1. Kaleidoscope: Kaleidoscope Introduction and the Lexer

1.1. The Kaleidoscope Language

This tutorial is illustrated with a toy language called "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, debug info, etc.

We want to keep things simple, so 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 really easy. 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.

Documentation
- Getting Started/Tutorials
- User Guides
- Reference

Getting Involved
- Contributing to LLVM
- Submitting Bug Reports
- Mailing Lists
- IRC
- Meetups and Social Events

Additional Links
- FAQ
- Glossary
- Publications
- Github Repository

This Page
Show Source

Quick search
Go

Let's 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. We use global variables for simplicity, but 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 straightforward code for processing input. When reading a numeric value from input, we use the C `strtod` function to convert it to a numeric value that we store in `NumVal`. Note that this isn't doing sufficient error checking: it will incorrectly read "1.23.45.67" and handle it as if you typed in "1.23". Feel free to extend it! Next we handle comments:

```
if (LastChar == '#') {
  // Comment until end of line.
  do
    LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn't match one of the above cases, it is either an operator character like '+' or the end of the file. These are handled with this code:

```
  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}
```

With this, we have the complete lexer for the basic Kaleidoscope

language (the full code listing for the Lexer is available in the next chapter of the tutorial). Next we'll build a simple parser that uses this to build an Abstract Syntax Tree. When we have that, we'll include a driver so that you can use the lexer and parser together.

Next: Implementing a Parser and AST