The Parts of a Language

Scanning or Lexing or Lexical Analysis

Parsing

Static Analysis

Binding

Scope resolution

Type Checking (if the language is static i.e. not dynamic... WTH! Do you mean?)

Symbol Tables or attribues store information related to identifiers

Intermediate representations

IRs acts as an interface between the front end and back end of a compiler pipeline, this allows you to run the same code on different architectures, and avoid repetition of front end on each architecture.

Optimization

Until IRs the user program and its intention is completely understood which means it can be optimized or in other words the IR can be modified in a way that is more efficient, it the user was lousy in terms of performance; some common example would be a variable that is constantly being accessed so many times could be stored in a register rather than the main memory, or constant folding, which is evaluating a mathematical expression by replacing it with its actual value. There are many other perf optimization techniques like dead code elimination, loop unrolling, constant propagation etc.

Code Generation

Generating primitives like assembly instructions. This is the backend of the compiler. During this stage we hit a dilemma, which is to directly compile to CPU code or a virtual machine. Unfortunately it is too expensive and infeasible to directly compile to CPU these days as the architectures are byzantine and way too complex, and implementing one means the compiler or the language is restricted to that particular architecture. Or the alternative route that is usually taken these days is to produce a virtual machine code known as p-code (portable code), known as bytecode and it is what it literally sounds, i.e. each instruction is a byte long.

Conclusion

So a compiler is basically a pipeline where each stage transforms the given data structure making it easy to manage by the next step of the pipeline, i.e scanning helps parsing which helps in intermediate code generation (like bytecode, or registers) which helps in optimization and finally code generation.

**Representing Code**

Design Patterns

Metaprogramming

Formal Grammars and Automata

Difference between OOP / Functional

Context Free Grammars

For scanning we used *regular grammar*

Regular grammar can not handle nested expressions, this is where Context Free Grammars (CFG)

What is a formal Grammar?

A formal grammar takes some axioms or basic atomic pieces known as alphabets and the defines the set of valid possible strings in the grammar.

In the lexical grammar the atomic pieces were individual characters and the strings were the valid lexemes i.e. tokens.

For our syntactic grammar the atomic pieces are now the tokens, and the strings are sequence of tokens i.e. an expression.

Formal grammar’s job is like that of natural language grammar, i.e. specify what sentences are valid, but in this case expressions. i.e. in English saying something like “*i up sitting down am side”* is non-sense but something like “*i am sitting upside down”* although weird but is grammatically correct.

Rules of the grammar

A grammar consists of rules, a rule is a production as they generate strings. The generated strings are known as derivations, as they are derived from the rules of the grammar.

Each production in the cfg has a head i.e. its name, and a body, that describes what it generates. A body is a sequence of symbols, the symbols are either terminals or non terminals, think recursively, terminals are like base conditions, and non-terminals are a call to another rule/production in the grammar. Since recursion is in play, this entails infinite possible strings. The notation widely used to represent grammars is the BNF (Backus-Naur form).

Implementing Syntax Trees

Using julia’s macro to blurt out a file each time we run it.

The expression problem

In oop language it is easy to add new rows, simply define a new class, but adding a new operation i.e. a column means opening each of those classes and manually adding in the function. This approach works for tiny projects or languages, but for large projects this quickly becomes a chore and a hard to maintain code.

Functional languages, in which types and functions are totally distinct. And use pattern matching which is a switch statement to implement operation for each type all in one place. This makes it easy to add new operations for types, simply define another function that pattern matches on all of the types.

The Visitor pattern

The visitor pattern extends is used to extend or add in new methods to existing classes, it is basically approximating functional style in oop language, in function languages it is easy to add in new operations, unlike oop, which means opening each of the class and adding a new function which becomes hard when you have hundreds of classes, and each of those require an operation particularly suited to each specific class, this leads to hard to maintain code and does not scale well. Hence the visitor pattern can be used to tackle this problem down.

Parsing

Transforming the tokens into an AST structure that better resembles the grammar.

Precedence is maintained such that each rule matches to either expression of same precedence or higher, using C precedence for operators, with equalities having lowest precedence and Unary with highest precedence.

Since equality has the lowest precedence, matching that means we can go higher and higher to the highest precedence (feeling high).

Recursive descent is a top-down parser that starts at the lowest precedence expression and goes down (what!!!) to the highest precedence. It is a direct translation of the grammar to code, the functions are mutually recursive, where each rule is translated into a function, and the body of the rule also roughly translates to code e.g. | becomes if else or switch statements, \* indiciates a loop and ? indicates an if statement, a non-terminal is an invocation of another rule. And finally a terminal is code to match or consume the given token.