

From parsing to interpretation

Let's build a language

# What are we talking about?

We're going to be talking about programming languages.

# What are we talking about?

More specifically, we're going to be talking about interpreters.

# What are we talking about?

And even more specifically than that, we're going to talk about how one can take a sequence of characters that only a human could understand and make a computer understand them.

And why would we talk about that?

Well, we're Software Engineers and as Software Engineers we write a lot of code.

And why would we talk about that?

And how do we write that code? Well, with programming languages.

And why would we talk about that?

Programming languages are tools. Can you think of a tool that you use more often?

Most likely no.

# And why would we talk about that?

An understanding of programming languages and their implementation, even at a high level, will help you improve as a developer. Even if these skills are not used every day, the knowledge will stay with you and help you throughout your career.




So what are we going to do about it?


Let's build an interpreter

# What's that?

## Dictionary




# in·ter·pret·er

/in'təprədər/ 

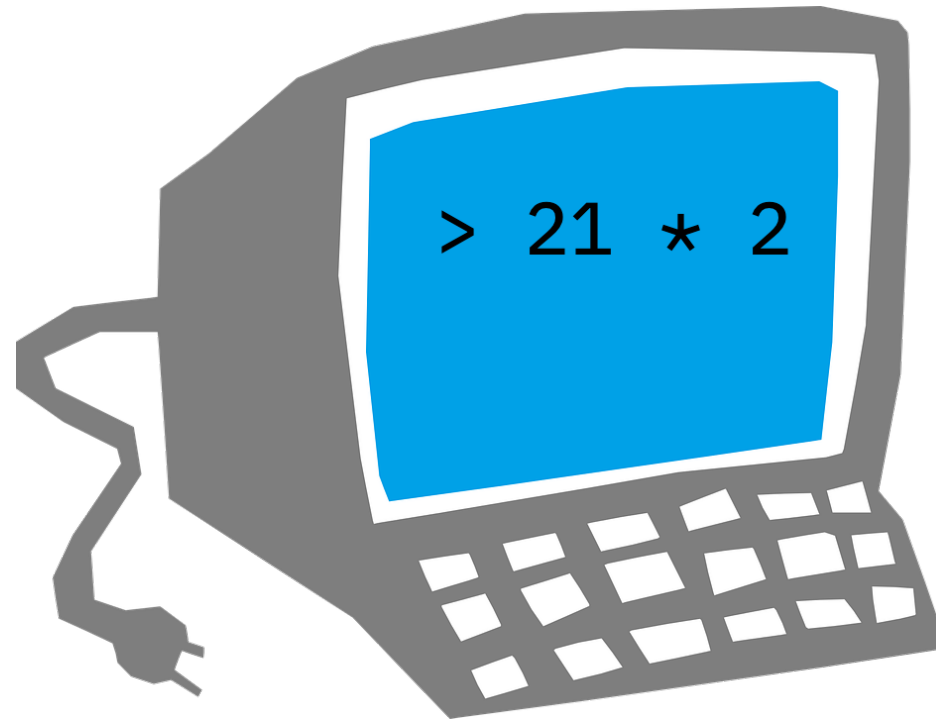
*noun*

a person who interprets, especially one who translates speech orally.  
*synonyms:* [translator](#), transcriber, transliterator [More](#)

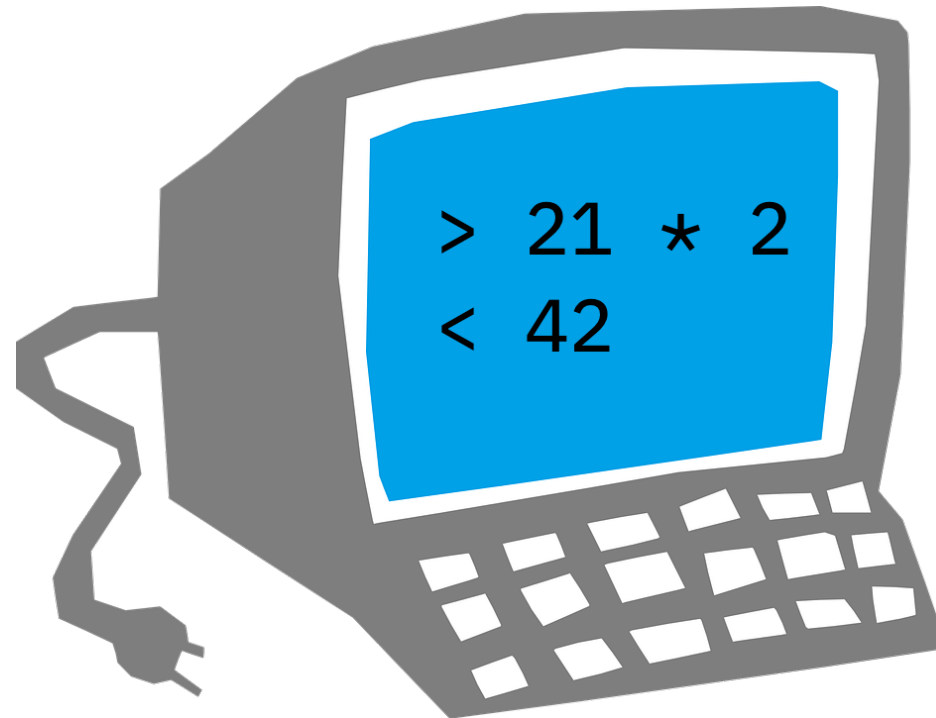
- **COMPUTING**  
a program that can analyze and execute a program line by line.

Translations, word origin, and more definitions

A program that can analyze a program



A program that can analyze a program



Where do we start?

How about with fancy buzzwords?

# Ohh, fancy.

- Grammars
- BNF/EBNF
- Lexer
- Parsers
- Parser generators
- Recursive descent parsers
- Scope
- Evaluation



# Where do we really start?

1 - Parse

2 - Evaluate

# This is where we start

- 1 - Define what our language looks like.
- 2 - Tokenize the input into a stream of valid tokens.
- 3 - Take the stream of tokens and compose them into complete expressions.
- 4 - Evaluate the expressions.

Let's define a language

What can our language do?

It can understand numbers

**7**

It can understand strings

`"Hello, world."`

It can understand something is true

#t

It can understand something is false

**#f**



It can run code conditionally

```
(cond (condition1 expression1)
      (condition2 expression2)
      (condition3 expression3)
      (condition4 expression4)
      (else default-expression))
```

It can express arithmetic operations

$(\ast \ 21 \ 2)$

It can define functions

```
(lambda (n) (* n 2))
```

It can apply functions to parameters

(double 21)

It can store all of those values

```
(define cool #t)
```

```
(define age 99)
```

```
(define name "Marcos")
```

```
(define double (lambda (n) (* n 2)))
```

# Does it look familiar?

Yes, it looks like a Lisp. Notice all of those parenthesized lists? Those are s-expressions and we'll be talking about them again soon.

Let's get a little more specific

Let's build a BNF grammar



# What's BNF?

Think of BNF as a language for languages. It's used in defining the structure in a computer language (and just not a programming language)

# What's BNF?

BNF is made up of rules and their expansions, such as:

`<expr> ::= <digit> "+" <digit>` where `<expr>` and `<digit>` are non-terminal symbols.

And terminal symbols: `<digit> ::= "1" | "2" | "3"`

Let's build an EBNF grammar

# What's EBNF?

EBNF is a set of extensions and modifications placed on top of BNF. Differences include dropping of the angled brackets,  $::=$  becomes  $=$ , and we add semicolons at the end of expressions.

Other improvements include the ability to repeat expressions with  $\{\}$ , group expressions with  $()$ , add optional expressions with  $[]$ , and explicit concatenation with  $,$ .

Some examples?

# Numbers

```
number = [ "-" ] , digit { digit } ;
```

```
digit = "0" | "1" | "2" | "3" | "4"  
      | "5" | "6" | "7" | "8" | "9" ;
```

# Strings

```
string = ' ' { letter } ' ' ;
```

```
letter = "A" | "B" | "C" | "D" | "E"  
        | "F" | "G" | "H" | "I" | "J"  
        | "K" | "L" | "M" | "N" | "O"  
        | "P" | "Q" | "R" | "S" | "T"  
        | "U" | "V" | "W" | "X" | "Y"  
        | "Z" | "a" | "b" | "c" | "d"  
        | "e" | "f" | "g" | "h" | "i"  
        | "j" | "k" | "l" | "m" | "n"  
        | "o" | "p" | "q" | "r" | "s"  
        | "t" | "u" | "v" | "w" | "x"  
        | "y" | "z" ;
```

# Booleans

```
boolean = "#t" | "#f" ;
```



# Identifiers

```
symbol = "<" | ">" | "*" | "+" | "-"  
        | "=" | "_" | "/" | "%" ;
```

```
identifier = ( letter | symbol ) ,  
             { letter | symbol | digit } ;
```

# S-expressions

```
sexpr = "(" { exprs } ")" ;
```

```
exprs = [ "'" ]  
        ( atom | sexpr | exprs ) ;
```

```
atom = identifier | number  
       | boolean | string ;
```

All together now. I present to you our Lisp.

```
main      = { exprs } ;
number    = [ "-" ] , digit { digit } ;
digit     = "0" | ... | "9" ;
string    = "'" , { letter } , "'" ;
letter    = "A" | ... | "z" ;
boolean   = "#t" | "#f" ;
identifier = ( letter | symbol ) ,
              { letter | symbol | digit } ;
symbol    = "<" | ">" | "*" | "+" | "-"
           | "=" | "_" | "/" | "%" ;
atom      = identifier | number
           | boolean | string ;
exprs     = [ "'" ]
           ( atom | sexpr | exprs ) ;
sexpr     = "(" { exprs } ")" ;
```

# What does this give us?

A reference for our ourselves or for a tool. A parser generator (like Yacc, GNU bison, ANTLR, etc.) could take our EBNF grammar and generate all of the code we need in order to parse our language.

But that's not what we're here for.

Let's build a parser

But wait!

Actually, let's take a step back. Characters are hard but what if we had 'words' instead? We need a lexer.

# What's a lexer?

Lexers analyze a string, character by character, and turn it into a series of tokens that can be used in the later steps of parsing.

|   |    |         |
|---|----|---------|
|   |    | OPAREN  |
|   |    | ID(+)   |
| ( | +  | NUM(21) |
|   | 21 | NUM(43) |
|   | 43 | CPAREN  |
| ) |    |         |

# Token types

```
sealed trait Token
```

```
case object SingleQuote extends Token
```

```
case object OpenParen extends Token
```

```
case object CloseParen extends Token
```

```
case object True extends Token
```

```
case object False extends Token
```

```
case class Number(value: Double)  
  extends Token
```

```
case class Str(value: String)  
  extends Token
```



## And even more tokens

```
case class InvalidToken(lexeme: String)  
  extends Token
```

```
case class Identifier(value: String)  
  extends Token
```

```
case class SExpr(values: List[Token])  
  extends Token
```

## Tokenizer function

```
def tokenize(str: String): Iterator[Token] = {  
  val src = str.toList.toIterator.buffered  
  for (c <- src if !c.isWhitespace)  
    yield c match {  
      // ...  
    }  
}
```

## Tokenizer function

```
def tokenize(str: String): Iterator[Token] = {  
  val src = str.toList.toIterator.buffered  
  for (c <- src if !c.isWhitespace)  
    yield c match {  
      case '(' => OpenParen  
      case ')' => CloseParen  
      case '\"' => SingleQuote  
      // ...  
    }  
}
```

## Tokenizer function

```
def tokenize(str: String): Iterator[Token] = {  
  val src = str.toList.toIterator.buffered  
  for (c <- src if !c.isWhitespace)  
    yield c match {  
      case '(' => OpenParen  
      case ')' => CloseParen  
      case '\"' => SingleQuote  
      case '\"' => ???  
      case n if isDigit(n) => ???  
      case c if isIdentifier(c) => ???  
      case '#' => ???  
      case c => ???  
    }  
}
```

# Tokenizing strings

```
val src = str.toList.toIterator.buffered

yield c match {
  case '"' =>
    Str(src.takeWhile(c => c != '"')
        .mkString)
}
```

## Tokenizing numbers

```
val src = str.toList.toIterator.buffered

yield c match {
  case n if isDigit(n) ||
    (n == '-' && isDigit(src.head)) =>

    val num =
      (n + consumeWhile(src, isDigit).mkString)

    Number(num.toDouble)
}
```

## Helper definitions

```
def isDigit(c: Char): Boolean =  
  c >= '0' && c <= '9'
```

```
def consumeWhile[T](  
  src: BufferedIterator[T],  
  predicate: T => Boolean  
): Iterator[T] = {  
  def aux(buff: List[T]): List[T] =  
    if (src.hasNext && predicate(src.head)) {  
      val curr = src.head  
      src.next ; aux(buff :+ curr)  
    } else buff  
  
  aux(List.empty).toIterator  
}
```

## Tokenizing identifiers

```
val src = str.toList.toIterator.buffered

yield c match {
  case c if isIdentifierStart(c) =>
    val name =
      c + consumeWhile(src, isIdentifier)

    Identifier(name.mkString)
}
```



## Helper definitions

```
def isIdentifierStart(c: Char): Boolean =  
    isLetter(c) || isSymbol(c)
```

```
def isIdentifier(c: Char): Boolean =  
    isDigit(c) || isLetter(c) || isSymbol(c)
```

```
def isLetter(c: Char): Boolean =  
    c >= 'A' && c <= 'z'
```

```
def isSymbol(c: Char): Boolean =  
    Set(  
        '<', '>', '*', '+', '-',  
        '=', '_', '/', '%'  
    ).contains(c)
```

## Tokenizing booleans

```
val src = str.toList.toIterator.buffered

yield c match {
  case '#' =>
    src.headOption match {
      case None =>
        InvalidToken("unexpected <eof>")
      case Some('f') => src.next; False
      case Some('t') => src.next; True
      case Some(c) =>
        src.next; InvalidToken(s"#$c")
    }
}
```

And now we have tokens

```
tokenize("(+ 21 43)").toList
```



```
List(  
  OpenParen,  
  Identifier(+),  
  Number(21.0),  
  Number(43.0),  
  CloseParen  
)
```

# Getting there

We nearly have a full representation of our grammar. So far we've covered cases the following cases: numbers, strings, booleans, and identifier. But we're still missing the structured expressions: s-expressions.

We need these

```
sexpr  = "(" { exprs } ")" ;
```

```
exprs  = [ "'" ]  
        ( atom | sexpr | exprs ) ;
```

```
atom   = identifier | number  
        | boolean | string ;
```

We need this

(+ 21 43)    ➡    OPAREN  
                 ID(+)  
                 NUM(21)    ➡    SEXPR(  
                 NUM(43)       ID(+),  
                 CPAREN       NUM(21),  
                                NUM(43))

# ASTs

An abstract syntax tree is a tree representation of source code structure. ASTs represent some tokens explicitly, like numbers, booleans, etc. and other implicitly, like parentheses and semicolons.

Let's extend our data structures to match that



# Implicit data

```
sealed trait Token  
case object SingleQuote extends Token  
case object OpenParen extends Token  
case object CloseParen extends Token
```

## Explicit data

```
sealed trait Expr extends Token
case object True extends Expr
case object False extends Expr
case class Number(value: Double) extends Expr
case class Str(value: String) extends Expr
case class Identifier(value: String)
  extends Expr
case class SExpr(values: List[Expr])
  extends Expr
```

## More expressions

```
case class Err(message: String) extends Expr
case class Quote(value: Expr) extends Expr
case class Lambda(args: List[Identifier],
  body: Expr) extends Expr
```

```
case class Proc(f: (List[Expr], Env)
  => (Expr, Env)) extends Expr
```

```
case class Builtin(f: (List[Expr], Env)
  => (Expr, Env)) extends Expr
```

## Parser function

```
def parse(ts: Iterator[Token]): Expr = {  
    val tokens = ts.buffered  
    tokens.next match {  
        // ...  
    }  
}
```

## Parser function

```
def parse(ts: Iterator[Token]): Expr = {  
  val tokens = ts.buffered  
  tokens.next match {  
    case SingleQuote => ???  
    case OpenParen => ???  
    case CloseParen => ???  
    case InvalidToken(lexeme) => ???  
    case expr => expr  
  }  
}
```

# Handling SingleQuote

```
tokens.next match {  
  case SingleQuote =>  
    if (tokens.hasNext)  
      Quote(parse(tokens))  
    else  
      Err("unexpected <eof>")  
}
```

# Handling OpenParen

```
tokens.next match {  
  case OpenParen =>  
    val values = parseExprs(tokens)  
  
    if (tokens.hasNext) {  
      tokens.next  
      SExpr(values)  
    } else Err("missing ') '")  
}
```

## Helper definitions

```
def parseExprs(  
    tokens: BufferedIterator[Token]  
): List[Expr] =  
  
    if (tokens.hasNext &&  
        tokens.head != CloseParen)  
        parse(tokens) :: parseExprs(tokens)  
    else  
        List.empty
```



## Handling CloseParen, InvalidToken, and everything else

```
tokens.next match {  
  case InvalidToken(lexeme) =>  
    Err(s"unexpected '$lexeme'")  
  
  case CloseParen =>  
    Err("unexpected ')'")  
  
  // True, False, Str, Number,  
  // Identifier, SExpr, Quote,  
  // Lambda, Builtin, Proc, Err  
  case expr => expr  
}
```

And now we have an AST

```
parse(tokenize("(+ 21 43)))
```



```
List(OpenParen, Identifier(+),  
      Number(21.0), Number(43.0),  
      CloseParen)
```



```
SExpr(List(Identifier(+),  
           Number(21.0),  
           Number(43.0)))
```

# Hey what about Lambda, Proc, and Builtin?

You may have noticed that our parser never returns Lambdas, Procs, or Builtins. There is a simple answer as to why Procs nor Builtins are returned, and that is because those are expressions that are meant to only be created programmatically, and as such the parser doesn't have to know how to parse them.

That is not the case of Lambdas.

This is what is happening right now

```
val code = "(lambda (x) (+ x x))"  
parse(tokenize(code))
```



```
SExpr(List(  
  Identifier(lambda),  
  SExpr(List(Identifier(x))),  
  SExpr(List(Identifier(+),  
              Identifier(x),  
              Identifier(x))))))
```

But this is what we need

```
val code = "(lambda (x) (+ x x))"  
parse(tokenize(code))
```



```
Lambda(List(Identifier(x)),  
        SExpr(List(Identifier(+),  
                    Identifier(x),  
                    Identifier(x))))
```

From this to that

```
SExpr(List(  
  Identifier(lambda),  
  SExpr(List(Identifier(x))),  
  SExpr(List(Identifier(+),  
             Identifier(x),  
             Identifier(x))))))
```



```
Lambda(List(Identifier(x)),  
        SExpr(List(Identifier(+),  
                    Identifier(x),  
                    Identifier(x))))
```

def passLambdas

```
def passLambdas(expr: Expr): Expr =  
  expr match {  
    // ...  
  }
```

```
def passLambdas
```

```
  expr match {  
    case SExpr(Identifier("lambda") ::  
                SExpr(args) ::  
                body ::  
                Nil) => ???  
  }
```



```
def passLambdas
```

```
  val (params, errs) = ???
```

```
  if (!errs.isEmpty)
```

```
    errs(0)
```

```
  else
```

```
    Lambda(params, body)
```

```
def passLambdas
```

```
args.foldRight(  
  List[Identifier](),  
  List[Err]()  
) {  
  case (curr, (params, errs)) =>  
    curr match {  
      case id @ Identifier(_) =>  
        (id :: params, errs)  
  
      case x => (  
        params,  
        Err("bad argument") :: errs  
      )  
    }  
}
```

calling passLambdas

```
def parse(ts:Iterator[Token]):Expr = {  
  val tokens = ts.buffered  
  
  passLambdas(tokens.next match {  
    // ...  
  })  
}
```

# Lambdas!

```
val code = "(lambda (x) (+ x x))"  
parse(tokenize(code))
```



```
Lambda(List(Identifier(x)),  
        SExpr(List(Identifier(+),  
                    Identifier(x),  
                    Identifier(x))))
```

# Multiple passes

We could employ this method of checking and manipulating an expression after it is parsed and before being executed to do many things. In our case we are adding a new feature, Lambda expressions, but one could also do optimizations, type checking, and other static analysis checks.

# So close

So far our interpreter can do a lot. I can parse numbers, booleans, strings, s-expression, and it even knows about lambdas! But still, it doesn't run any code.

Let's build an evaluator

## def evaluate

In its simplest form, an evaluator is a function that takes an expression and returns another expression. The returned expression can be thought of as the simplified version of the original.



Evaluate this!

324 ➡ 324

#t ➡ #t

"Hello, world." ➡ "Hello, world."

(+ 21 43) ➡ 64

((lambda (x)  
 (add x 20)) 22) ➡ 42

def evaluate

```
def evaluate(expr: Expr, env: Env):  
    (Expr, Env) =  
  
    expr match {  
        // ...  
    }
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case expr @ (True | False  
      | _: Str | _: Number  
      | _: Quote | _: Lambda  
      | _: Builtin | _: Proc  
      | _: Err  
    ) =>  
  
    (expr, env)  
  }
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case id @ Identifier(name) =>  
      val err = Err(  
        s"unbound variable: $name")  
  
      (env.getOrElse(err), env)  
  }
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case SExpr(Nil) =>  
      (Err("empty expression"), env)  
  }
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case SExpr((id @ Identifier(_))  
      :: body) =>  
  
    val (head, _) =  
      evaluate(id, env)  
  
    evaluate(  
      SExpr(head :: body),  
      env)  
  }
```

```
def evaluate

case SExpr(Lambda(args, body)
  :: values) =>

  val scope = args.zip(values)
    .foldLeft(env) {
      case (env, (arg, value)) =>
        env ++ Map(arg -> value)
    }

  val (ret, _) =
    evaluate(body, scope)

  (ret, env)
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case SExpr(Proc(fn) :: args) =>  
      val evaled = args.map {  
        arg => evaluate(arg, env)._1  
      }  
  
      fn(evaled)  
  }
```



def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case SExpr(Builtin(fn) :: args) =>  
      fn(args, env)  
  }
```

def evaluate

```
def evaluate(expr: Expr, env: Env):  
  (Expr, Env) =  
  
  expr match {  
    case SExpr(head :: _) =>  
      val err = Err(  
        s"cannot call $head")  
  
      (err, env)  
  }
```

# That's all for evaluate

You may have noticed our evaluate function was missing some functionality. What happened to conditionals? What about variable bindings?

This is what Proc and Builtin are for

## Builtin: define

```
Builtin((args, env) => args match {  
  case (id @ Identifier(_))  
    :: expr :: Nil =>  
  
    val update = env ++  
      Map(id -> expr)  
  
    (expr update)  
  
  case _ =>  
    (Err("bad call to define"), env)  
})
```

## Builtin: cond

```
Builtin((args, env) => {  
  def aux(conds: List[Expr]): Expr =  
    // ...  
  
    (aux(args), env)  
}),
```

## Builtin: cond

```
def aux(conds: List[Expr]): Expr =  
  conds match {  
    case SExpr(check :: body :: Nil)  
      :: rest => ???  
  
    case Nil => SExpr(List.empty)  
    case _ => Err("bad syntax. cond")  
  }
```

## Builtin: cond

```
def aux(conds: List[Expr]): Expr =  
  conds match {  
    case SExpr(check :: body :: Nil)  
      :: rest =>  
  
    evaluate(check, env)._1 match {  
      case False => aux(rest)  
      case _ =>  
        evaluate(body, env)._1  
    }  
  
    case Nil => SExpr(List.empty)  
    case _ => Err("bad syntax. cond")  
  }
```



## Builtin: add

```
Proc((args, env) =>
  (args match {
    case Number(a) :: Number(b)
      :: Nil =>

      Number(a + b)

    case _ => Err("bad call to add")
  }, env))
```

Let's test it out

def run

```
def run(code: String, env: Env):  
  Expr =  
  
  evaluate(parse(tokenize(code)),  
    env)._1
```

```
val code = """  
  ((lambda (x) (add x 20)) 22)  
  """
```

```
val env = Map(  
  Identifier("add") -> builtinAdd  
)
```

```
run(code, env)
```



Number(42.0)

From parsing to interpretation

We've built a language