2019-05-06-adl-language-tools (/github/jpivarski/2019-05-06-adl-language-tools/tree/master)
/ 01-overview.ipynb (/github/jpivarski/2019-05-06-adl-language-tools/tree/master/01-overview.ipynb)

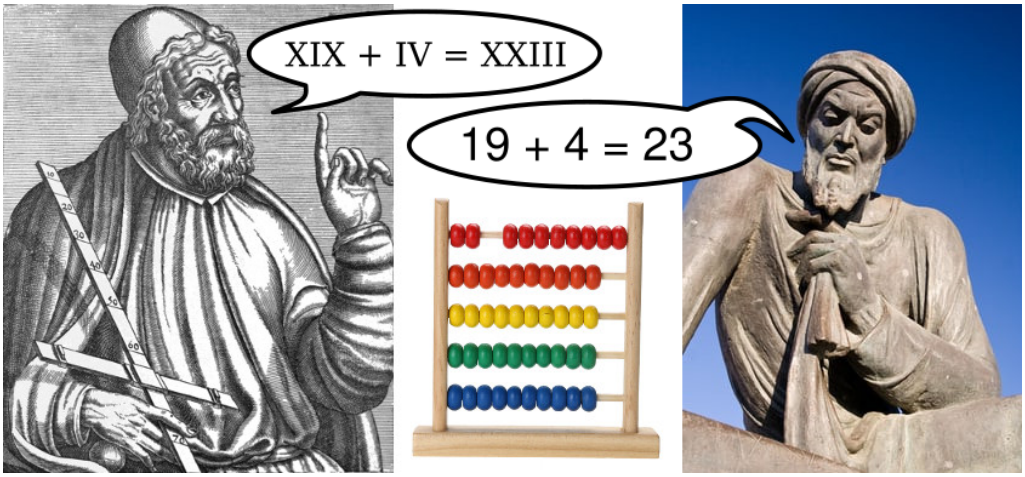
How to build your own language

We tend to engage in magical thinking when it comes to programming: we say that the program (a written text) *does* X, Y, or Z.

A computer is a machine representing mathematics, controlled by changing its state, like an abacus.

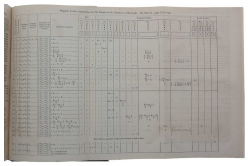
A programming language *is a human language* that describes what happens when we push the beans.

Some languages are better at describing the state of the beans than others.



Modern programming languages are attached to the device in such a way that they push the beans for us—that's where the confusion comes from. The distinction between talking about the beans and moving the beans is hidden. It wasn't always so.

Ada Lovelace's algorithm for computing Bernoulli numbers was written for a computer that never ended up being invented.



John McCarthy, creator of Lisp: "This EVAL was written and published in the paper and Steve Russell said, 'Look, why don't I program this EVAL?' and I said to him, 'Ho, ho, you're confusing theory with practice—this EVAL is intended for reading, not for computing!' But he went ahead and did it." *(Talk at MIT in 1974, published here (https://dl.acm.org/citation.cfm?id=802047&dl=ACM&coll=DL).)*

APL (ancestor of MATLAB, R, and Numpy) was also a notation for understanding programs years before it was executable. The book was named A Programming Language (http://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE

We're here to develop better languages for describing what we do in physics analysis.

Despite the physics focus of our goals, the techniques for creating languages are standard. This tutorial will introduce specific tools in Python along with their general concepts.

1. *(this introduction)*
2. Language basics: parsing and interpreting (02-parsers-and-interpreters.ipynb)
3. Compiling and transpiling to another language (03-compiling-and-transpiling.ipynb)
4. Type checking: proving program correctness (04-type-checking.ipynb)

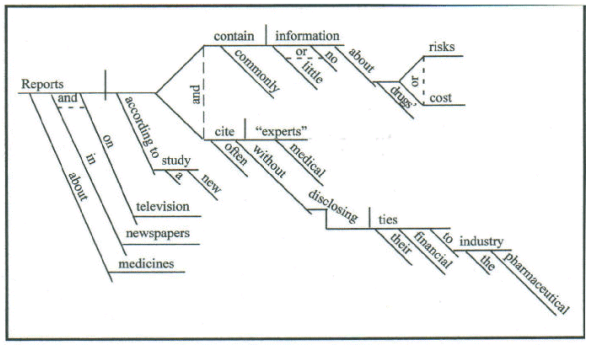
5. Embedded DSLs: using an existing language's syntax (05-embedded-dsl.ipynb)

Click on the tutorials above or choose from the left sidebar.

2019-05-06-adl-language-tools (/github/jpivarski/2019-05-06-adl-language-tools/tree/master)
/
02-parsers-and-interpreters.ipynb (/github/jpivarski/2019-05-06-adl-language-tools/tree/master/02-parsers-and-interpreters.ipynb)

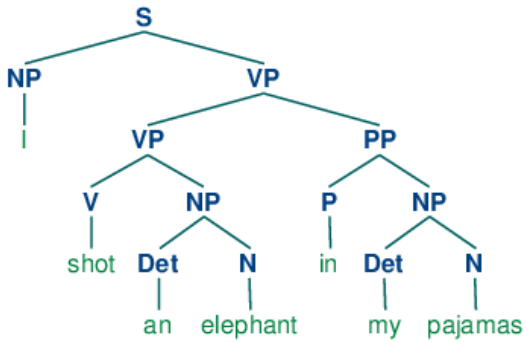
Language basics: parsing and interpreting

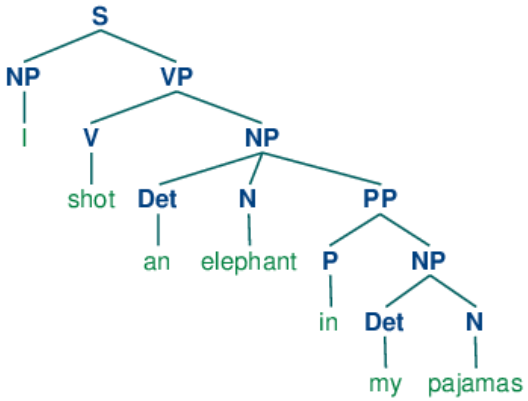
Parsing is the conversion of source code text into a tree representing relationships among tokens (words & symbols).



Reports about medicines in newspapers and on television commonly contain little or no information about drugs' risks and cost, and often cite medical "experts" without disclosing their financial ties to the pharmaceutical industry, according to a new study.

- Susan Okie, The Washington Post (published on June 1, 2000, in Louisville, KY, in The Courier-Journal, page A3)

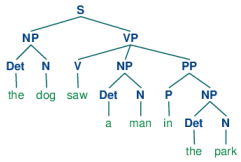


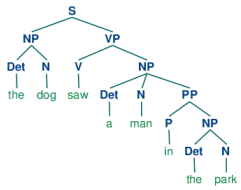


"How he got into my pajamas, I'll never know." — Groucho Marx

Grammar: a list of rules to convert tokens into trees and trees into bigger trees.

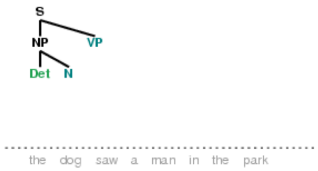
```
sentence (S):              noun_phrase verb_phrase
prepositional_phrase (PP): preposition noun_phrase
verb_phrase (VP):          verb noun_phrase | verb noun_phrase preposi
tional_phrase
noun_phrase (NP):          "John" | "Mary" | "Bob"
                           | determiner noun | determiner noun preposi

tional_phrase
preposition (P):           "in" | "on" | "by" | "with"
verb (V):                  "saw" | "ate" | "walked"
determiner (Det):          "a" | "an" | "the" | "my"
noun (N):                  "man" | "dog" | "cat" | "telescope" | "par
k"
```



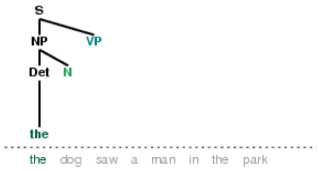


1. Initial stage

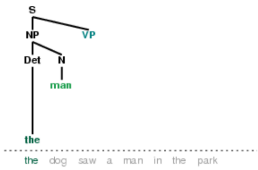
2. Second production



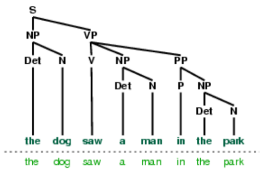
3. Matching *the*



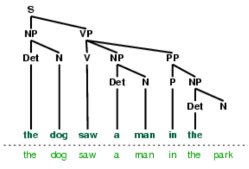
4. Cannot match *man*

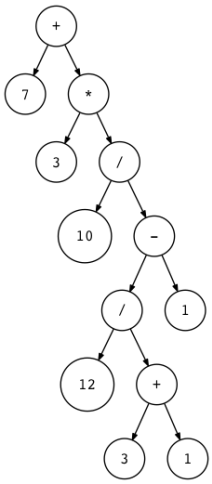


5. Completed parse



6. Backtracking





Mathematical expressions and computer programs can be parsed the same way.

```
7 + 3 * (10 / (12 / (3 + 1) - 1))
```

We definitely don't need to write the parsing algorithm—decades of computer science research has already gone into that.

I used to have a favorite (PLY), but while preparing this demo, I found a better one (Lark).

Let's get started!

Lark - a modern parsing library for Python

Parse any context-free grammar, FAST and EASY!

Beginners: Lark is not just another parser. It can parse any grammar you throw at it, no matter how complicated or ambiguous, and do so efficiently. It also constructs a parse-tree for you, without additional code on your part.

Experts: Lark implements both Earley(SPPF) and LALR(1), and several different lexers, so you can trade-off power and speed, according to your requirements. It also provides a variety of sophisticated features and utilities.

Lark can:

- Parse all context-free grammars, and handle any ambiguity
- Build a parse-tree automagically, no construction code required
- Outperform all other Python libraries when using LALR(1) (Yes, including PLY)
- Run on every Python interpreter (it's pure-python)
- Generate a stand-alone parser (for LALR(1) grammars)

In [1]:
```
import lark

expression_grammar = """
arith:   term   | term "+" arith  -> add | term "-" arith      -> sub
term:    factor | factor "*" term -> mul | factor "/" term     -> div
factor:  pow    | "+" factor      -> pos | "-" factor          -> neg
pow:     call ["**" factor]
call:    atom   | call trailer
atom:    "(" expression ")" | CNAME -> symbol | NUMBER -> literal
trailer: "(" arglist ")"
arglist: expression ("," expression)*

%import common.CNAME
%import common.NUMBER
%import common.WS
"""

grammar = "\n".join(["start: expression", "expression: arith", "%ignore WS"])

parser = lark.Lark(grammar)
```

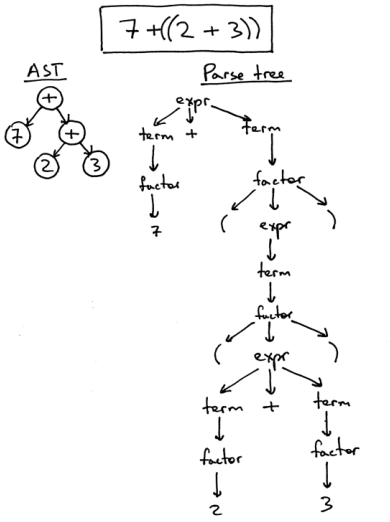
In [2]:
```
print(parser.parse("2 + 2").pretty())
```

```
start
  expression
    add
      term
        factor
          pow
            call
              literal   2
      term
        factor
          pow
            call
              literal   2
```

If the prepositional phrase "in my pajamas" had a well-defined operator precedence in "I shot an elephant in my pajamas," there would be no ambiguity.

Alternatively, if we could use parentheses ( ) in English to denote nesting, there would be no ambiguity.

Building the operator precedence into the grammar created a lot of superflous tree nodes, though.



The parsing tree has too much detail because it includes nodes for rules even if they were just used to set up operator precedence.

Let's reduce it to a tree that contains only what is necessary to understand the meaning of the program.

Such a tree is called an **Abstract Syntax Tree** (AST).

This is easy enough (and particular enough to our specific needs) that we should write it ourselves.

In [3]:

```
class AST:                                      # only three types (and a sup
    _fields = ()
    def __init__(self, *args, line=None):
        self.line = line
        for n, x in zip(self._fields, args):
            setattr(self, n, x)
            if self.line is None: self.line = x.line

class Literal(AST):                             # Literal: value that appears
    _fields = ("value",)
    def __str__(self): return str(self.value)

class Symbol(AST):                              # Symbol: value referenced by
    _fields = ("symbol",)
    def __str__(self): return self.symbol

class Call(AST):                                # Call: evaluate a function c
    _fields = ("function", "arguments")
    def __str__(self):
        return "{0}({1})".format(str(self.function), ", ".join(str(x) for x in
```

In [4]:
```
def toast(ptnode):  # Recursively convert parsing tree (PT) into abstract synt
    if ptnode.data in ("add", "sub", "mul", "div", "pos", "neg"):
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "pow" and len(ptnode.children) == 2:
        arguments = [toast(ptnode.children[0]), toast(ptnode.children[1])]
        return Call(Symbol("pow", line=arguments[0].line), arguments)
    elif ptnode.data == "call" and len(ptnode.children) == 2:
        return Call(toast(ptnode.children[0]), toast(ptnode.children[1]))
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]), line=ptnode.children[0].line)
    elif ptnode.data == "literal":
        return Literal(float(ptnode.children[0]), line=ptnode.children[0].line
    elif ptnode.data == "arglist":
        return [toast(x) for x in ptnode.children]
    else:
        return toast(ptnode.children[0])    # many other cases, all of them si

print(toast(parser.parse("2 + 2")))
```

```
add(2.0, 2.0)
```

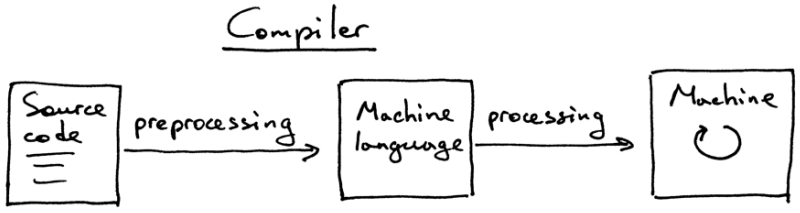
Execution

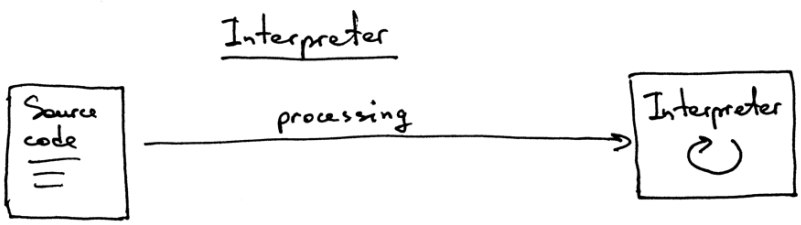
The simplest way to run a program is to repeatedly walk over the AST, evaluating each step. This is an **interpreter**.

Historical interlude:

- The first high-level programming language, Short Code (https://www.computer.org/csdl/magazine/an/1988/01/man1988010007/13rRUxCitB8) ("Short Order Code"), was an interpreter.
- Created by a physicist, John Mauchly, in 1949 for UNIVAC I.
- It ran 50× slower than the corresponding machine instructions.
- His company hired Grace Hopper, who improved the situation by inventing compilers (in particular, COBOL in 1959).

A **compiler** scans the AST to generate a sequence of machine instructions, natively recognized and executed by the computer.





In [5]:
```
def run(astnode, symbols):
    if isinstance(astnode, Literal):
        return astnode.value

    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]

    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        return function(*arguments)

import math, operator
symbols = {"add": operator.add, "sub": operator.sub, "mul": operator.mul, "di
           "pos": operator.pos, "neg": operator.neg, "pow": math.pow, "sqrt":
run(toast(parser.parse("2 + 2")), symbols)
```

Out[5]:
```
4.0
```

This is the pattern we will use for the rest of this tutorial:

- **Lark:** grammar → parsing tree
- **toast:** parsing tree → AST
- **interpreter:** AST + inputs → outputs

We will just be adding to the grammar, the AST, and the interpreter as we go.

Error handling

If a bad condition is encountered at runtime, like `sqrt(-5)`, the interpreter stops because the underlying Python execution engine raises an exception.

When writing a language, we must distinguish between our own internal errors and the users' logic mistakes. In the latter case, we have to let them know that they can fix it and provide a hint about where to start.

Line numbers are the most useful hint—but only when they're lines in the user's code, not the execution engine itself. The parser knows about line numbers—we must propagate that information into the AST (for an interpreter) and the final executable (for a compiler with debugging symbols included).

In [6]:

```
# We've already propagated line numbers from parsing tree tokens to all AST no
def showline(ast):
    if isinstance(ast, list):
        for x in ast:
            showline(x)
    if isinstance(ast, AST):
        print("{0:5s} {1:10s} {2}".format(str(ast.line), type(ast).__name__, a
        for n in ast._fields:
            showline(getattr(ast, n))

print("{0:5s} {1:10s} {2}".format("line", "AST type", "expression"))
print("-----------------------------------------------------------")
showline(toast(parser.parse("""sqrt(-5)""")))
```

```
line  AST type   expression
-----------------------------------------------------------
1     Call       sqrt(neg(5.0))
1     Symbol     sqrt
1     Call       neg(5.0)
1     Symbol     neg
1     Literal    5.0
```

In [7]:
```
# Short exercise: change the line below to report UserErrors with line numbers

class UserError(Exception): pass

def run(astnode, symbols):
    if isinstance(astnode, Literal):
        return astnode.value
    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]
    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        try:
            return function(*arguments)
        except Exception as err:
            raise err   # CHANGE THIS LINE

run(toast(parser.parse("""sqrt(-5)""")), {**operator.__dict__, **math.__dict__
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-c8ce6ea2c630> in <module>
     16             raise err   # CHANGE THIS LINE
     17
---> 18 run(toast(parser.parse("""sqrt(-5)""")), {**operator.__dict__, **math.

<ipython-input-7-c8ce6ea2c630> in run(astnode, symbols)
     14             return function(*arguments)
     15         except Exception as err:
---> 16             raise err   # CHANGE THIS LINE
     17
     18 run(toast(parser.parse("""sqrt(-5)""")), {**operator.__dict__, **math.

<ipython-input-7-c8ce6ea2c630> in run(astnode, symbols)
     12         arguments = [run(x, symbols) for x in astnode.arguments]
     13         try:
---> 14             return function(*arguments)
     15         except Exception as err:
     16             raise err   # CHANGE THIS LINE

ValueError: math domain error
```

Assignments

So far, all we've implemented is a calculator. As a next step, let's extend the language to include assignments.

For clarity, we'll use `:=` as an assignment operator.

These are our first **statements**, which do not compose as **expressions** do. Whereas expressions can be nested in parentheses like mathematical formulae, statements form a sequence that can only be nested with some block-syntax. We'll use curly brackets: `{...}` .

A block of statements could be used as an expression if it has a value. We'll use a common convention (among functional languages) in which the last statement of a block is an expression, and its value is the value of the block.

In [8]:
```
assignment_grammar = """
statements: NEWLINE* (assignment (NEWLINE | ";"))* expression NEWLINE*
assignment: CNAME ":=" expression
          | CNAME ":=" "{" statements "}"

%import common.WS_INLINE
%import common.NEWLINE

%ignore WS_INLINE
"""

grammar = "\n".join(["start: statements", "expression: arith"]
                    ) + expression_grammar + assignment_grammar
parser = lark.Lark(grammar)
```

In [9]:
```
print(parser.parse("""
x := 5
x
""").pretty())
```

```
start
  statements


    assignment
      x
      expression
        arith
          term
            factor
              pow
                call
                  literal       5


    expression
      arith
        term
          factor
            pow
              call
                symbol  x
```

In [10]:
```
class Block(AST):                                   # Block: sequence of statemer
    _fields = ("statements",)                       # (Doesn't have to be an AST
    def __str__(self):                              # function that returns its l
        return "{" + "; ".join(str(x) for x in self.statements) + "}"

class Assign(AST):                                  # Assign: put a value into a
    _fields = ("symbol", "value")
    def __str__(self):
        return "{0} := {1}".format(str(self.symbol), str(self.value))
```

In [11]:
```
def toast(ptnode):
    if ptnode.data == "statements":
        statements = [toast(x) for x in ptnode.children if x != "\n"]
        return Block(statements, line=statements[0].line)

    elif ptnode.data == "assignment":
        return Assign(str(ptnode.children[0]), toast(ptnode.children[1]), line

    ######################################## from this point onward, it's the
    elif ptnode.data in ("add", "sub", "mul", "div", "pos", "neg"):
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "pow" and len(ptnode.children) == 2:
        arguments = [toast(ptnode.children[0]), toast(ptnode.children[1])]
        return Call(Symbol("pow", line=arguments[0].line), arguments)
    elif ptnode.data == "call" and len(ptnode.children) == 2:
        return Call(toast(ptnode.children[0]), toast(ptnode.children[1]))
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]), line=ptnode.children[0].line)
    elif ptnode.data == "literal":
        return Literal(float(ptnode.children[0]), line=ptnode.children[0].line
    elif ptnode.data == "arglist":
        return [toast(x) for x in ptnode.children]
    else:
        return toast(ptnode.children[0])    # many other cases, all of them si
```

In [12]:
```
print(toast(parser.parse("x := 5; x")))
```

```
{x := 5.0; x}
```

In [13]:
```
# Short exercise: change toast so that x := {5} produces the same AST as x :=

def toast(ptnode):
    if ptnode.data == "statements":
        statements = [toast(x) for x in ptnode.children if x != "\n"]
        return Block(statements, line=statements[0].line)

    elif ptnode.data == "assignment":
        return Assign(str(ptnode.children[0]), toast(ptnode.children[1]), line

    ######################################## the change you need to make is /
    elif ptnode.data in ("add", "sub", "mul", "div", "pos", "neg"):
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "pow" and len(ptnode.children) == 2:
        arguments = [toast(ptnode.children[0]), toast(ptnode.children[1])]
        return Call(Symbol("pow", line=arguments[0].line), arguments)
    elif ptnode.data == "call" and len(ptnode.children) == 2:
        return Call(toast(ptnode.children[0]), toast(ptnode.children[1]))
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]), line=ptnode.children[0].line)
    elif ptnode.data == "literal":
        return Literal(float(ptnode.children[0]), line=ptnode.children[0].line
    elif ptnode.data == "arglist":
        return [toast(x) for x in ptnode.children]
    else:
        return toast(ptnode.children[0])    # many other cases, all of them si

print(toast(parser.parse("x := {5}; x")))
```

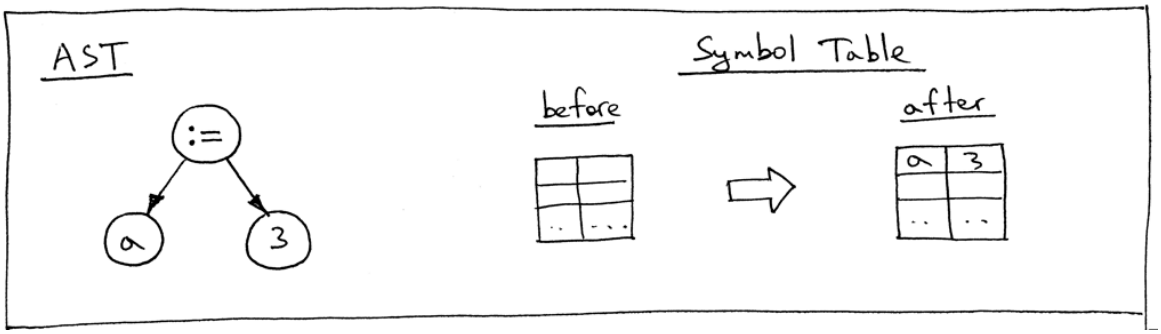
```
{x := {5.0}; x}
```

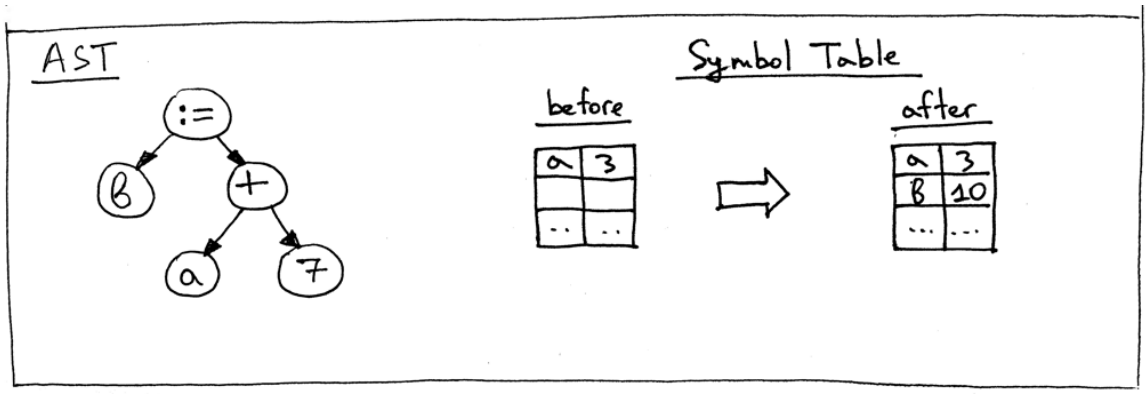
In [14]:
```
def run(astnode, symbols):
    if isinstance(astnode, Literal):
        return astnode.value
    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]
    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        return function(*arguments)
    elif isinstance(astnode, Block):
        for statement in astnode.statements:
            last = run(statement, symbols)
        return last
    elif isinstance(astnode, Assign):
        symbols[astnode.symbol] = run(astnode.value, symbols)

symbols = {**operator.__dict__, **math.__dict__}
run(toast(parser.parse("x := 5; x + 2")), symbols)
```

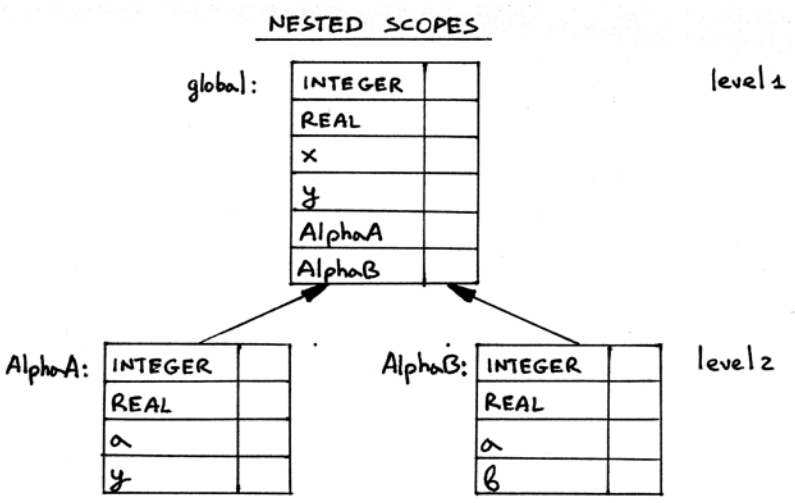
Out[14]:
```
7.0
```

In [15]:
```
# Should variables be accessible outside of the block where they're defined (l

symbols = {**operator.__dict__, **math.__dict__}
run(toast(parser.parse("x := {y := 5; y + 2}; y")), symbols)
```

Out[15]:
```
5.0
```


AST
:=
a
3
Symbol Table
before
after
a
3


AST
:=
B
+
a
7
Symbol Table
before
after
a
3
B
10


NESTED SCOPES
global:
INTEGER
REAL
x
y
AlphaA
AlphaB
level 1
AlphaA:
INTEGER
REAL
a
y
AlphaB:
INTEGER
REAL
a
B
level 2

In [16]:
```
class SymbolTable:
    def __init__(self, parent=None, **symbols):
        self.parent = parent
        self.symbols = symbols

    def __getitem__(self, symbol):
        if symbol in self.symbols:
            return self.symbols[symbol]
        elif self.parent is not None:
            return self.parent[symbol]
        else:
            raise KeyError(symbol)

    def __setitem__(self, symbol, value):
        self.symbols[symbol] = value

builtins = SymbolTable(**{**operator.__dict__, **math.__dict__})
```

In [17]:
```
# Short exercise: change the following so that scopes are nested.
# Bonus: what SHOULD reassigning a variable from a parent's scope do? Can you
def run(astnode, symbols):
    if isinstance(astnode, Literal):
        return astnode.value
    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]
    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        return function(*arguments)
    elif isinstance(astnode, Block):
        for statement in astnode.statements:
            last = run(statement, symbols)
        return last
    elif isinstance(astnode, Assign):
        symbols[astnode.symbol] = run(astnode.value, symbols)

run(toast(parser.parse("x := {y := 5; y + 2}; y")), SymbolTable(builtins))
```

Out[17]:
```
5.0
```

Branching

A calculator that can assign quantities is still just a calculator—though it may make the expressions easier to read.

The next level is to introduce control structures—if/then/else, while loops, and subroutines. For brevity, we'll just do if/then/else.

We could either allow variable definitions in then/else clauses to leak or we could always require an else clause and let the if/then/else block have a value. The former is a state-changing imperative language; the latter is functional (and easier to implement). We'll do the latter.

In [18]:

```
branching_grammar = """
block:      expression | "{" statements "}"
branch:     "if" expression "then" block "else" block

or:         and        | and "or" and
and:        not        | not "and" not
not:        comparison | "not" not -> not_test
comparison: arith | arith "==" arith -> eq | arith "!=" arith -> ne
                  | arith ">" arith -> gt  | arith ">=" arith -> ge
                  | arith "<" arith -> lt  | arith "<=" arith -> le
"""

grammar = "\n".join(["start: statements", "expression: or | branch"]
                    ) + expression_grammar + assignment_grammar + branching_gra
parser = lark.Lark(grammar)
```

In [19]:
```
print(parser.parse("if x > 0 then 1 else -1").pretty())
```

```
start
  statements
    expression
      branch
        expression
          or
            and
              not
                gt
                  arith
                    term
                      factor
                        pow
                          call
                            symbol      x
                  arith
                    term
                      factor
                        pow
                          call
                            literal     0
        block
          expression
            or
              and
                not
                  comparison
                    arith
                      term
                        factor
                          pow
                            call
                              literal   1
        block
          expression
            or
              and
                not
                  comparison
                    arith
                      term
                        neg
                          factor
                            pow
                              call
                                literal 1
```

In [20]:
```
def toast(ptnode):
    if ptnode.data == "branch":
        predicate, consequent, alternate = [toast(x) for x in ptnode.children]
        return Call(Symbol("if", line=predicate.line), [predicate, consequent,

    elif ptnode.data in ("or", "and", "eq", "ne", "gt", "ge", "lt", "le") and
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument

    elif ptnode.data == "not_test":
        argument = toast(ptnode.children[0])
        return Call(Symbol("not", line=argument.line), [argument])

    ######################################## from this point onward, it's the
    elif ptnode.data == "statements":
        statements = [toast(x) for x in ptnode.children if x != "\n"]
        if len(statements) == 1:
            return statements[0]
        else:
            return Block(statements, line=statements[0].line)
    elif ptnode.data == "assignment":
        return Assign(str(ptnode.children[0]), toast(ptnode.children[1]), line
    elif ptnode.data in ("add", "sub", "mul", "div", "pos", "neg"):
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "pow" and len(ptnode.children) == 2:
        arguments = [toast(ptnode.children[0]), toast(ptnode.children[1])]
        return Call(Symbol("pow", line=arguments[0].line), arguments)
    elif ptnode.data == "call" and len(ptnode.children) == 2:
        return Call(toast(ptnode.children[0]), toast(ptnode.children[1]))
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]), line=ptnode.children[0].line)
    elif ptnode.data == "literal":
        return Literal(float(ptnode.children[0]), line=ptnode.children[0].line
    elif ptnode.data == "arglist":
        return [toast(x) for x in ptnode.children]
    else:
        return toast(ptnode.children[0])    # many other cases, all of them si
```

In [21]:
```
print(toast(parser.parse("if not x == 0 or y > 1 and z <= 2 then 2 + 2 else 11
```

```
if(or(not(eq(x, 0.0)), and(gt(y, 1.0), le(z, 2.0))), add(2.0, 2.0), mul(111.0,
```

In [22]:
```
# The interpreter doesn't need any new features; all of these operators are ju

builtins["if"]  = lambda predicate, consequent, alternate: consequent if predi
builtins["or"]  = lambda p, q: p or q
builtins["and"] = lambda p, q: p and q
builtins["not"] = lambda p: not p

run(toast(parser.parse("if not x == 0 or y > 1 and z <= 2 then 2 + 2 else 111
    SymbolTable(builtins, x = 0, y = 10, z = 1))
```

Out[22]:
```
4.0
```

In [23]:
```
# However, both then/else clauses are evaluated, regardless of the predicate.
def show(x, y, f):
    print(f, x, y, f(x, y))
    return f(x, y)
builtins["add"] = lambda x, y: show(x, y, operator.add)
builtins["mul"] = lambda x, y: show(x, y, operator.mul)

run(toast(parser.parse("if x == x then 2 + 2 else 111 * 9")), SymbolTable(buil
```

```
<built-in function add> 2.0 2.0 4.0
<built-in function mul> 111.0 9.0 999.0
```

Out[23]:
```
4.0
```

In [24]:
```
# same for and/or, which are traditionally only evaluated fully if their resul

builtins["gt"] = lambda x, y: show(x, y, operator.gt)
builtins["lt"] = lambda x, y: show(x, y, operator.lt)

run(toast(parser.parse("x > 0 and x < 0")), SymbolTable(builtins, x = 0))
```

```
<built-in function gt> 0 0.0 False
<built-in function lt> 0 0.0 False
```

Out[24]:
```
False
```

Much like the decision that if/then/else would return a value, rather than changing state, decisions about order of evaluation has a subtle effect on what kinds of programs will be written in the language.

The and/or operators are left-right symmetric: we could

(1) evaluate `q` in `p and q` only if `p` is true, or

(2) evaluate `p` in `p and q` only if `q` is true (and equivalently for `p or q`).

Both are mathematically valid, but (2) would break every bash script on Earth.

In [25]:
```
# Short exercise: customize if-handling to only evaluate consequent or alterna

def run(astnode, symbols):
    if isinstance(astnode, Call) and astnode.function.symbol == "if":
        predicate  = run(astnode.arguments[0], symbols)
        consequent = run(astnode.arguments[1], symbols)
        alternate  = run(astnode.arguments[2], symbols)
        return consequent if predicate else alternate

    ######################################## the change you need to make is A
    elif isinstance(astnode, Literal):
        return astnode.value
    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]
    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        return function(*arguments)
    elif isinstance(astnode, Block):
        symboltable = SymbolTable(symbols)
        for statement in astnode.statements:
            last = run(statement, symboltable)
        return last
    elif isinstance(astnode, Assign):
        symbols[astnode.symbol] = run(astnode.value, symbols)

# only "add" or "mul" will be printed; not both
run(toast(parser.parse("if x == x then 2 + 2 else 111 * 9")), SymbolTable(buil
```

```
<built-in function add> 2.0 2.0 4.0
<built-in function mul> 111.0 9.0 999.0
```

Out[25]:
```
4.0
```

To evaluate only one of the two branches, we had to implement a special rule in the interpreter because the general rule is "evaluate all function arguments before evalutaing the function."

Some languages provide control over argument evaluation, such that users of the language can create control structures like if/then/else. Lisp has a general-purpose "quote" form:

```
(setq consequent (quote (+ 2 2)))                ; setq is assignment
(setq alternate  (quote (* 999 1)))
(if (> x 0) (eval consequent) (eval alternate))   ; eval is the opposit
```

that passes on the ASTs of `(+ 2 2)` and `(* 999 1)` to be evaluated later (selectively).

We can also do this in a language that allows functions to be defined and passed around as objects.

In [26]:
```
function_grammar = """
paramlist: CNAME | "(" ("," CNAME)* ")"
function: paramlist "=>" block
"""

grammar = "\n".join(["start: statements", "expression: or | branch | function"
                     ) + expression_grammar + assignment_grammar + branching_gra
parser = lark.Lark(grammar)
```

In [27]:
```
print(parser.parse("x => x**2").pretty())
```

```
start
  statements
    expression
      function
        paramlist       x
        block
          expression
            or
              and
                not
                  comparison
                    arith
                      term
                        factor
                          pow
                            call
                              symbol    x
                            factor
                              pow
                                call
                                  literal       2
```

In [28]:
```
class Function(AST):                                # Function: defines a new fur
    _fields = ("paramlist", "body")
    def __str__(self):
        return "({0}) => {1}".format(", ".join(self.paramlist), str(self.body)

def toast(ptnode):
    if ptnode.data == "function":
        paramlist = [str(x) for x in ptnode.children[0].children]
        body = toast(ptnode.children[1])
        return Function(paramlist, body, line=body.line)

    ######################################## from this point onward, it's the
    elif ptnode.data == "branch":
        predicate, consequent, alternate = [toast(x) for x in ptnode.children]
        return Call(Symbol("if", line=predicate.line), [predicate, consequent,
    elif ptnode.data in ("or", "and", "eq", "ne", "gt", "ge", "lt", "le") and
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "not_test":
        argument = toast(ptnode.children[0])
        return Call(Symbol("not", line=argument.line), [argument])
    elif ptnode.data == "statements":
        statements = [toast(x) for x in ptnode.children if x != "\n"]
        if len(statements) == 1:
            return statements[0]
        else:
            return Block(statements, line=statements[0].line)
    elif ptnode.data == "assignment":
        return Assign(str(ptnode.children[0]), toast(ptnode.children[1]), line
    elif ptnode.data in ("add", "sub", "mul", "div", "pos", "neg"):
        arguments = [toast(x) for x in ptnode.children]
        return Call(Symbol(str(ptnode.data), line=arguments[0].line), argument
    elif ptnode.data == "pow" and len(ptnode.children) == 2:
        arguments = [toast(ptnode.children[0]), toast(ptnode.children[1])]
        return Call(Symbol("pow", line=arguments[0].line), arguments)
    elif ptnode.data == "call" and len(ptnode.children) == 2:
        return Call(toast(ptnode.children[0]), toast(ptnode.children[1]))
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]), line=ptnode.children[0].line)
    elif ptnode.data == "literal":
        return Literal(float(ptnode.children[0]), line=ptnode.children[0].line
    elif ptnode.data == "arglist":
        return [toast(x) for x in ptnode.children]
    else:
        return toast(ptnode.children[0])    # many other cases, all of them si
```

In [29]:
```
print(toast(parser.parse("""
f := x => x**2
f(y)
""")))
```

```
{f := (x) => pow(x, 2.0); f(y)}
```

In [30]:
```
def run(astnode, symbols):
    if isinstance(astnode, Function):
        def function(*args):
            if len(args) != len(astnode.paramlist):
                raise UserError(astnode.line, "wrong number of arguments")
            symboltable = SymbolTable(symbols, **dict(zip(astnode.paramlist, a
            return run(astnode.body, symboltable)
        return function

    ######################################## from this point onward, it's the
    elif isinstance(astnode, Literal):
        return astnode.value
    elif isinstance(astnode, Symbol):
        return symbols[astnode.symbol]
    elif isinstance(astnode, Call):
        function = run(astnode.function, symbols)
        arguments = [run(x, symbols) for x in astnode.arguments]
        return function(*arguments)
    elif isinstance(astnode, Block):
        symboltable = SymbolTable(symbols)
        for statement in astnode.statements:
            last = run(statement, symboltable)
        return last
    elif isinstance(astnode, Assign):
        symbols[astnode.symbol] = run(astnode.value, symbols)
```

In [31]:
```
symboltable = SymbolTable(**{**operator.__dict__, **math.__dict__})

run(toast(parser.parse("""
f := x => 2*x
f(y)
""")), SymbolTable(symboltable, y = 5))
```

Out[31]:
```
10.0
```

In [32]:
```
# Now we can define "if" as a plain function that takes and calls zero-argumer
# to customize the order of evaluation.

symboltable = SymbolTable(**{**operator.__dict__, **math.__dict__})

symboltable["if"] = (lambda predicate, consequent, alternate:
                         consequent() if predicate else alternate())

symboltable["add"] = lambda x, y: show(x, y, operator.add)
symboltable["mul"] = lambda x, y: show(x, y, operator.mul)

run(toast(parser.parse("if x == x then () => 2 + 2 else () => 111 * 9")),
    SymbolTable(symboltable, x = 0))
```

```
<built-in function add> 2.0 2.0 4.0
```

Out[32]:
```
4.0
```

In [33]:
```
# Passing functions as arguments allows us to create control structures that c

symboltable["for"] = lambda n, f: [f(i) for i in range(int(n))]

run(toast(parser.parse("for(6, i => i**2)")), SymbolTable(symboltable))
```

Out[33]:
```
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0]
```

Recently, we have been talking about **Domain Specific Languages** (DSL) but referring to them as "declarative programming."

Declarative has to do with the **evaluation order** we've just seen.

- **Strictly Evaluated** languages evaluate expressions in lexical order (i.e. arguments before function calls).
- **Lazily Evaluated** languages eventually produce the same results but give the program more flexibility in deciding *when* or *where* the code will run. There are several kinds of objects representing a calculation that has not been executed:
  - An **AST** is the most powerful (Lisp's quote (https://www.gnu.org/software/emacs/manual/html_node/elisp/Quoting.html) and C#'s expression tree (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/index)): you can edit the code as data!
  - Passing **function objects** is the fundamental idea of functional programming—you don't need the language to have control structures because you can build them yourself.
  - **Promises/futures** are placeholders for calculations running elsewhere—another thread or another computer.
- **Declarative** languages produce the same output as strictly or lazily evaluated languages, but hide the distinction between them: *the order in which the code runs is an implementation detail*.

Examples of **declarative evaluation** include:

- rendering HTML: order determines placement, but not the sequence of graphics commands to draw it
- execution of SQL: user's queries are rewritten and optimized by query planner
- commands in a Makefile: only executed if targets are out of date
- cached function calls ("memoized"): function is only executed if not cached; e.g. an HTTP GET request
- machine code instructions in a CPU: modern processors sometimes engage in speculative execution (https://en.wikipedia.org/wiki/Out-of-order_execution).

One place we might consider declarative evaluation is in hiding the distinction between columnar and fused array operations (ask me later).

We have now seen all the essential elements of a programming language.

- **Parsing:** conversion of source code text into a **Parsing Tree**, then an **Abstract Syntax Tree** (AST).
- **Interpreter:** runs the program by walking over the AST at runtime.

- **Compiler:** converts the program to another language, such as machine instructions, and runs that.
- **Expressions:** nested elements of a mathematical formula.
  - **Literal:** values in the source code text.
  - **Symbol:** named reference to a value.
  - **Function Call:** evaluation of a function (including binary operations).
  - **Function Definition:** creation of a new function (named or unnamed).
- **Statements:** sequential elements defining a process.
  - **Assign:** creation or replacement of a named reference (possibly to a function).
- **Symbol Scope:** at which parts of a program symbols are bound to values (only considered **Lexical Scope**).
- **Evaluation Order:** temporal order in which expressions and statements are evaluated.
  - **Strict/Eager Evaluation:** the usual order; arguments of a function call before the function call.
  - **Lazy Evaluation:** pass unevaluated code (e.g. function definition) to let the called function decide when to evaluate.
  - **Declarative:** evaluation order is not visible to the programmer.

2019-05-06-adl-language-tools (/github/jpivarski/2019-05-06-adl-language-tools/tree/master)
/
03-compiling-and-transpiling.ipynb (/github/jpivarski/2019-05-06-adl-language-tools/tree/master/03-compiling-and-transpiling.ipynb)

Compiling and transpiling to another language

In the previous tutorial, we **interpreted** our little language, rather than **compiling** it.

The interpreter simply did in Python what was written in the new language—we filled the symbol table with functions like:

```
builtins["+"] = lambda x, y: x + y
```

Have we accomplished anything?

What we'll see in this tutorial is that a compiler isn't much more substantial than a translator, either. Ultimately, it just replaces each `+` AST node with the machine instruction for `+`.

Programming languages do not perform actions. They only express a user's intention in another layer of abstraction.

In our interpreter, there were 6 layers of abstraction:

transistor gates → machine code → C compiler → Python implementation (C code) → our interpreter → our little language

Interpreters vs compilers:

- An **interpreter** walks over the AST (or even parses the source code!) at runtime.
- A **compiler** serializes the AST into a state machine or a sequence of instructions, virtual or physical.
- A **transpiler** serializes the AST into code in another human-readable language. (Subjective: what's human-readable?)

Compilation targets:

- A **finite state machine** is a graph of executable steps *only* (not a full interpreter). Regular expressions are often compiled to finite state machines. A non-recurrent neural network is also a finite state machine.
- A **push-down machine** is a state machine with a stack of memory—parsers are almost always push-down machines.
- A **virtual machine** is a complete processor+memory driven by a sequence of instructions, like a physical computer, but implemented in software. Python and Java are not interpreters: they *compile* their source code to *virtual machines.*
- A **Von Neumann machine** is a physical computer driven by a sequence of instructions.

Other:

- **FPGA/ASIC:** physical computer consisting of raw gates, not instructions; Verilog isn't *compiled* like C, it's *synthesized*.

The general flow of a compiler is

linear (source code) → tree-like data structure (AST) → linear (instructions or other source code)

In this notebook, we'll write a transpiler, converting our little language into C++.

In [1]:
```
# First of all, did you know that you can do this?
import ROOT

ROOT.gInterpreter.Declare("""
double new_function(double x, double y) {
    return sqrt(x*x + y*y);
}""")

ROOT.new_function(3, 4)
```

```
Welcome to JupyROOT 6.17/01
```

Out[1]:
```
5.0
```

In [2]:
```
# And what about this?
import pycparser.c_parser, pycparser.c_generator    # pure Python C99 compiler/

c_parser = pycparser.c_parser.CParser()
ast = c_parser.parse("double f(double x) { return x*x; }")
ast.show()
```

```
FileAST:
  FuncDef:
    Decl: f, [], [], []
      FuncDecl:
        ParamList:
          Decl: x, [], [], []
            TypeDecl: x, []
              IdentifierType: ['double']
        TypeDecl: f, []
          IdentifierType: ['double']
    Compound:
      Return:
        BinaryOp: *
          ID: x
          ID: x
```

In [3]:
```
c_generator = pycparser.c_generator.CGenerator()

ast = c_parser.parse("double f(double x) { return x*x; }")

print(c_generator.visit(ast))
```

```
double f(double x)
{
  return x * x;
}
```

We can compile and run C++ code in ROOT (Cling) and we have a general C99 AST in a Python library (pycparser). The compilation chain could look like this:

our source language → our AST → C99 AST → C++ source code → compile and run in ROOT

We can compile and run C++ code in ROOT (Cling) and we have a general C99 AST in a Python library (pycparser). The compilation chain could look like this:

our source language → our AST → C99 AST → C++ source code → compile and run in ROOT

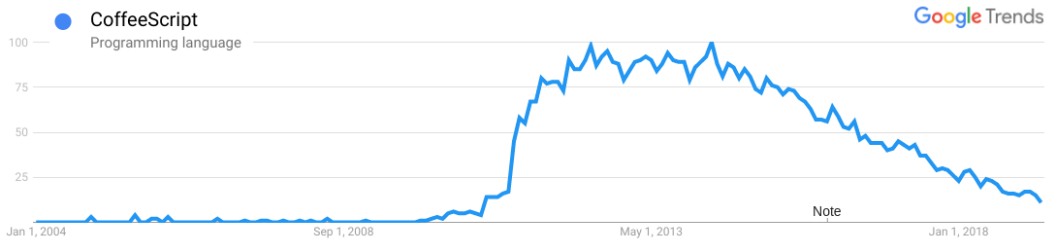
Why not output C++ strings directly from our AST?

We can compile and run C++ code in ROOT (Cling) and we have a general C99 AST in a Python library (pycparser). The compilation chain could look like this:

our source language → our AST → C99 AST → C++ source code → compile and run in ROOT

Why not output C++ strings directly from our AST?

It's hard to compose source code strings properly and it's hard to debug them. (Coffeescript famously skipped this step. (https://www.kickstarter.com/projects/michaelficarra/make-a-better-coffeescript-compiler) :)



In [4]:
```
import lark        # a grammar for Dirac's bra-ket notation!
grammar = """
start:  term
term:   factor [term]
factor: bra "|" ket | bra "|" operators "|" ket
bra:    "<" value
ket:    value ">"

operators: operator [operators]
operator: spinor | operator "*" -> conjugate
spinor:   "σ₁" -> s1 | "sigma1" -> s1 | "σ₂" -> s2 | "sigma2" -> s2 | "σ₃" -> s

value:   CNAME | complex complex
complex: NUMBER -> real | NUMBER "i" -> imag | "i" -> imag1
       | NUMBER "+" NUMBER "i" -> complex | NUMBER "+" "i" -> complex1

%import common.CNAME
%import common.NUMBER
%import common.WS
%ignore WS
"""
parser = lark.Lark(grammar)
```

In [5]:
```
# the cute thing about bra-ket notation is that it turns type errors into synta
print(parser.parse("<0 1| σ₁* σ₂ |0 1>").pretty())
```

```
start
  term
    factor
      bra
        value
          real  0
          real  1
      operators
        conjugate
          operator
            s1
        operators
          operator
            s2
      ket
        value
          real  0
          real  1
```

In [6]:

```
# I said that it's unwise to turn our AST directly into strings of the output l
# That's because strings are hard to COMPOSE. Don't avoid using strings to crea

from pycparser.c_ast import *

def c_ast(c_code):
    return c_parser.parse("void f() {" + c_code + ";}").ext[0].body.block_items

ast = c_ast("x * y")
ast.right.name = "z"
ast
```

Out[6]:

```
BinaryOp(op='*',
         left=ID(name='x'
                 ),
         right=ID(name='z'
                  )
         )
```

In [7]:

```
def mul(x, y): return BinaryOp("*", x, y)                                  # helper
def call(f, args): return FuncCall(f, ExprList(args))
def transpileall(args, names): return [transpile(x, names) for x in args]

def transpile(ast, names):
    "Converts bra-ket Parsing Tree into a C99 AST (skipping the PT → AST step '

    if (ast.data == "term" or ast.data == "operators") and len(ast.children) ==
        return mul(*transpileall(ast.children, names))

    elif ast.data == "factor":                                             # <bra|op
        scalar = mul(*transpileall(ast.children[:2], names))
        if len(ast.children) > 2:
            scalar = mul(scalar, transpile(ast.children[2], names))
        return call(scalar, [Constant("int", "0")] * 2)                    # get sca

    elif ast.data == "bra" or ast.data == "ket":
        return call(ID(str(ast.data)), transpile(ast.children[0], names))

    elif ast.data == "conjugate":                                          # conjuga
        return call(ID(str(ast.data)), [transpile(ast.children[0], names)])

    elif ast.data == "s1" or ast.data == "s2" or ast.data == "s3":         # Pauli s
        return ID(str(ast.data))

    elif ast.data == "value" and len(ast.children) == 1:                   # one nam
        n = str(ast.children[0])
        names.append(n)
        return [call(ID("C"), [ID("r1_" + n), ID("i1_" + n)]), call(ID("C"), [I
    elif ast.data == "value" and len(ast.children) == 2:                   # two com
        return transpileall(ast.children, names)

    elif ast.data == "real":                                               # real nu
        return call(ID("C"), [Constant("double", str(ast.children[0])), Constan
    elif ast.data == "imag":                                               # pure im
        return call(ID("C"), [Constant("double", "0"), Constant("double", str(a
    elif ast.data == "imag1":                                              # pure im
        return call(ID("C"), [Constant("double", "0"), Constant("double", "1")]
    elif ast.data == "complex":                                            # complex
        return call(ID("C"), [Constant("double", str(ast.children[0])), Constan
    elif ast.data == "complex1":                                           # complex
        return call(ID("C"), [Constant("double", str(ast.children[0])), Constan

    else:
        return transpile(ast.children[0], names)                           # pass-th
```

In [8]:
```
names = []
ast = transpile(parser.parse("<0 1| σ₁* σ₂ |0 1>"), names)

ast.show()
# print(c_generator.visit(ast))
names
```

```
FuncCall:
  BinaryOp: *
    BinaryOp: *
      FuncCall:
        ID: bra
        ExprList:
          FuncCall:
            ID: C
            ExprList:
              Constant: double, 0
              Constant: double, 0
          FuncCall:
            ID: C
            ExprList:
              Constant: double, 1
              Constant: double, 0
      BinaryOp: *
        FuncCall:
          ID: conjugate
          ExprList:
            ID: s1
        ID: s2
    FuncCall:
      ID: ket
      ExprList:
        FuncCall:
          ID: C
          ExprList:
            Constant: double, 0
            Constant: double, 0
        FuncCall:
          ID: C
          ExprList:
            Constant: double, 1
            Constant: double, 0
  ExprList:
    Constant: int, 0
    Constant: int, 0
```

Out[8]:
```
[]
```

In [9]:
```
# Define some helper functions in the output language instead of complicating t

ROOT.gInterpreter.Declare("""
complex<double> C(double real, double imag) {
    return complex<double>(real, imag);
}
ROOT::Math::SMatrix<complex<double>, 1, 2> bra(complex<double> up, complex<doub
    return ROOT::Math::SMatrix<complex<double>, 1, 2>((complex<double>[]){up, d
}
ROOT::Math::SMatrix<complex<double>, 2, 1> ket(complex<double> up, complex<doub
    return ROOT::Math::SMatrix<complex<double>, 2, 1>((complex<double>[]){up, d
}
ROOT::Math::SMatrix<complex<double>, 2, 2> conjugate(ROOT::Math::SMatrix<comple
    auto out = ROOT::Math::Transpose(S);
    out(0, 0) = conj(out(0, 0));
    out(0, 1) = conj(out(0, 1));
    out(1, 0) = conj(out(1, 0));
    out(1, 1) = conj(out(1, 1));
    return out;
}
ROOT::Math::SMatrix<complex<double>, 2, 2> matrix(complex<double> a, complex<do
    return ROOT::Math::SMatrix<complex<double>, 2, 2>((complex<double>[]){a, b,
}
auto s1 = matrix(C(0, 0), C(1,  0), C(1, 0), C( 0, 0));
auto s2 = matrix(C(0, 0), C(0, -1), C(0, 1), C( 0, 0));
auto s3 = matrix(C(1, 0), C(0,  0), C(0, 0), C(-1, 0));
""")
```

Out[9]: True

In [10]:
```
# Much of the work is actually rearranging inputs and outputs (matching Python
def c_args(names):
    return ", ".join("double r1_{0}, double i1_{0}, double r2_{0}, double i2_{0

def python_args(names, root_function):
    def prepare(kwargs):
        for n in names:
            x, y = complex(kwargs[n][0]), complex(kwargs[n][1])
            yield x.real
            yield x.imag
            yield y.real
            yield y.imag
    return lambda **kwargs: root_function(*prepare(kwargs))

def python_ret(root_complex):
    return complex(root_complex.real(), root_complex.imag())

print(c_args(["x", "y"]))
# python_args(["x", "y"], print)(x=[0, 1j], y=[1, 0])
```

```
double r1_x, double i1_x, double r2_x, double i2_x, double r1_y, double i1_y, d
```

In [11]:
```
# Again, don't shy away from using strings of the output language when COMPOSIN
def braket(code):
    names = []
    c_ast = transpile(parser.parse(code), names)

    braket.function_num += 1           # new function name each time a new func
    function_name = "braket_{0}".format(braket.function_num)

    ROOT.gInterpreter.Declare("""      // this amount of composing C++ is not h
complex<double> {function_name}({args}) {{
    return {c_code};
}}""".format(
        function_name = function_name,
        args = c_args(names),
        c_code = c_generator.visit(c_ast)))
    root_function = getattr(ROOT, function_name)

    return lambda **kwargs: python_ret(python_args(names, root_function)(**kwar

braket.function_num = 0                # the function knows the number of times
```

In [12]:
```
compiled1 = braket("<0 1| σ₁* σ₂ |0 i>")
print(compiled1())

compiled2 = braket("<x| σ₁* σ₂ |y>")
print(compiled2(x=(0, 1), y=(0, 1j)))
```

```
(1+0j)
(1+0j)
```

Exercise (on your own):

The ROOT function for transposing a matrix is `ROOT::Math::Transpose`. Add a transpose operator to the grammar:

```
operator: spinor
        | operator "*" -> conjugate
        | operator "ᵀ" -> transpose | operator "T" -> transpose
```

and propagate it through the transpiler to the final output so that `<0 1| σ₁ᵀ σ₂ |0 1>` returns `-1j`.

Summary

Some of the work that an **interpreter** performs is unnecessarily repeated. Performing that part once to create a lean executable is **compilation**.

All **compilation** simply translates code from one language to another, though machine instructions are rather hard to read. Compilation to a human-readable language is called source-to-source compilation or **transpilation**.

When transpiling, it's recommended to convert the input language's AST into the output language's AST, rather than directly emitting strings in the output language because strings of code are hard to compose. It's not a hard-edged rule to always avoid strings, but don't rely on *composing* strings.

2019-05-06-adl-language-tools (/github/jpivarski/2019-05-06-adl-language-tools/tree/master)
/
04-type-checking.ipynb (/github/jpivarski/2019-05-06-adl-language-tools/tree/master/04-type-checking.ipynb)

Type checking: proving program correctness

Most compiled languages perform an additional tree-to-tree transformation: **type checking**.

Generally, an **untyped AST** (such as the ones we've been dealing with) gets replaced by a **typed AST**, in which each node is marked by a data type, such as `double` or `boolean`. (It's also possible to mark an AST in-place with type labels, but if so, be sure that node instances are unique!)

Type checking was traditionally motivated by the need to generate the right instructions in the output language (e.g. `__add_int32__` vs `__add_float32__` on unlabled 32-bit registers), but it can be much more general than that:

> **type checking is a formal proof that the program satisfies certain properties.**

The properties to prove are encoded in the **type system**, which can be specialized to a domain like particle physics.

What properties do we want particle physics analysis scripts to satisfy?

Some terminology:

- A **type** is a *set of possible values* that a symbol or expression can have at runtime. Types may be
  - **abstract** if they're specified without reference to a bit-representation, like "all non-negative integers less than `2**32`"
  - **concrete** if a bit-representation is given, like "two's complement 32-bit integers without a sign bit."
- A **strongly typed** language stops processing if it encounters values that do not match function argument types: it either stops the compilation or the runtime execution.
- A **weakly typed** language either passes bits without checking them or converts values to fit expectations.
- A **statically typed** language undergoes a type-checking pass before programs are run, usually as part of a compilation.
- A **dynamically typed** language checks types at runtime. Types may be valid at one time and invalid at another.

Weakly typed (values are *assumed* to fit operations)

- Most assembly languages treat all values as raw bits; programmer has to keep track of types and call the right instructions.
- C is often used as a weakly typed language (e.g. passing everything as `void*`).

Weakly typed (values are *converted* to fit operations)

- Javascript: `"2" + 8 → "28"` (string + number is string concatenation).
- Perl: `"2" + 8 → "10"` (string + number is addition) and unknown or unconvertable variables are presumed to be zero.
- MATLAB: `"2" + 8 → 58` (because the ASCII value of `"2"` is `50`...)
- Python predicates: `None` or `[]` resolves to `False`, `[0]` resolves to `True` when used in `if/and/or/not`.
- Python 2's handling of byte-strings vs unicode, but not Python 3's.
- Most languages promote integers to floating-point values in mixed arithmetic—and that's a good thing!

Strongly but dynamically typed

- Everything else in Python (`"2" + 8` is a `TypeError`).
- Lisp, Ruby, R, Erlang, Lua, Tcl, Smalltalk, PostScript...

Strongly and statically typed

- C++, Java, C#, Rust, Go, Swift, Fortran, Haskell, ML, Scala, Julia, mypy (Python linter), LLVM's assembly language...

In [1]:
```
import lark      # arithmetic, comparisons, and logic: types will b
grammar = """
start: or
or:       and -> pass | and "or" and
and:      not -> pass | not "and" not
not:  compare -> pass | "not" not
compare: term -> pass | term "==" term -> eq | term "!=" term -> ne
                      | term  "<" term -> lt | term "<=" term -> le
                      | term  ">" term -> gt | term ">=" term -> ge
term:  factor -> pass | factor "+" term -> add | factor "-" term ->
factor:  atom -> pass | atom "*" factor -> mul | atom "/" factor ->
atom:      "(" or ")" | CNAME -> symbol | INT -> int | FLOAT -> flo

%import common.CNAME
%import common.INT
%import common.FLOAT
%import common.WS
%ignore WS
"""
parser = lark.Lark(grammar)
```

In [2]:
```
print(parser.parse("not x > 0.0 and 2 + 2").pretty())
```

```
start
  pass
    and
      not
        pass
          gt
            pass
              pass
                symbol  x
            pass
              pass
                float   0.0
      pass
        pass
          add
            pass
              int       2
            pass
              pass
                int     2
```

In [3]:
```
# Define AST nodes, as before. This is the untyped AST.

class AST:
    _fields = ()
    def __init__(self, *args):
        for n, x in zip(self._fields, args):
            setattr(self, n, x)

class Literal(AST):                                  # a literal alw
    _fields = ("value", "type")                      # even in the U
    def __str__(self): return "{0}({1})".format(self.type.__name__,

class Symbol(AST):
    _fields = ("symbol",)
    def __str__(self): return self.symbol

class Call(AST):
    _fields = ("function", "arguments")
    def __str__(self):
        return "{0}({1})".format(str(self.function), ", ".join(str(
```

In [4]:
```
# Simplify the Parsing Tree (PT) into an Abstract Syntax Tree (AST)

def toast(ptnode):
    if ptnode.data == "start" or ptnode.data == "pass" or ptnode.da
        return toast(ptnode.children[0])
    elif ptnode.data == "int":
        return Literal(int(ptnode.children[0]), int)
    elif ptnode.data == "float":
        return Literal(float(ptnode.children[0]), float)
    elif ptnode.data == "symbol":
        return Symbol(str(ptnode.children[0]))
    else:
        return Call(str(ptnode.data), [toast(x) for x in ptnode.chi

print(toast(parser.parse("not x > 0.0 and 2 + 2")))
```

```
and(not(gt(x, float(0.0))), add(int(2), int(2)))
```

In [5]:
```
# The typed AST is just like the untyped AST except that each node

class Typed:
    def __init__(self, thetype, *args):
        self.type = thetype
        super(Typed, self).__init__(*args)
    def __str__(self):
        return "{0} as {1}".format(super(Typed, self).__str__(), se

class TypedLiteral(Typed, Literal): pass

class TypedSymbol(Typed, Symbol): pass

class TypedCall(Typed, Call): pass
```

In [6]:
```
def totyped(ast, symbols):
    if isinstance(ast, Literal):
        return TypedLiteral(ast.type, ast.value)
    elif isinstance(ast, Symbol):
        return TypedSymbol(symbols[ast.symbol], ast.symbol)
    else:
        arguments = [totyped(x, symbols) for x in ast.arguments]
        if ast.function in ("add", "sub", "mul", "truediv"):
            if any(x.type != int and x.type != float for x in argum
                raise TypeError("{0} requires numerical arguments".
            return TypedCall(float, ast.function, arguments)
        elif ast.function in ("eq", "ne", "lt", "le", "gt", "ge"):
            if any(x.type != int and x.type != float for x in argum
                raise TypeError("{0} requires numerical arguments".
            return TypedCall(bool, ast.function, arguments)
        elif ast.function in ("and", "or", "not"):
            if any(x.type != bool for x in arguments):
                raise TypeError("{0} requires boolean arguments".fo
            return TypedCall(bool, ast.function, arguments)
```

In [7]:
```
# Our syntactically correct example has a type error.

code = "not x > 0.0 and 2 + 2"
print(toast(parser.parse(code)))
print(totyped(toast(parser.parse(code)), symbols={"x": float}))
```

```
and(not(gt(x, float(0.0))), add(int(2), int(2)))

---------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
<ipython-input-7-27844ff319a2> in <module>
      3 code = "not x > 0.0 and 2 + 2"
      4 print(toast(parser.parse(code)))
----> 5 print(totyped(toast(parser.parse(code)), symbols={"x": floa

<ipython-input-6-168e13941712> in totyped(ast, symbols)
     16         elif ast.function in ("and", "or", "not"):
     17             if any(x.type != bool for x in arguments):
---> 18                 raise TypeError("{0} requires boolean argume
     19             return TypedCall(bool, ast.function, arguments)

TypeError: 'and' requires boolean arguments
```

In [8]:
```
# The totyped handling can be simplified by looking up a signature

def totyped(ast, signatures, symbols):
    if isinstance(ast, Literal):
        return TypedLiteral(ast.type, ast.value)

    elif isinstance(ast, Symbol):
        return TypedSymbol(symbols[ast.symbol], ast.symbol)

    else:
        arguments = [totyped(x, signatures, symbols) for x in ast.a
        types = [x.type for x in arguments]

        # search for a (name, args) match; apply the corresponding
        for name, args, ret in signatures:
            if name == ast.function and args == types:
                return TypedCall(ret, ast.function, arguments)

        raise TypeError("illegal arguments: {0}({1})".format(
            ast.function, ", ".join(x.__name__ for x in types)))
```

In [9]:
```
# Short exercise: add and test truediv. How does its signature diff

signatures = [("add", [int, int], int),
              ("add", [int, float], float),
              ("add", [float, int], float),
              ("add", [float, float], float),
              ("gt",  [int, int], bool),
              ("gt",  [int, float], bool),
              ("gt",  [float, int], bool),
              ("gt",  [float, float], bool),
              ("not", [bool], bool),
              ("and", [bool, bool], bool),
              ("or",  [bool, bool], bool)]

code = "not x > 0.0 and 3 / 2"
print(toast(parser.parse(code)))
print(totyped(toast(parser.parse(code)), signatures, symbols={"x":
```

```
and(not(gt(x, float(0.0))), truediv(int(3), int(2)))

---------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
<ipython-input-9-4434fdbc0d78> in <module>
     15 code = "not x > 0.0 and 3 / 2"
     16 print(toast(parser.parse(code)))
---> 17 print(totyped(toast(parser.parse(code)), signatures, symbols

<ipython-input-8-0b04e1e1e8f5> in totyped(ast, signatures, symbols)
      9
     10     else:
---> 11         arguments = [totyped(x, signatures, symbols) for x i
     12         types = [x.type for x in arguments]
     13

<ipython-input-8-0b04e1e1e8f5> in <listcomp>(.0)
      9
     10     else:
---> 11         arguments = [totyped(x, signatures, symbols) for x i
     12         types = [x.type for x in arguments]
     13

<ipython-input-8-0b04e1e1e8f5> in totyped(ast, signatures, symbols)
     18
     19         raise TypeError("illegal arguments: {0}({1})".format
---> 20             ast.function, ", ".join(x.__name__ for x in type

TypeError: illegal arguments: truediv(int, int)
```

Parameterized types

To support more data structures, we can consider "functions of types." Like functions in a programming language, they allow us to build what we need from simpler primitives.

Examples include:

- C++ templates: think of the `< >` brackets as `( )` around a function's arguments.
- Arrays, structs, and unions in C, which don't have a function-like syntax but compose like functions.
- `tuple<T1, T2, T3>` values are points in `T1` and `T2` and `T3` (**product type**).
- `variant<T1, T2, T3>` values are each in `T1` or `T2` or `T3` (**sum type**).

Type specifications are described in some language: in C++, they're in the same source file but template arguments have a different syntax than runtime functions. Scala uses square brackets for type functions and parentheses for runtime functions. Dynamically typed languages like Lisp and Python manipulate types at runtime using ordinary functions.

In [10]:
```
import lark

# matrix multiplication language, like bra-ket but without syntacti
grammar = """
start: term
term:  factor -> pass | term "+" factor -> add
factor:  atom -> pass | atom factor -> mul
atom:    "(" term ")" | CNAME -> symbol

%import common.CNAME
%import common.WS
%ignore WS
"""

parser = lark.Lark(grammar)

print(toast(parser.parse("x + A y")))
```

```
add(x, mul(A, y))
```

In [11]:
```
class Matrix:
    def __init__(self, rows, cols):
        self.rows, self.cols = rows, cols
        self.__name__ = str(self)
    def __str__(self):
        return "({0}×{1})".format(self.rows, self.cols)

def totyped(ast, symbols):
    if isinstance(ast, Symbol):
        return TypedSymbol(symbols[ast.symbol], ast.symbol)
    else:
        arguments = [totyped(x, symbols) for x in ast.arguments]
        left, right = [x.type for x in arguments]
        if ast.function == "add":
            if left.rows != right.rows or left.cols != right.cols:
                raise TypeError("cannot add {0} to {1}".format(left
            return TypedCall(left, ast.function, arguments)
        elif ast.function == "mul":
            if left.cols != right.rows:
                raise TypeError("cannot mul {0} to {1}".format(left
            return TypedCall(Matrix(left.rows, right.cols), ast.fun
```

In [12]:
```
code = "x + A y"
print(toast(parser.parse(code)))
print(totyped(toast(parser.parse(code)),
              symbols={"x": Matrix(1, 5), "A": Matrix(5, 4), "y": M
```

```
add(x, mul(A, y))

---------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
<ipython-input-12-052d33f5e167> in <module>
      2 print(toast(parser.parse(code)))
      3 print(totyped(toast(parser.parse(code)),
----> 4               symbols={"x": Matrix(1, 5), "A": Matrix(5, 4)

<ipython-input-11-501fa9f77437> in totyped(ast, symbols)
     14         if ast.function == "add":
     15             if left.rows != right.rows or left.cols != right
---> 16                 raise TypeError("cannot add {0} to {1}".form
     17             return TypedCall(left, ast.function, arguments)
     18         elif ast.function == "mul":

TypeError: cannot add (1×5) to (5×1)
```

Traditionally (in FORTRAN and C, for instance), the reason for the type-check is to ensure that the right machine instructions are applied to each value. Recently (particularly in Haskell and Scala), the focus has been to ensure that the programmer is not making other kinds of mistakes.

Examples include:

- Avoiding null-pointer exceptions by wrapping type `T` as `optional<T>`. (Won't compile without unpacking/null-checking.)
- Rust includes ownership rules as part of each type, so if a program compiles, it can't have a memory leak or double-free.
- Parameterized function types: `f: (int, double) → string` as `Function[Int, Double, String]`.
- Parameter bounds: `List[T <: Particle]` and `List[T <: Particle with Charged]`.

Refinement types

One of these new ideas is to specify the set of values that a type can represent with a predicate. For instance, instead of just `int` and `unsigned int`, how about `{x:ℤ | x > 0}` (positive integers) or `{x:ℤ | x % 2 == 0}` (even integers)?

These are **refinement types**, implemented as libraries in Haskell (https://ucsd-progsys.github.io/liquidhaskell-tutorial/) and Scala (https://github.com/fthomas/refined). Example use: ensure that a programmer can't ask for the first element of an empty list. Many runtime errors become compile-time errors.

Unlike other non-mainstream type ideas, this could be particularly relevant for us because we deal in measurable quantities with intervals of validity. It could help to know at compile-time whether $\phi \in (-\pi, \pi]$ or $[0, 2\pi)$, or to forbid 0/0 and ∞/∞ at compile-time, rather than silently returning `nan`.

In this next example, we'll implement a compile-time 0/0 and ∞/∞ check.

In [13]:
```
import lark

grammar = """
start:  term
term:   factor -> pass | term "+" factor -> add | term "-" factor -
factor: atom -> pass | atom "*" factor -> mul | atom "/" factor ->
atom:   "(" term ")" | CNAME -> symbol

%import common.CNAME
%import common.WS
%ignore WS
"""

parser = lark.Lark(grammar)

print(toast(parser.parse("(x + y) / z")))
```

```
truediv(add(x, y), z)
```

In [14]:
```
class Interval:
    def __init__(self, low=0, high=float("inf"), includeslow=True,
        if low < 0 or high <= low:                      # simplifyi
            raise TypeError("Intervals must be positive (low >= 0)
        self.low, self.high = low, high
        self.includeslow, self.includeshigh = includeslow, includes
        self.__name__ = str(self)

    def __str__(self):
        return "{0}{1}, {2}{3}".format("[" if self.includeslow else
                                       self.high, "]" if self.inclu

    def __contains__(self, value):
        return (self.low <= value if self.includeslow else self.low
               (value <= self.high if self.includeshigh else value

print(Interval(0, float("inf"), includeshigh=False))
```

```
[0, inf)
```

In [15]:
```
def add(a, b):                  # addition in subintervals of the ex
    return Interval(a.low + b.low, a.high + b.high,
                    a.includeslow and b.includeslow, a.includeshigh
def truediv(a, b):              # division in subintervals of the ex
    def extended_div(x, y):     # division of extended reals
        try:
            return x / y
        except ZeroDivisionError:
            return float("inf")
    if 0 in a and 0 in b:
        raise TypeError("0/0 could happen at runtime")
    elif float("inf") in a and float("inf") in b:
        raise TypeError("∞/∞ could happen at runtime")
    else:
        return Interval(extended_div(a.low, b.high), extended_div(a
                        a.includeslow and b.includeshigh, a.include

Interval.__add__, Interval.__truediv__ = add, truediv
print(Interval(3, 8) + Interval(10, 20))
```

```
[13, 28]
```

In [16]:
```
def totyped(ast, symbols):
    if isinstance(ast, Symbol):
        return TypedSymbol(symbols[ast.symbol], ast.symbol)
    else:
        arguments = [totyped(x, symbols) for x in ast.arguments]
        left, right = [x.type for x in arguments]
        if ast.function == "add":
            return TypedCall(left + right, ast.function, arguments)
        elif ast.function == "truediv":
            return TypedCall(left / right, ast.function, arguments)

code = "(x + y) / z"
print(toast(parser.parse(code)))
print(totyped(toast(parser.parse(code)),
              symbols={"x": Interval(3, 8), "y": Interval(10, 20),
```

```
truediv(add(x, y), z)
truediv(add(x as [3, 8], y as [10, 20]) as [13, 28], z as [0, 100])
```

Long exercise (on your own):

- Add literal numbers, a few comparisons (e.g. `<` and `>`), and `if/then/else` as in the parsing tutorial.
- Add a boolean type in addition to `Interval`. (Should it restrict to only one variable? Is `Boolean(only=True)` useful for anything?)
- Let the predicate of `if/then/else` refine the type that is passed into the `then/else` clauses. For instance, if the type of `x` is `[3, 8]` and it passes

through an `if x > 5` predicate, what should its type be in the `then/else` clauses? Use nested scopes.

Even longer exercise:

- Add function definitions and propagate types through functions.
- If the function has no type annotations, its types can be defined by how it's called. In that case, though, it can be typed differently when called with different arguments.

I started thinking along these lines in a (defunct) project called Femtocode (https://github.com/diana-hep/femtocode).

Trailing off without conclusions...

Type theory is a very active branch of CS. (Unlike parsing.) New typing features often imply constraints on what the language can express, but that's what a Domain-Specific Language (DSL) is supposed to do. In fact, the *right* constraint could add a lot of value to the DSL.

A reading list:

- Curry-Howard correspondence (https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence): programs are equivalent to proofs, and type-checkers are automated theorem provers. (Really!)
- Total functional programming (https://en.wikipedia.org/wiki/Total_functional_programming): if we don't need a Turing complete (https://en.wikipedia.org/wiki/Turing_completeness) language, total functional languages are guaranteed to never raise exceptions at runtime: *if it compiles, it will run*. Recursion is hard in these languages (leading to a long discussion of primitive recursion and data vs codata), but we don't care about that: non-recursive math would be enough for a lot of particle physics applications.
- Structural type system (https://en.wikipedia.org/wiki/Structural_type_system): instead of checking type equivalence by name or inheritance, only check to see if it has the components the function needs. This is the duck typing (https://en.wikipedia.org/wiki/Duck_typing) of type-safe languages.

2019-05-06-adl-language-tools (/github/jpivarski/2019-05-06-adl-language-tools/tree/master)
/
05-embedded-dsl.ipynb (/github/jpivarski/2019-05-06-adl-language-tools/tree/master/05-embedded-dsl.ipynb)

Embedded DSLs: using an existing language's syntax

Do we need a new language?

Or more specifically, do we need a new syntax?

An **embedded DSL** is a domain-specific language that is implemented in another language. The difference between an embedded DSL and an ordinary library is subjective.

For example:

- Numpy (https://docs.scipy.org/doc/numpy/reference/) implements array programming idioms in Python, which had been language features in MATLAB, R, and APL.
- My awkward-array (https://github.com/scikit-hep/awkward-array) is an embedded DSL for physics: `(muons[:, 0] + muons[:, 1]).mass` → Z peak.
- Akka (https://akka.io/) implements the actor model of concurrency, which had been a language feature in Erlang.
- Boost.Proto (https://www.boost.org/doc/libs/1_58_0/doc/html/proto.html) provides tools to build DSLs in C++.
- Scala's syntax rules (https://scalac.io/encog-dsl-scala-part1) are flexible enough for some radical DSLs (next pages).

Libraries are more on the "embedded DSL" end of the spectrum if they consist of shallow functions that only do interesting things when combined, like the constructs of a programming language.

In Scala, a class's methods can be accessed through a dot `.` (like most languages) or a space `. Methods taking a single argument need not have parentheses (`), and they can be named with unicode characters.

The following are equivalent:

```
a.cross(b)      // a and b are ThreeVectors, which has a cros
a cross b       // the dot and parentheses may be omitted
a × b           // cross can also be named ×
```

Scala also lets functions decide whether they want to be eagerly or lazily evaluated, and has Lisp-like macros. This leads to some extreme DSLs, like ScalaTest (http://www.scalatest.org/):

```
x should not equal 1

List("one", "two", "three") should contain ("two")

an [IndexOutOfBoundsException] should be thrownBy string.char
```

The Chisel (https://chisel.eecs.berkeley.edu/) library specifies circuits for FPGAs several times more succinctly than Verilog. This is a vending machine:

```
class VendingMachine extends Component {
  val io = new Bundle {
    val nickel = Bool(INPUT)
    val dime   = Bool(INPUT)
    val ready  = Bool(OUTPUT)
  }
  val s_idle :: s_5 :: s_10 :: s_15 :: s_ok :: Nil = Enum(5)(
  val state = Reg(resetVal = s_idle)
  switch (state) {
    is (s_idle) { when (io.nickel) { state := s_5 }  when (ic
    is (s_5)    { when (io.nickel) { state := s_10 } when (ic
    is (s_10)   { when (io.nickel) { state := s_15 } when (ic
    is (s_15)   { when (io.nickel) { state := s_ok } when (ic
    is (s_ok)   { state := s_idle }
  }
  io.ready := (state === s_ok)
}
```

But most physicists use Python for high-level analysis.

Python's syntax is not nearly as flexible as Scala's, nor does it give control over eager/lazy evaluation like Scala and C# (unless you ask the user to always pass functions as arguments!), so options for embedded DSLs in Python are limited.

In all the demos I've shown so far, we've defined new grammars.

- This has the *advantage* of complete freedom in syntax and interpretation.
- This has the *disadvantage* that the language must always live in quoted strings or separate files.

It's okay for small snippets of the new language, like regular expressions or SQL queries, but for longer programs, the user will be limited by their favorite editor not interpreting the language for syntax highlighting, auto-indentation, tab-completion, integrated documentation...

In [1]:
```
# Best of both? Code in Python FUNCTIONS are syntax-checked but oth

def function(x, y):
    return x**2 + y**2

# The source code has been compiled for Python's virtual machine, w
# but that language can be parsed (not by humans).

print(function.__code__.co_code)
```

```
b'|\x00d\x01\x13\x00|\x01d\x01\x13\x00\x17\x00S\x00'
```

In [2]:

```
import sys, uncompyle6.parser, uncompyle6.scanner

def parse(function):
    code = function.__code__
    scanner = uncompyle6.scanner.get_scanner(float(sys.version[0:3]
    parser = uncompyle6.parser.get_python_parser(float(sys.version[
    tokens, customize = scanner.ingest(code)
    return uncompyle6.parser.parse(parser, tokens, customize)

parse(lambda x, y: x**2 + y**2)
```

Out[2]:

```
stmts
    sstmt
        stmt
            return (2)
                 0. ret_expr
                    expr
                        binary_expr (3)
                             0. expr
                                binary_expr (3)
                                     0. expr
                                        L.  10       0  LOAD_FAST
                                     1. expr
                                                     2  LOAD_CONST
                                     2. binary_op
                                                     4  BINARY_POWEI
                             1. expr
                                binary_expr (3)
                                     0. expr
                                                     6  LOAD_FAST
                                     1. expr
                                                     8  LOAD_CONST
                                     2. binary_op
                                                    10  BINARY_POWEI
                             2. binary_op
                                                    12  BINARY_ADD
                 1.             14  RETURN_VALUE
```

In [3]:
```
# Make our own AST, not that parsing tree or Python's own AST. We w

class AST:
    _fields = ()
    def __init__(self, *args):
        for n, x in zip(self._fields, args):
            setattr(self, n, x)

class Literal(AST):
    _fields = ("value",)
    def __str__(self): return str(self.value)

class Symbol(AST):
    _fields = ("symbol",)
    def __str__(self): return self.symbol

class Call(AST):
    _fields = ("function", "arguments")
    def __str__(self):
        return "{0}({1})".format(str(self.function), ", ".join(str(
```

In [4]:
```
# Now we just need a toaster for uncompyle6's parsing tree.

def toast(ptnode):
    if ptnode.kind == "binary_expr":
        return Call(toast(ptnode[2]), [toast(x) for x in ptnode[:2]
    elif ptnode.kind == "BINARY_ADD":
        return Symbol("add")
    elif ptnode.kind == "BINARY_POWER":
        return Symbol("pow")
    elif ptnode.kind == "LOAD_FAST":
        return Symbol(ptnode.pattr)
    elif ptnode.kind == "LOAD_CONST":
        return Literal(ptnode.pattr)
    else:
        # a lot of nodes are purely structural
        return toast(ptnode[0])

# It's incomplete, but it covers enough for our example.
print(toast(parse(lambda x, y: x**2 + y**2)))
```

```
add(pow(x, 2), pow(y, 2))
```

Python's grammar (https://docs.python.org/3/reference/grammar.html) is not very large, not hard to fully translate—at least the expressions (86 rules, 36 are for expressions):

```
test:           or_test ['if' or_test 'else' test] | lambdef
test_nocond:    or_test | lambdef_nocond
lambdef:        'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test:        and_test ('or' and_test)*
and_test:       not_test ('and' not_test)*
not_test:       'not' not_test | comparison
comparison:     expr (comp_op expr)*
comp_op:        '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not'
'in'|'is'|'is' 'not'
star_expr:      '*' expr
expr:           xor_expr ('|' xor_expr)*
xor_expr:       and_expr ('^' and_expr)*
and_expr:       shift_expr ('&' shift_expr)*
shift_expr:     arith_expr (('<<'|'>>') arith_expr)*
arith_expr:     term (('+'|'-') term)*
term:           factor (('*'|'@'|'/'|'%'|'//') factor)*
factor:         ('+'|'-'|'~') factor | power
power:          atom_expr ['**' factor]
atom_expr:      ['await'] atom trailer*
atom:           ('(' [yield_expr|testlist_comp] ')' | '[' [t
estlist_comp] ']' |
                '{' [dictorsetmaker] '}' | NAME | NUMBER | S
TRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp:  (test|star_expr) ( comp_for | (',' (test|sta
r_expr))* [','] )
trailer:        '(' [arglist] ')' | '[' subscriptlist ']' |
'.' NAME
subscriptlist:  subscript (',' subscript)* [',']
subscript:      test | [test] ':' [test] [sliceop]
sliceop:        ':' [test]
exprlist:       (expr|star_expr) (',' (expr|star_expr))*
[',']
testlist:       test (',' test)* [',']
dictorsetmaker: (((test ':' test | '**' expr) (comp_for |
(',' (test ':' test | '**' expr))* [',' ])) |
                ((test | star_expr) (comp_for | (',' (test |
star_expr))* [',' ])))
arglist:        argument (',' argument)*  [',']
argument:       ( test [comp_for] | test '=' test | '**' tes
t | '*' test )
comp_iter:      comp_for | comp_if
sync_comp_for:  'for' exprlist 'in' or_test [comp_iter]
comp_for:       ['async'] sync_comp_for
comp_if:        'if' test_nocond [comp_iter]
yield_expr:     'yield' [yield_arg]
yield_arg:      'from' test | testlist
```

In [5]:
```
# And the whole apparatus can be wrapped up in a Python decorator.
# (Numba uses the same mechanism to specify that a function is to b

def quote(function):                    # like Lisp's "quote"
    return toast(parse(function))

@quote
def sum_quadrature(x, y):
    return x**2 + y**2

print(sum_quadrature)
```

```
add(pow(x, 2), pow(y, 2))
```

With the ability to turn any Python function into an AST, we can add

- lazy evaluation or a declarative interpretation;
- type-checking, possibly with physics-motivated features, like refinement types;
- compilation through C++ or directly into bytecode (though Numba does that);
- AST transformations, such as derivatives (Calculus);
- AST rewriting, such as algebraic simplifications;
- ...

New syntax vs Python syntax is a *separate question* from the above features. It's also possible to have two syntaxes for the same language (different parsing steps produce the same ASTs).

Using Python as a syntax limits us to Python syntax rules and Pythonic expectations (users would be upset if we changed the *meaning* of the operators), but it provides a better editing experience.

I started thinking along these lines in a project called rejig (https://github.com/diana-hep/rejig).