



How to write a programming language

Andy Balaam, OpenMarket

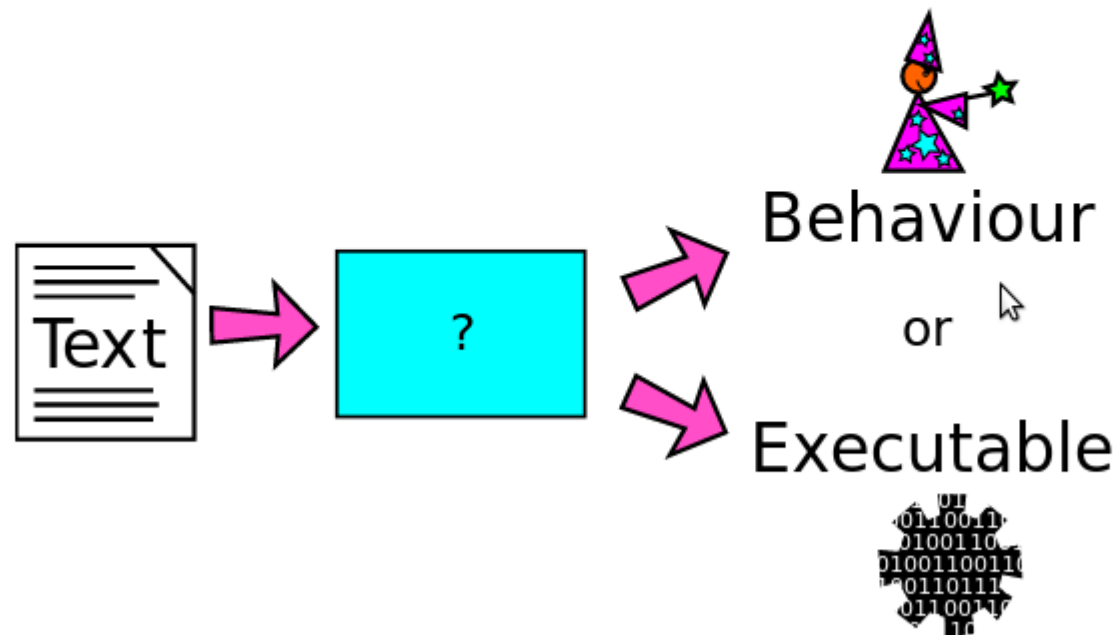
artificialworlds.net/blog



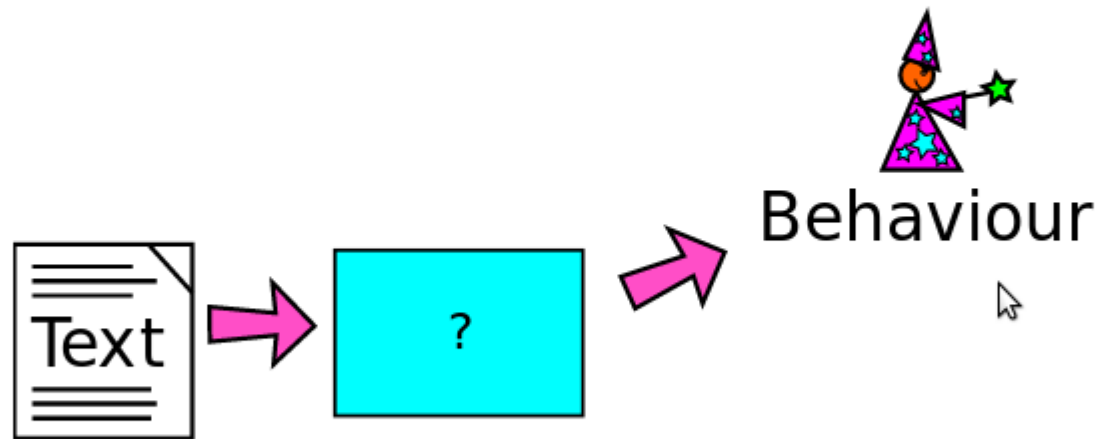
Contents

- What is a programming language?
- Introducing Cell
- Lexing
- Parsing
- Evaluation
- Discussion

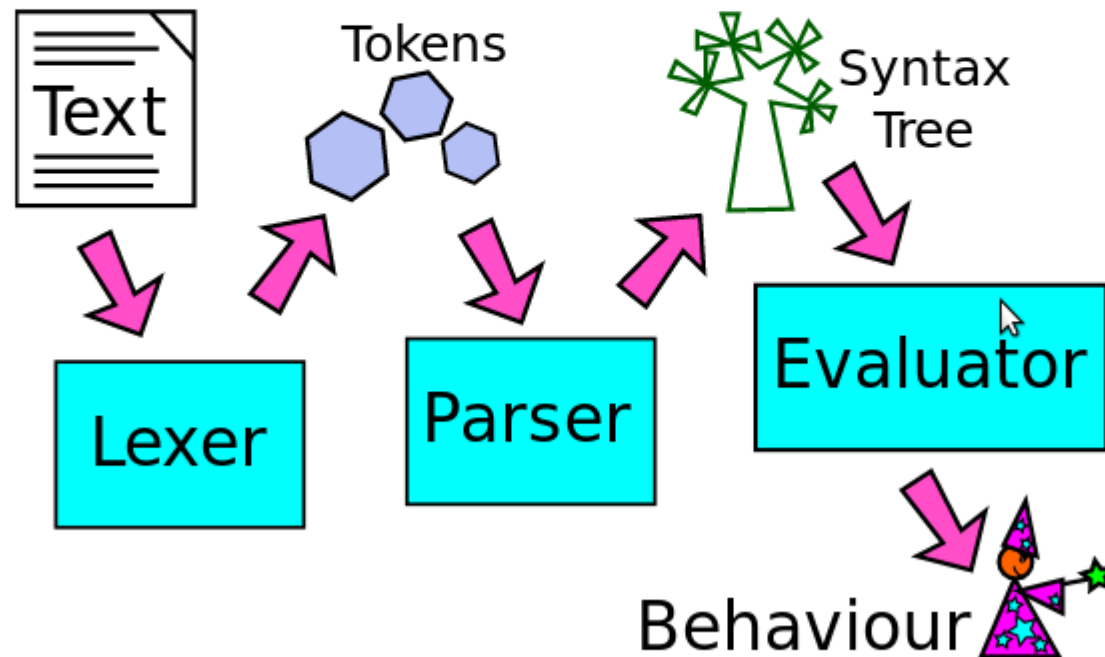
What is a programming language?



What is a programming language?



What is a programming language?



Introducing Cell

Cell is a programming language with:

- Short implementation
- (Hopefully) understandable implementation

There is nothing else good about it.

github.com/andybalaam/cell

Introducing Cell

```
num1 = 3;  
  
square = {:(x) x * x};  
  
num2 = square( num1 );
```



Introducing Cell

Cell has four expression types:

- Assignment `x = 3`
- Operations `4 + y`
- Function calls `sqrt(-1)`
- Function definitions `{:(x, y) x*x + y*y;}`

Introducing Cell

The coolest thing about Cell is:

- `if` is a function, not a keyword

More at: github.com/andybalaam/cell

Introducing Cell

- `if` is a function, not a keyword

```
if(  
  is_even( 2 ),  
  { print "Even!"; },  
  { print "Odd."; }  
);
```

More at: github.com/andybalaam/cell

Introducing Cell - Scope

```
x = "World!";  
myfn = {  
  x = "Hello, ";  
  print( x );  
};  
myfn();  
print( x );
```

Introducing Cell - Scope

```
x = "World!";  
myfn = {  
  x = "Hello, ";  
  print( x );  
};  
myfn();  
print( x );
```

```
Hello,  
World!
```

Introducing Cell - Scope

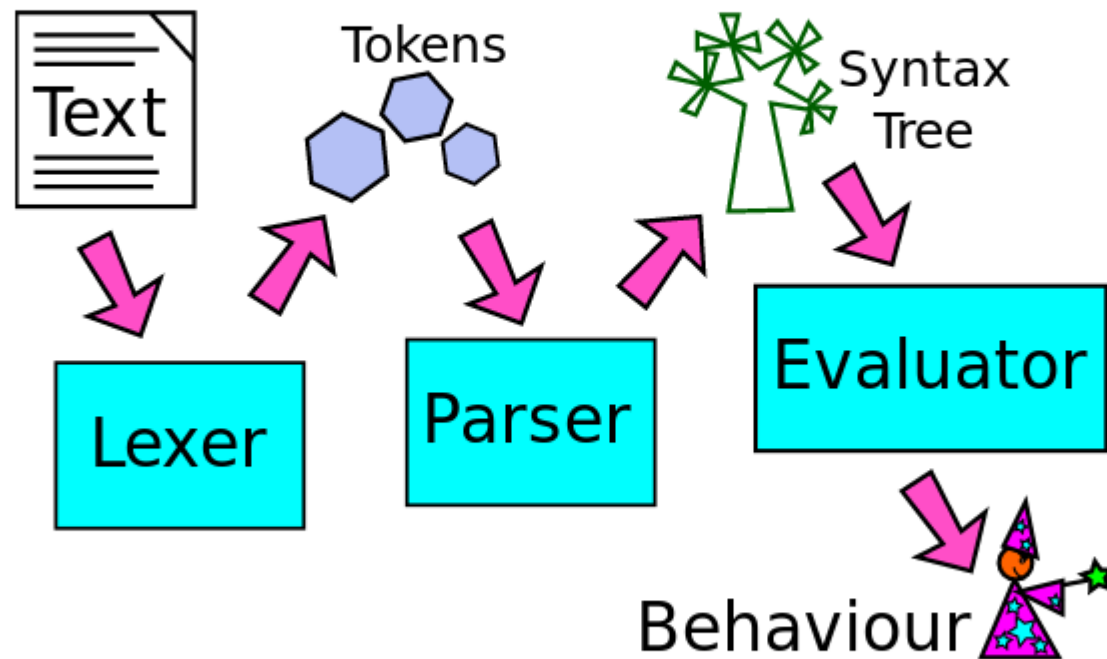
```
outerfn = {  
  x = 12;  
  innerfn = {  
    print(x);  
  };  
  innerfn;  
};  
  
thing = outerfn();  
thing();
```

Introducing Cell - Scope

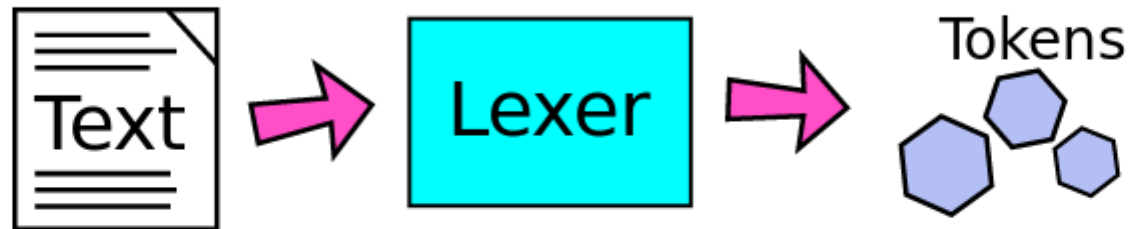
```
outerfn = {  
  x = 12;  
  innerfn = {  
    print(x);  
  };  
  innerfn;  
};  
  
thing = outerfn();  
thing();
```

12

A programming language



Lexing



Lexers emit tokens

```
foo = "bar";
```

becomes:

```
("symbol", "foo")  
("=", "" )  
("string", "bar")  
(";", "" )
```

Lexers emit tokens

```
200 - 158
```

becomes:

```
("number", "200")  
("operator", "-")  
("number", "158")
```

Cell's lexer

Lexing in Cell consists of identifying:

- Numbers: 12, 4.2
- Strings: "foo", 'bar'
- Symbols: baz, qux_Quux
- Operators: +, -
- Other punctuation: (, }

Cell's lexer (40 lines)

```
import re

from pycell.peekablestream import PeekableStream

def _scan(first_char, chars, allowed):
    ret = first_char
    p = chars.next
    while p is not None and re.match(allowed, p):
        ret += chars.move_next()
        p = chars.next
    return ret

def _scan_string(delim, chars):
    ret = ""
    while chars.next != delim:
        c = chars.move_next()
        if c is None:
            raise Exception("A string ran off the end of the program.")
        ret += c
    chars.move_next()
    return ret

def lex(chars_iter):
    chars = PeekableStream(chars_iter)
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n":
            pass # Ignore white space
        elif c in "(){}.,;=:":
            yield (c, "") # Special characters
        elif c in "+-*/":
            yield ("operation", c)
        elif c in ("'", '"'):
            yield ("string", _scan_string(c, chars))
        elif re.match("[.0-9]", c):
            yield ("number", _scan(c, chars, "[.0-9]"))
        elif re.match("[_a-zA-Z]", c):
            yield ("symbol", _scan(c, chars, "[_a-zA-Z0-9]"))
        elif c == "\t":
            raise Exception("Tab characters are not allowed in Cell.")
        else:
            raise Exception("Unrecognised character: '" + c + "'.")
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": ...
        elif c in "+-*/": ...
        elif c in "(){};=:": ...
        elif c in ("'", '"'): ...
        elif re.match("[.0-9]", c): ...
        elif re.match("[_a-zA-Z]", c): ...
        else: ...
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        elif c in "+-*/*": ...
        elif c in "(){};=:": ...
        elif c in ("'", '"'): ...
        elif re.match("[.0-9]", c): ...
        elif re.match("[_a-zA-Z]", c): ...
        else: ...
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        elif c in "+-*/": yield ("operation", c)
        elif c in "(){};=:": ...
        elif c in ("'", '"'): ...
        elif re.match("[.0-9]", c): ...
        elif re.match("[_a-zA-Z]", c): ...
        else: ...
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        elif c in "+-*/": yield ("operation", c)
        elif c in "(){} ,;=:": yield (c, "")
        elif c in ("'", '"'): ...
        elif re.match("[.0-9]", c): ...
        elif re.match("[_a-zA-Z]", c): ...
        else: ...
```


Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        elif c in "+-*/*": yield ("operation", c)
        elif c in "(){};=:": yield (c, "")
        elif c in ("'", '"'):
            yield ("string", _scan_string(c, chars))
        ...
```

Cell's lexer

```
def _scan_string(delim, chars):  
    ret = ""  
    while chars.next != delim:  
        c = chars.move_next()  
        if c is None:  
            raise Exception(...)   
        ret += c  
    chars.move_next()  
    return ret
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        elif c in "+-*/": yield ("operation", c)
        elif c in "(){};=:": yield (c, "")
        elif c in ("'", '"'): yield ("string...", c)
        elif re.match("[.0-9]", c):
            yield ("number", _scan(c, chars, "[.0-9]"))
        ...
```

Cell's lexer

```
def _scan(first_char, chars, allowed):  
    ret = first_char  
    p = chars.next  
    while p is not None and re.match(allowed, p):  
        ret += chars.move_next()  
        p = chars.next  
    return ret
```

Cell's lexer

```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": pass
        ...
        elif re.match("[_a-zA-Z]", c):
            yield (
                "symbol",
                _scan(c, chars, "[_a-zA-Z0-9]")
            )
        ...
```

Cell's lexer

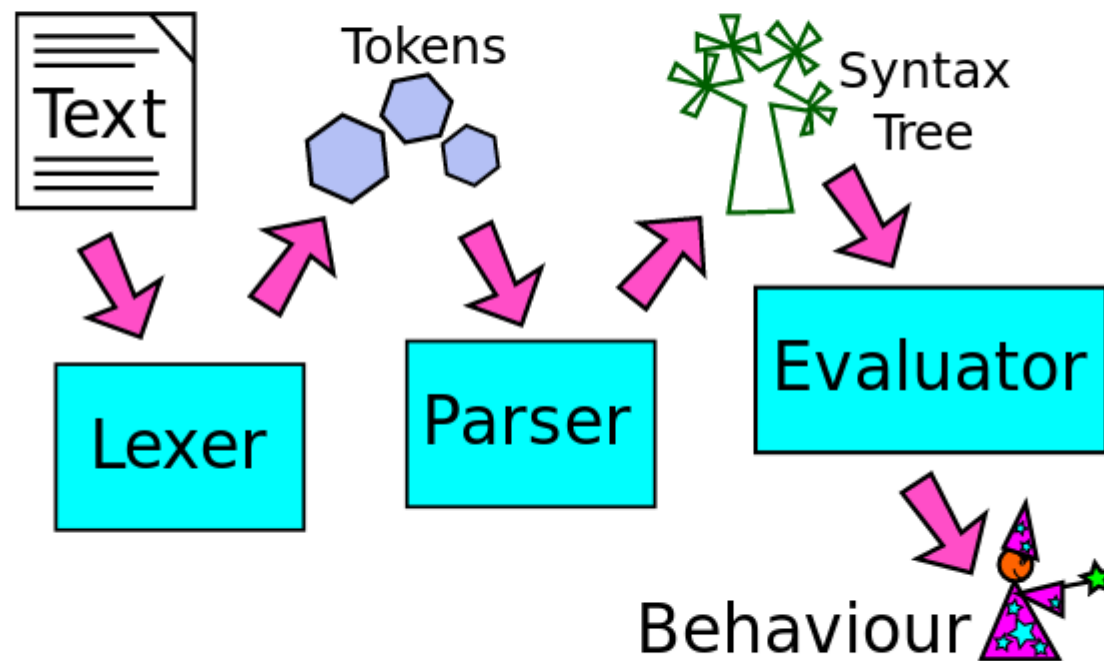
```
def lex(chars):
    while chars.next is not None:
        c = chars.move_next()
        if c in " \n": ...
        elif c in "+-*/": ...
        elif c in "(){} ,;=:": ...
        elif c in ("'", '"'): ...
        elif re.match("[.0-9]", c): ...
        elif re.match("[_a-zA-Z]", c): ...
        else: raise Exception(...)
```

Cell's lexer

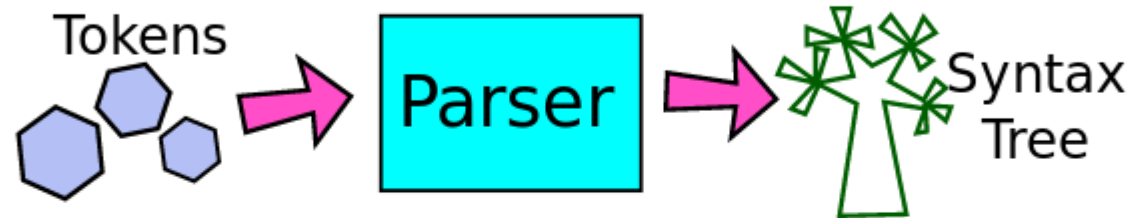


```
assert (  
  list(lex('print("Hello, world!");'))  
  ==  
  [ ("symbol", "print")  
    , ("(", "")  
    , ("string", "Hello, world!")  
    , (")", "")  
    , (";", "")  
    ]  
)
```

A programming language



Parsing



Parsing

```
x = 3 + 4;
```

becomes:

```
Assignment:  
  Symbol: x  
  Value:  
    Operation:  
      Type: +  
      Arguments:  
        3  
        4
```

Parsing

```
print( x + 2 );
```

becomes:

```
FunctionCall:  
  Function: print  
  Arguments:  
    Operation:  
      Type: +  
      Arguments:  
        x  
        2
```

Parsing

```
print( x + 2 );
```

becomes:

```
("call",  
  ("symbol", "print"),  
  [  
    ("operation",  
      "+",  
      ("symbol", "x"),  
      ("number", "2"))  
  ]  
)
```

Cell's parser (81 lines)

Cell's parser

```
def parse(tokens):  
    parser = Parser(tokens, ";")  
    while parser.tokens.next is not None:  
        p = parser.next_expr(None)  
        if p is not None:  
            yield p  
        parser.tokens.move_next()
```

Cell's parser

```
class Parser:  
    def __init__(self, tokens, stop_at):  
        self.tokens = tokens  
        self.stop_at = stop_at
```

Cell's parser

```
def next_expr(self, prev):
    self.fail_if_at_end(";")
    typ, value = self.tokens.next
    if typ in self.stop_at:
        return prev
    self.tokens.move_next()
    # ...
```


Cell's parser

```
def next_expr(self, prev):  
    # ...  
    if prev is None and typ in (# ...  
    elif typ == "operation": # ...  
    elif typ == "(": # ...  
    elif typ == "{": # ...  
    elif typ == "=": # ...  
    else: # ...
```

Cell's parser

```
def next_expr(self, prev):
    # ...
    if prev is None and typ in (
        "number", "string", "symbol"):
        return self.next_expr((typ, value))
    elif typ == "operation": # ...
    elif typ == "(": # ...
    elif typ == "{": # ...
    elif typ == "=": # ...
    else: # ...
```

Cell's parser

```
def next_expr(self, prev):  
    # ...  
    if prev is None and typ in (...  
    elif typ == "operation":  
        nxt = self.next_expr(None)  
        return self.next_expr(  
            ("operation", value, prev, nxt))  
    elif typ == "(": # ...  
    elif typ == "{": # ...  
    elif typ == "=": # ...  
    else: # ...
```

Cell's parser

```
def next_expr(self, prev):
    # ...
    if prev is None and typ in (...
    elif typ == "operation": # ...
    elif typ == "(":
        args = self.multi_exprs(",", " ")
        return self.next_expr(("call", prev, args))
    elif typ == "{": # ...
    elif typ == "=": # ...
    else: # ...
```

Cell's parser

```
def multi_exprs(self, sep, end):  
    ret = []  
    self.fail_if_at_end(end)  
    typ = self.tokens.next[0]  
    if typ == end:  
        self.tokens.move_next()  
    else:  
        # ...  
    return ret
```

Cell's parser

```
def multi_exprs(self, sep, end):
    # ...
    else:
        arg_parser = Parser(self.tokens, (sep, end))
        while typ != end:
            p = arg_parser.next_expr(None)
            if p is not None:
                ret.append(p)
            typ = self.tokens.next[0]
            self.tokens.move_next()
            self.fail_if_at_end(end)
        return ret
```

Cell's parser

```
def next_expr(self, prev):  
    # ...  
    elif typ == "(": # ...  
    elif typ == "{":  
        params = self.parameters_list()  
        body = self.multi_exprs(";", "{")  
        return self.next_expr(("function", params, body))  
    elif typ == "=": # ...  
    else: # ...
```

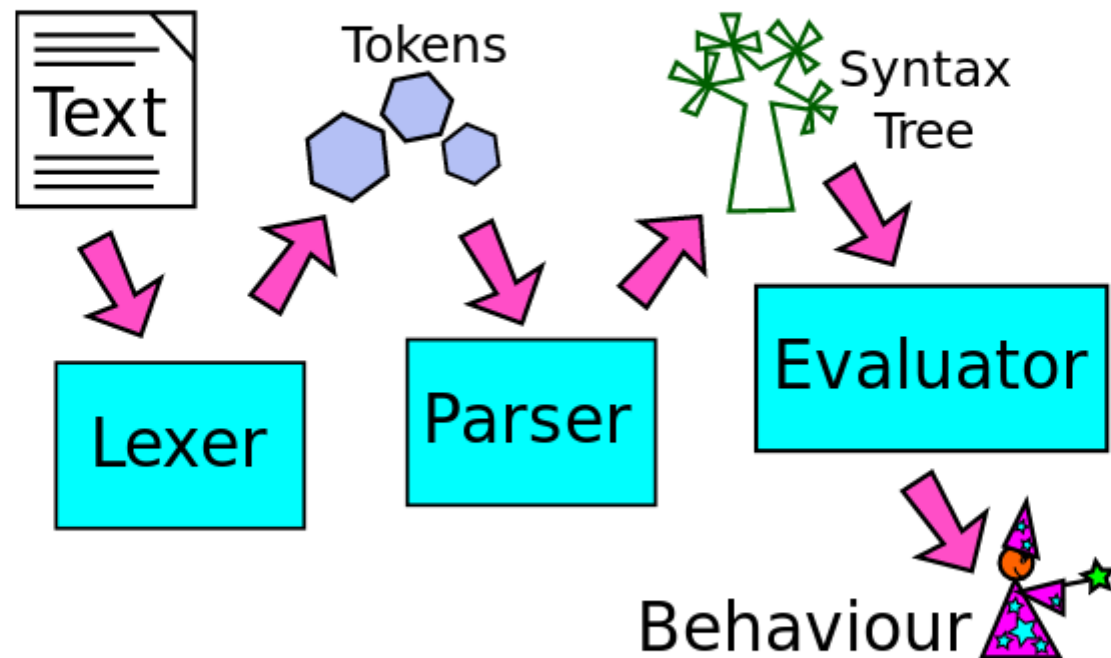
Cell's parser

```
def next_expr(self, prev):  
    # ...  
    elif typ == "(": # ...  
    elif typ == "{": # ...  
    elif typ == "=":  
        if prev[0] != "symbol":  
            raise Exception(...)  
        nxt = self.next_expr(None)  
        return self.next_expr(("assignment", prev, nxt))  
    else: # ...
```


Cell's parser

```
def next_expr(self, prev):  
    # ...  
    elif typ == "(": # ...  
    elif typ == "{": # ...  
    elif typ == "=": # ...  
    else:  
        raise Exception("Unexpected token: " ...
```

A programming language



Recap - Lexing

```
foo = "bar";
```

becomes:

```
("symbol", "foo")  
("=", "" )  
("string", "bar")  
(";", "" )
```

Recap - Parsing

```
print( x + 2 );
```

becomes:

```
("call",  
  ("symbol", "print"),  
  [  
    ("operation",  
      "+",  
      ("symbol", "x"),  
      ("number", "2"))  
  ]  
)
```

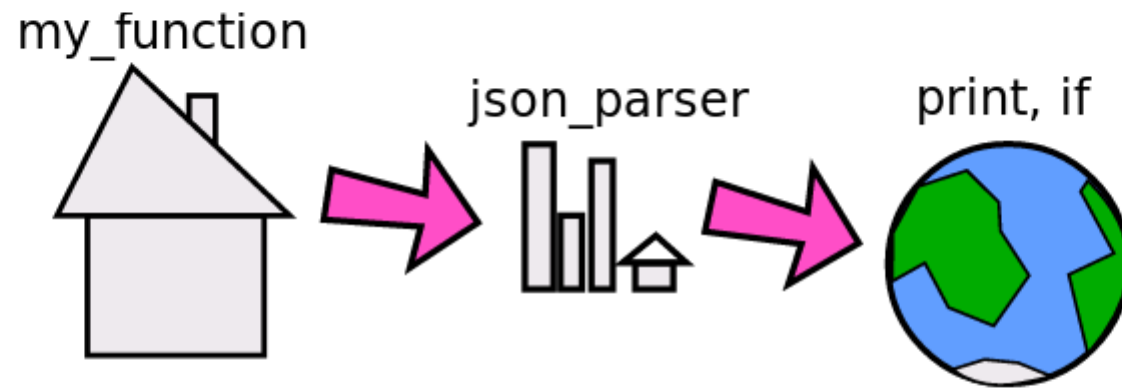
Recap - Cell

```
num1 = 3;  
  
square = {(x) x * x};  
  
num2 = square( num1 );
```

Recap - Cell, scope

```
outerfn = {  
  x = 12;  
  innerfn = {  
    print(x);  
  };  
  innerfn;  
};  
  
thing = outerfn();  
thing();
```

Environments



Environments

```
class Env:
    # ...
    def get(self, name):
        if name in self.items:
            return self.items[name]
        elif self.parent is not None:
            return self.parent.get(name)
        else:
            return None
```


Environments

```
class Env:  
    # ...  
    def set(self, name, value):  
        self.items[name] = value
```

Cell's evaluator (94 lines)

[illegible]

Cell's evaluator

```
def eval_expr(expr, env):  
    typ = expr[0]  
    if typ == "number": ...  
    elif typ == "string": ...  
    elif typ == "none": ...  
    elif typ == "operation": ...  
    elif typ == "symbol": ...  
    elif typ == "assignment": ...  
    elif typ == "call": ...  
    elif typ == "function": ...  
    else: ...
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    if typ == "number":  
        return ("number", float(expr[1]))
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "string":  
        return ("string", expr[1])
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "none":  
        return ("none",)
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "operation":  
        return _operation(expr, env)
```

Cell's evaluator

```
def _operation(expr, env):  
    arg1 = eval_expr(expr[2], env)  
    arg2 = eval_expr(expr[3], env)  
    if expr[1] == "+":  
        return ("number", arg1[1] + arg2[1])  
    elif expr[1] == "-":  
        return ("number", arg1[1] - arg2[1])  
    # ...  
    else:  
        raise Exception("Unknown operation: " + expr[1])
```


Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "symbol":  
        name = expr[1]  
        ret = env.get(name)  
        if ret is None:  
            raise Exception("Unknown symbol '%s'." % name)  
        else:  
            return ret
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "assignment":  
        var_name = expr[1][1]  
        if var_name in env.items:  
            raise Exception("Overwriting '%s'." % var_name)  
        val = eval_expr(expr[2], env)  
        env.set(var_name, val)  
        return val
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "call":  
        return _function_call(expr, env)
```

Cell's evaluator

```
def _function_call(expr, env):  
    fn = eval_expr(expr[1], env)  
    args = list((eval_expr(a, env) for a in expr[2]))  
    if fn[0] == "function": ...  
    elif fn[0] == "native": ...  
    else: ...
```

Cell's evaluator

```
args = list((eval_expr(a, env) for a in expr[2]))
...
if fn[0] == "function":
    params = fn[1]
    fail_if_wrong_number_of_args(expr[1], params, args)
    body = fn[2]
    fn_env = fn[3]
    new_env = Env(fn_env)
    for p, a in zip(params, args):
        new_env.set(p[1], a)
    return eval_list(body, new_env)
```

Cell's evaluator

```
args = list((eval_expr(a, env) for a in expr[2]))
...
if fn[0] == "function": ...
elif fn[0] == "native":
    py_fn = fn[1]
    params = inspect.getargspec(py_fn).args
    fail_if_wrong_number_of_args(expr[1], params[1:], args)
    return fn[1](env, *args)
```

Cell's evaluator

```
args = list((eval_expr(a, env) for a in expr[2]))
...
if fn[0] == "function": ...
elif fn[0] == "native":
else:
    raise Exception( "Not a function: %s" % str(fn))
```

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    elif typ == "function":  
        return ("function", expr[1], expr[2], Env(env))
```

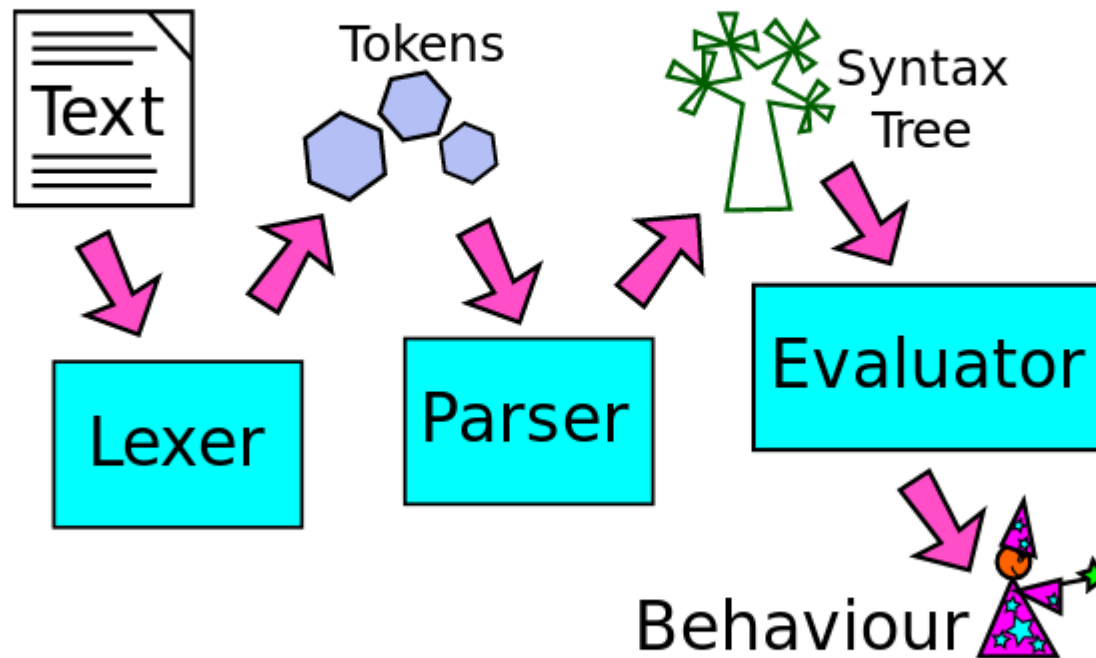

Cell's evaluator

```
def eval_expr(expr, env):  
    ...  
    else:  
        raise Exception("Unknown expression type: " + str(expr))
```

End result: value + side effects

print()

A programming language



Discussion

github.com/andybalaam/cell

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#)

