Colossal Cave Adventure in Python
. . . in the browser!

Christopher Swenson

PyCon; May 12, 2018

The What

What is this talk?

Colossal Cave Adventure, the PDP-10, FORTRAN IV, and a Python interpreter written in JavaScript.

Who is this talk for?

Curious programmery people

Slides available on GitHub

`https://github.com/swenson/adventure-talk-pycon`

Alternative titles

- Being lazy in the hardest way possible
- Adventure: The Programming Turducken
- FORthonScript
- Full-stack FORTRAN IV

The Who

Christopher Swenson, Ph.D

Currently at Twilio (prev. Google, Government, Simple)

Occasional BeeWare core contributor and PyDX organizer

I love programming languages and stuff.

Motivation

Idea: write a game with text messaging!

Motivation

Idea: write a game with text messaging!

. . . why not “port” the first text adventure?!

ADVENTURE

ADVENTURE

a.k.a., Colossal Cave

1976 text adventure, probably the first

Wildly popular and influential

Written in FORTRAN IV for the PDP-10

Text to +1 (669) 238-3683 to play now!

Or play on the web:
`https://swenson.github.io/adventurejs/`

ADVENTURE beginning

```
SOMEWHERE NEARBY IS COLOSSAL CAVE, WHERE OTHERS HAVE FOUND
FORTUNES IN TREASURE AND GOLD, THOUGH IT IS RUMORED
THAT SOME WHO ENTER ARE NEVER SEEN AGAIN. MAGIC IS SAID
TO WORK IN THE CAVE. I WILL BE YOUR EYES AND HANDS. DIRECT
ME WITH COMMANDS OF 1 OR 2 WORDS.
(ERRORS, SUGGESTIONS, COMPLAINTS TO CROWTHER)
(IF STUCK TYPE HELP FOR SOME HINTS)

YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK
BUILDING . AROUND YOU IS A FOREST. A SMALL
STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.
```

PDP-10

Pic from `http://www.columbia.edu/cu/computinghistory/pdp10.html`

PDP-10 FORTRAN IV

We're talking all the good stuff:

- All caps
- No recursion
- No indentation
- Line numbers
- Spaces don't matter
- Punch cards
- Tab = 6 spaces

Code

```
C ADVENTURES
        IMPLICIT INTEGER(A-Z)
        REAL RAN
        COMMON RTEXT,LLINE
        DIMENSION IOBJ(300),ICHAIN(100),IPLACE(100)
        1 ,IFIXED(100),COND(300),PROP(100),ABB(300),LLINE
            (1000,22)
        2 ,LTEXT(300),STEXT(300),KEY(300),DEFAULT(300),TRAVEL
            (1000)
        3 ,TK(25),KTAB(1000),ATAB(1000),BTEXT(200),DSEEN(10)
        4 ,DLOC(10),ODLOC(10),DTRAV(20),RTEXT(100),JSPKT(100)
        5 ,IPLT(100),IFIXT(100)
```

Code (cont'd.)

Or possibly:

```
C ADVENTURES
      IMPLICIT INTEGER(A-Z)
      REAL RAN
      COMMON RTEXT,LLINE
      DIMENSION IOBJ(300),ICHAIN(100),IPLACE(100)
     1 ,IFIXED(100),COND(300),PROP(100),ABB(300),LLINE(1000,22)
     2 ,LTEXT(300),STEXT(300),KEY(300),DEFAULT(300),TRAVEL
       (1000)
     3 ,TK(25),KTAB(1000),ATAB(1000),BTEXT(200),DSEEN(10)
     4 ,DLOC(10),ODLOC(10),DTRAV(20),RTEXT(100),JSPKT(100)
     5 ,IPLT(100),IFIXT(100)
```

Code (cont'd.)

```
C READ THE PARAMETERS

        IF(SETUP.NE.0) GOTO 1
        SETUP=1
        KEYS=1
        LAMP=2
        GRATE=3
C ...
        DATA(JSPKT(I),I=1,16)
            /24,29,0,31,0,31,38,38,42,42,43,46,77,71
        1 ,73,75/
        DATA(IPLT(I),I=1,20)
            /3,3,8,10,11,14,13,9,15,18,19,17,27,28,29
        1 ,30,0,0,3,3/
```

Code (cont'd.)

```
      DO 1001 I=1,300
      STEXT(I)=0
      IF(I.LE.200) BTEXT(I)=0
      IF(I.LE.100)RTEXT(I)=0
1001  LTEXT(I)=0
```

Code (cont'd.)

```
1002     READ(1,1003) IKIND
1003     FORMAT(G)
```

Computed GOTO

```
GOTO(1100,1004,1004,1013,1020,1004,1004)(IKIND+1)
```

Reading data

```
1004      READ(1,1005)JKIND,(LLINE(I,J),J=3,22)
1005      FORMAT(1G,20A5)
```

Calling subroutines

```
1         CALL YES(65,1,0,YEA)
```

Subroutines

```
      SUBROUTINE YES(X,Y,Z,YEA)
      IMPLICIT INTEGER(A-Z)
      CALL SPEAK(X)
      CALL GETIN(JUNK,IA1,JUNK,IB1)
      IF(IA1.EQ.'NO'.OR.IA1.EQ.'N') GOTO 1
      YEA=1
      IF(Y.NE.0) CALL SPEAK(Y)
      RETURN
1     YEA=0
      IF(Z.NE.0)CALL SPEAK(Z)
      RETURN
      END
```

36-bit Words

Pre-1980 or so, many different default word sizes

Nowadays, 8/16/32/64/128/256 are common

DEC (PDP, VAX) used 12, 36, 32

PDP-10 uses 36-bit words

PDP-10 (1966) used 7-bit ASCII from 1963

36-bit ASCII???

Packed left-to-right, 1 pad bit on the right

A	B	C	D	E	–
1 0 0 0 0 0 1	1 0 0 0 0 1 0	1 0 0 0 0 1 1	1 0 0 0 1 0 0	1 0 0 0 1 0 1	0

Why does it matter?

Because the program tokenizes user input itself!

```
      SUBROUTINE GETIN(TWOW,B,C,D)
      IMPLICIT INTEGER(A-Z)
      DIMENSION A(5),M2(6)
      DATA M2/"4000000000,"20000000,"100000,"400,"2,0/
6     ACCEPT 1,(A(I), I=1,4)
1     FORMAT(4A5)
      TWOW=0
      S=0
      B=A(1)
      DO 2 J=1,4
      DO 2 K=1,5
      MASK1="774000000000
      IF(K.NE.1) MASK1="177*M2(K)
      IF(((A(J).XOR."201004020100).AND.MASK1).EQ.0)GOTO 3
      IF(S.EQ.0) GOTO 2
      TWOW=1
      CALL SHIFT(A(J),7*(K-1),XX)
      CALL SHIFT(A(J+1),7*(K-6),YY)
      MASK=-M2(6-K)
      C=(XX.AND.MASK)+(YY.AND.(-2-MASK))
      GOTO 4
3     IF(S.EQ.1) GOTO 2
      S=1
      IF(J.EQ.1) B=(B.AND.-M2(K)).OR.("201004020100.AND.
      1 (-M2(K).XOR.-1))
2     CONTINUE
4     D=A(2)
      RETURN
      END
```

Code (cont'd.)

```
PAUSE 'INIT DONE'
```

Compilers

How a normal compiler works

1. Scan text into token stream
2. Parse tokens into syntax tree
3. Optimize syntax tree
4. Generate code

Compilers (cont'd.)

But that just sounds exhausting

And I only have a few days

Quick and Dirty Compiler

General strategy for coding a quick-and-dirty compiler

1. Split by lines
2. Split line by whitespace, commas, parens
3. Check for which statement this is
4. Parse the line

Python namedtuple

Python namedtuple is your friend

```
# raw lines
Line = namedtuple(’Line’, ’comment,label,continuation,
    statements’.split(’,’))

# lexical analysis
Token = namedtuple(’Token’, [’name’, ’value’])
```

Pseudo-grammar

Build a pseudo-grammar

```
# grammar structure
If = namedtuple(’If’, [’expr’, ’statement’])
IfNum = namedtuple(’IfNum’, [’expr’, ’neg’, ’zero’, ’pos’])
Goto = namedtuple(’Goto’, [’labels’, ’choice’])
Assign = namedtuple(’Assign’, [’lhs’, ’rhs’])
Comparison = namedtuple(’Compare’, [’a’, ’op’, ’b’])
Name = namedtuple(’Name’, [’name’])
Int = namedtuple(’Int’, [’value’])
Float = namedtuple(’Float’, [’value’])
# ...
```

Load data and source code

Load the “tape drive” and source code

```
# code and data
with open(’advdat.77-03-31.txt’) as fin:
    data = fin.read()
# remove blank line
data = data.replace(’\n\n’, ’\n’)

with open(’advf4.77-03-31.txt’) as fin:
    code = fin.read()

# ...
lines = combine_lines(parse_lines(code))
```

Lexical Analysis

Scanning

```
# lexical analysis

def parse_lines(text):
    return [parse_line(line) for line in text.split(’\n’)]

def parse_line(line):
    comment = False
    line = line.replace(’\t’, ’ ’ * 8)
    if not line:
        return commentLine
    if line[0] == ’C’ or line[0] == ’*’:
        return commentLine
    label = line[0:5].strip()
    if label:
        label = int(label)
```

Lexical Analysis (cont'd.)

Continuations

```
continuation = line[5] != ' '
statements = line[6:].strip()
if statements[0].isdigit() and statements[1] == ' ':
    continuation = True
    statements = statements[2:]
return Line(comment, label, continuation, statements)
```

Main loop

execute loop

```
def execute(self, current):
  next = self.execute_statement(self.prog[current], current)
  if next is None:
      next = self.current + 1
  if next == -1 or \
      (self.dostack and self.dostack[-1][1] == self.current and
           next == self.current + 1):
      # return to the beginning of the Do
      return self.dostack[-1][0]
  return next
```

Giant switch

Statement switch

```
def execute_statement(self, stmt, current):
  if isinstance(stmt, If):
      expr = self.eval_expr(stmt.expr)
      if isinstance(expr, bool) or isinstance(expr, int):
          if expr:
              return self.execute_statement(stmt.statement,
                  current)
          else:
              return
```

Expressions

Expression evaluation

```
def eval_expr(self, expr):
  if isinstance(expr, int):
      return expr
  if isinstance(expr, str):
      return expr
  if isinstance(expr, Op):
      a = self.eval_expr(expr.a)
      b = self.eval_expr(expr.b)
      if expr.op == ’.XOR.’:
          if isinstance(a, str):
              a = string_to_dec_num(a)
          if isinstance(b, str):
              b = string_to_dec_num(b)
          return a ˆ b
```

Statements

Statement parsing

```
def parse_statement(self, statement):
  if statement.startswith(’IF ’) or statement.startswith(’IF(’)
      :
      # parse if-statement
      statement = statement[2:].strip()
      r = match_right_paren(statement)
      expr = parse_expr(statement[1:r].strip())
      stmt = statement[r+1:].strip()
      if numericIfRegex.match(stmt):
          # numerical if
          m = numericIfRegex.match(stmt)
          a, b, c = int(m.group(1)), int(m.group(2)), int(m.
              group(3))
          return IfNum(expr, a, b, c)
      stmt = self.parse_statement(stmt)
      return If(expr, stmt)
```

Printing

Type statement

```
def execute_type(self, format, vars):
  if isinstance(vars, ArrayRange): # hack ...
  ai, vi = 0, 0
  while ai < len(format.args) and vi < len(vars):
      arg = format.args[ai]
      ai += 1
      if isinstance(arg, AsciiFormat):
          for c in xrange(arg.count):
            if vi >= len(vars): break
            var = vars[vi]
            vi += 1
            self.handler.write(to_string(self.eval_expr(var)))
          continue
      elif isinstance(arg, String):
          self.handler.write(arg.value)
          continue
      print ’halt on format’, format, vars
      exit()
```

Outer main loop

Main main loop

```
def go(self):
  self.current_subroutine = ’__main__’
  while True:
      self.current = self.execute(self.current)
```

Reading keyboard

Keyboard input

```
def execute_accept(self, format, vars):
  if isinstance(vars, ArrayRange): # hack ...
  self.waiting_for_user = True
  line = self.handler.read()
  self.waiting_for_user = False
  old_data, old_data_cursor = self.data, self.data_cursor
  self.data, self.data_cursor = line, 0
  for ai in range(len(format.args)):
      vi = 0
      arg = format.args[ai]
      if isinstance(arg, AsciiFormat):
          for c in xrange(arg.count):
              var = vars[vi]
              vi += 1
              chars = self.read_chars(int(arg.read)).upper()
              self.assign(var, chars)
          continue
  self.data, self.data_cursor = old_data, old_data_cursor
```

Interfaces

Three interfaces we need

- Tape
- Teletype input
- Teletype output

SMS!

Can use Twilio to make an SMS app to play

Host on Heroku with a little Flask app

Structured so that the state can be serialized, saved for each phone number

Flask app

Flask

```
@app.route("/incoming-sms", methods=[’GET’, ’POST’])
def sms_reply():
    try:
        cur = conn.cursor()

        from_ = str(request.values.get(’From’))
        inp = str(request.values.get(’Body’, ’’)).upper().strip
            ()
        inp = inp[:20] # commands shouldn’t be longer than this

        cur.execute("SELECT state FROM adventure WHERE num = %s
            ", (from_,))
        row = cur.fetchone()
        exists = row is not None
        ignore_input = False
```

Flask app

Flask

```
if inp == ’RESET’ or inp == ’QUIT’:
    if from_ in states:
        del states[from_]
        exists = False # force a reset
        cur.execute("DELETE FROM adventure WHERE num = %s",
             (from_,))
if not exists:
    print ’starting new game for’, from_
    handler = TwilioHandler()
    game = Game(handler)
    t = threading.Thread(target=game.go)
    t.daemon = True
    t.start()
    states[from_] = [handler, game, t]
    ignore_input = True
```

Flask app

Flask

```
if exists and from_ not in states:
    # load from backup
    handler = TwilioHandler()
    game = Game(handler)
    t = threading.Thread(target=game.go)
    t.daemon = True
    t.start()
    states[from_] = [handler, game, t]
    # wait fot it to boot
    while not game.waiting():
        time.sleep(0.001)
    # empty the queues
    while not handler.outqueue.empty():
        handler.outqueue.get_nowait()
    game.setstate(row[0])
    states[from_] = [handler, game, t]
```

Flask app

Flask

```
handler, game, _ = states[from_]
if not ignore_input:
    handler.inqueue.put(inp)
time.sleep(0.001)
while not game.waiting():
    time.sleep(0.001)
text = ’’
while not text:
    while not handler.outqueue.empty():
        text += handler.outqueue.get()
        time.sleep(0.001)
```

Flask app

Flask

```
    # now save the game state to the database
    state = game.getstate()
    if exists:
        cur.execute("UPDATE adventure SET state = %s, modified
            = NOW() WHERE num = %s", (psycopg2.Binary(state),
            from_))
    else:
        cur.execute("INSERT INTO adventure (num, state) VALUES
            (%s,%s)", (from_, psycopg2.Binary(state)))
    conn.commit()

    resp = twiml.Response()
    resp.message(text)
    return str(resp)
finally:
    cur.close()
```

State Saving

State Saving

```
# state
class Game(object):
    def getstate(self):
        d = dict(
            globals=self.globals,
            subroutines=self.subroutines,
            substack=self.substack,
            stmtstack=self.stmtstack,
            current=self.current,
            varstack=self.varstack,
            progstack=self.progstack,
            dostack=self.dostack,
            prog=self.prog,
            labels=self.labels,
            current_subroutine=self.current_subroutine,
            waiting_for_user=self.waiting_for_user)
        return bz2.compress(pickle.dumps(d))
```

In the browser?

The title of the talk says "in the browser", so where does that come in?

BeeWare Batavia!

Batavia

Batavia is a Python bytecode interpreter written in JavaScript, so that you can run Python in the browser or in Node.

Batavia

Batavia is a Python bytecode interpreter written in JavaScript, so that you can run Python in the browser or in Node.

. . . It technically works.

Batavia challenges

JS and Python concurrency models don't align. Like, at all.

JS expects a callback soup, but Python expects to be interrupted all the time.

Current Batavia has some challenges due to lack of callbacks. That's okay, I just added some.

Bytecode only

Batavia is still in early stages, and can only execute bytecode.

It cannot parse Python.

But is otherwise relatively feature-complete.

namedtuple

namedtuple

namedtuple

```
def namedtuple(typename, field_names):
  class_definition = _class_template.format(
    typename = typename,
    field_names = tuple(field_names),
    num_fields = len(field_names),
    arg_list = repr(tuple(field_names)).replace("’", "")[1:-1],
    repr_fmt = ’, ’.join(_repr_template.format(name=name)
                         for name in field_names),
    field_defs = ’\n’.join(_field_template.format(index=index,
        name=name)
                           for index, name in enumerate(
                               field_names)))
  try:
    exec class_definition
  except SyntaxError as e:
    raise SyntaxError(e.message + ’:\n’ + class_definition)
  result = namespace[typename]
  return result
```

Other JS wats

You have to be very, very careful when converting between Python and JavaScript types.

wat

```
> (1 == 2) * -1
```

Other JS wats

wat

```
> (1 == 2) * -1
-0
```

Other JS wats

But it works!

Demo time! Visit at your own risk: `https://swenson.github.io/adventurejs/`