```
"""
=======================
INT BASIC aka GAME BASIC
=======================

Why INTEGER BASIC? Woz explains: "I wrote down a complete syntax chart of the
commands that were in the H-P BASIC manual, and I included floating point
arithmetic, decimal points, numbers and everything. Then I started thinking
it was going to take me a month longer - I could save a month if I left out
the floating point."

Wozniak was confident that a good mathematician can work around the
limitation of integers: "We made the first handheld scientific calculators at
H-P, that's what I was designing, and they would work with transcendental
numbers, like SIN and COSIN. And was everything floating point? No! We did
all the calculations inside our calculators digitally, for higher accuracy
and higher speed, with integers. I said, basically integers can solve
anything. I was a mathematician of the type that wanted to solve things with
integers. You could always have dollars and cents as separate integer numbers
- all you need for games is integers. So I thought, I'll save a month writing
my BASIC, and I'll have a chance to be known as the first one to write a
BASIC for the 6502 processor. I said: I'll become famous like Bill Gates if I
write it the fastest I could. So I stripped out the floating point."

Lastly, Woz reminisces I liked Integer BASIC. There's a lot of things you
could do to save time, but you have to think mathematically. And most people
would rather just have the easy world - so it wasn't good for most people.
But for my type of person, Integer was great. I would want to do things with
integers."

Given that it's been 50 years since BASIC was formed, does he believe it
still has a place in the world? "I think it does. I still recommend it
frequently, as the right way to start programming classes. Or at least a
simpler language like BASIC - it's probably pretty hard to find the exact,
plain old original BASIC in this day of graphics on computers. But yes, I do.

"But as far as the introduction to computing... To me BASIC and FORTRAN are
the same. Either one of those, that's the right way to start, and not a real
super structured language where you have to learn so much about the
structure. It's better to learn structure from the ground up, the basic
atoms. Which is what BASIC is. To learn the structure from the ground up,
once you've learned it you will apply it in a structured language. Then
you're ready for it.

~~~~~~

Implements Woz's Integer BASIC with a few additional commands from Applesoft
```

```
48   BASIC (AS) or GW-BASIC (GW):
49
50   DATA (AS) : Define inline data; can be literals (unquoted strings), strings or numbers
51   READ (AS) : Read the next DATA value
52   RESTORE (AS) : Restore the DATA pointer to the first value
53   HOME (AS) : Clear text display
54   LOCATE r,c (GW) : Move cursor to the specified position
55   GET (AS) : Read single key
56
57   OBS:
58   TAB x where x=1-40 same as HTAB (AS)
59   VTAB x where x=1-24
60   POP : Convert last GOSUB into a GOTO
61
62   Todo:
63   =====
64   MUL8(): 8-bit integer multiplication, result is 2-bytes
65   DIV8(): 8-bit integer division, result is 2-bytes
66   MOD8(): 8-bit integer modulo, result is 2-bytes
67   MUL() : 32-bit fixed-point multiplication (Woz's) 4-bytes, 16-bit:16-bit
68   DIV() : 32-bit fixed-point division (Woz's) 4-bytes, 16-bit:16-bit
69   FMUL(): 32-bit floating-point multiplication (Woz's) 4-bytes
70   FDIV(): 32-bit floating-point division (Woz's) 4-bytes
71   LOG,LN,ATN,COS,SIN,SQR,TAN,PI
72           (add all these to a math library on 8K HIGH RAM)
73   >>  : bitwise operator right shift
74   <<  : bitwise operator left shift
75   &   : bitwise operator AND
76   |   : bitwise operator OR
77   ^   : bitwise operator XOR
78   ~   : bitwise operator NOT
79   !   : ??
80   #   : NOT EQUAL
81   !=  : NOT EQUAL
82   ++
83   --
84   Add Zero Page variables
85   Add r0-r15 16-bit registers
86   Add .B suffix for unsigned byte (normal VAR is 16-bit)
87   Add ADD8(), SUB8(), MUL8() and DIV8() for .B
88   Add support for hexadecimals $ff5c
89   BASIC V2 commands: CHR$,GET,TIME,ASC
90
91   Fixme:
92   ======
93   DIM: Re-dimension of a pre-existing A$; memory problems
94
```

```
Notes:
======

Integer BASIC's string handling was based on the system in HP BASIC. This
treated string variables as arrays of characters which had to be DIMed prior
to use. This is similar to the model in C or Fortran 77. This is in contrast
to MS-like BASICs where strings are an intrinsic variable-length type. Before
MS-derived BASICs became the de facto standard, this style was not uncommon;
North Star BASIC and Atari BASIC used the same concept, as did others.

Strings in Integer Basic used a fixed amount of memory regardless of the
number of characters used within them, up to a maximum of 255 characters.
This had the advantage of avoiding the need for the garbage collection of the
heap that was notoriously slow in MS BASIC but meant that strings that
were shorter than the declared length was wasted.

Integer BASIC, as its name implies, uses signed integers as the basis for its
math package. These were stored internally as a 16-bit number, little-endian (as
is the 6502). This allowed a maximum value for any calculation between -32767
and 32767. No fraction, just QUOTIENT (/) and REMAINDER (MOD).

Only single-dimension arrays were allowed, limited in size only by the available
memory.

Integer BASIC used the parameter in RND(6) which returned an integer from 0 to 5.

The position of the controller could be read using the PDL function, passing
in the controller number, 0 or 1, like A=PDL(0):PRINT A, returning a value
between 0 and 255.

A=SCRN(X,Y) returned the color of the screen at X,Y.

Integer BASIC included a POP command to exit from loops. This popped the
topmost item off the FOR stack. Atari BASIC also supported the same command,
while North Star BASIC used EXIT.

Although Integer BASIC contained its own math routines, the Apple II ROMs
also included a complete floating-point library located in ROM memory between
$F425-F4FB and $F63D-F65D. The source code was included in the Apple II
manual. BASIC programs requiring floating-point calculations could CALL into
these routines.
"""

from lark import Lark, Tree, Token

try:
    input = raw_input   # For Python2 compatibility
```

```
142   except NameError:
143       pass
144
145   basic_grammar = """
146       start: line+
147
148       line: INT statement [(":" statement)*]
149
150       ?statement: ("HOME" | "CLS") -> home
151                 | "END" -> end
152                 | "DATA" constant [("," constant)*] -> data
153                 | "READ" ID [("," ID)*] -> read
154                 | "CALL" expression -> call
155                 | "GOTO" expression -> goto
156                 | "GOSUB" expression -> gosub
157                 | "RETURN" -> return
158                 | "POKE" expression "," expression -> poke
159                 | "TAB" expression -> tab
160                 | "VTAB" expression -> vtab
161                 | "DIM" (VAR_ID | STR_ID) "(" expression ")" [("," ID "(" expression ↵
      ↳             ")")*] -> dim
162                 | "INPUT" [STRING ","] ID [("," ID)*] -> input
163                 | ("PRINT" | "?") (expression (PRINT_OP expression?)*)? -> print
164                 | "IF" expression "THEN" (INT | statement) -> if
165                 | "NEXT" VAR_ID [("," VAR_ID)*] -> next
166                 | "FOR" VAR_ID "=" expression "TO" expression ("STEP" expression)? -> for
167                 | "COLOR" "=" expression -> color
168                 | "LET"? (VAR_ID | STR_ID) "=" expression -> assignment
169                 | COMMENT -> comment
170                 | "POP" -> pop
171                 | "GR" -> gr
172                 | "TEXT" -> text
173                 | "PLOT" expression "," expression -> plot
174                 | "HLIN" expression "," expression "AT" expression -> hlin
175                 | "VLIN" expression "," expression "AT" expression -> vlin
176                 | STR_ID "(" INT ")" "=" (STRING | STR_ID) -> concat
177
178       ?expression: or_exp
179
180       ?or_exp: [(expression "OR")*] and_exp
181
182       ?and_exp: [(and_exp "AND")*] not_exp
183
184       ?not_exp: "NOT" not_exp -> not
185               | compare_exp
186
187       ?compare_exp: [(compare_exp REL_OP)*] add_exp
```

```
188
189       ?add_exp: [(add_exp ADD_OP)*] mul_exp
190
191       ?mul_exp: [(mul_exp MUL_OP)*] neg_exp
192
193       ?neg_exp: "-" power_exp -> neg
194               | power_exp
195
196       ?power_exp: power_exp "^" sub_exp -> power
197                 | sub_exp
198
199       ?sub_exp: "(" expression ")"
200               | value
201
202       ?value: (VAR_ID | STR_ID)
203             | STR_ID "(" INT "," INT ")" -> substring
204             | "ABS" "(" expression ")" -> abs
205             | "LEN" "(" expression ")" -> len
206             | "PEEK" "(" expression ")" -> peek
207             | "RND" "(" expression ")" -> rnd
208             | "SGN" "(" expression ")" -> sgn
209             | "ASC" "(" expression ")" -> asc
210             | "PDL" "(" expression ")" -> pdl
211             | "SCRN" "(" expression "," expression ")"
212             | constant
213
214       ?constant: INT
215                | STRING
216
217       PRINT_OP: "," | ";"
218       REL_OP: "=" | "#" | "!=" | ">=" | ">" | "<=" | "<>" | "<"
219       ADD_OP: "+" | "-"
220       MUL_OP: "*" | "/" | "%" | "MOD"
221       VAR_ID: (LETTER)(LETTER|INT)*
222       STR_ID: VAR_ID "$"
223       ID: STR_ID | VAR_ID
224       STRING: "\\"" /.*?/ "\\""
225       COMMENT: "REM" /[^\\n]/*
226
227       %import common.LETTER
228       %import common.INT
229       %import common.WS
230       %ignore WS
231   """
232
233   parser = Lark(basic_grammar)
234
```

```
235   text = '''
236   10 REM FOR LOOP
237   20 FOR I = 1 TO 10 STEP 2
238   23 FOR J = 1 TO 3
239   25 PRINT "INSIDE LOOP"
240   28 NEXT J
241   30 PRINT "THE", "END"
242   40 NEXT I
243   50 END
244   '''
245
246   print(text)
247   print(parser.parse(text).pretty())
248
249   str_count = 0       # counter for string labels L0, L1, ...
250   str_list = []       # list of strings to use .byte
251   var_id_list = []    # list of VAR_ID
252   dim_list = {}       # dictionary of STR_ID with size
253   #loop_count = 0
254   loop_list = []
255
256   ###
257   ### COMPILER FUNCTION
258   ###
259   def compile(t):
260       global str_count
261       global str_list     # STRING
262       global var_id_list  # VAR_ID
263       global dim_list     # STR_ID
264       #global loop_count
265       global loop_list
266
267       ###
268       ### PROCESS TOKEN OBJECT = <class 'lark.lexer.Token'>
269       ###
270       if isinstance(t, Token):
271           # GAMBLE: will put all INT and VAR into stack HERE. Seems to work well!
272           if t.type == 'INT':              # lval is an INT
273               print("\t\tPushInt " + t)
274           elif t.type == 'VAR_ID':
275               print("\t\tPushVar " + t)
276           # GAMBLE 2: need to return STR_ID or STRING for other routines??? e.g. PRINT ↵
      ↳        and INPUT
277           else:
278               return t
279
280       ###
```

```
281         ### PROCESS TREE OBJECT = <class 'lark.tree.Tree'>
282         ###
283         elif isinstance(t, Tree):
284             ###
285             ### LINE NUMBER
286             ###
287             if t.data == 'line':
288                 line_num = t.children.pop(0)
289                 print("L" + line_num + ":", end="")
290                 for cmd in t.children:
291                     compile(cmd)
292 
293             ###
294             ### REM
295             ###
296             elif t.data == 'comment':
297                 print("\t\t; " + t.children[0])
298 
299             ###
300             ### HOME
301             ###
302             elif t.data == 'home':
303                 print("\t\tlda #HOME")
304                 print("\t\tjsr CHROUT")
305 
306             ###
307             ### END
308             ###
309             elif t.data == 'end':
310                 print("\t\trts")
311 
312             ###
313             ### ABS
314             ###
315             elif t.data == 'abs':
316                 compile(t.children[0])
317                 print("\t\tjsr ABS")
318 
319             ###
320             ### DIM
321             ###
322             elif t.data == 'dim':
323                 dim_name = t.children[0]
324                 dim_size = compile(t.children[1])
325                 if dim_name not in dim_list:                # dim_name is new
326                     dim_list.update({dim_name:dim_size})    # Add dim_name to dim_list
327                 else:
```

```
328                     print('*** DIM ERR')
329                     print("Error: DIM redimensioning not implemented")
330                     exit()
331                 print("\t\t; DIM " + dim_name + "(" + dim_size + ")")
332 
333             ###
334             ### ASSIGNMENT
335             ###
336             elif t.data == 'assignment':
337                 var_name = t.children[0]
338                 if var_name.type == 'VAR_ID':
339                     if var_name not in var_id_list:
340                         var_id_list.append(var_name)
341                     #
342                     # IMPLEMENT DIRECT ASSIGNMENT AS DONE IN TAB
343                     #
344                     compile(t.children[1])
345                     print("\t\tPullVar " + var_name)
346                 elif var_name.type == 'STR_ID':
347                     if var_name not in dim_list:
348                         print("Error: STR_ID not defined: " + var_name)
349                         exit()
350 
351             ###
352             ### FOR
353             ###
354             elif t.data == 'for':
355                 var_name = t.children[0]
356                 if var_name.type == 'VAR_ID':
357                     if var_name not in var_id_list:
358                         var_id_list.append(var_name)
359                     if (var_name + "END") not in var_id_list:
360                         var_id_list.append(var_name + "END")
361                     #if len(t.children) == 4:
362                     if (var_name + "STEP") not in var_id_list:
363                         var_id_list.append(var_name + "STEP")
364                     loop_list.append(var_name)  # keep track for NEXT tokens
365                     #print(len(t.children))
366                     loop_label = "LOOP" + str(len(loop_list)-1)
367                     #loop_count += 1
368                     #
369                     # IMPLEMENT DIRECT ASSIGNMENT AS DONE IN TAB
370                     #
371                     compile(t.children[1])
372                     print("\t\tPullVar " + var_name)
373                     compile(t.children[2])
374                     print("\t\tPullVar " + var_name + "END")
```

```
375                 if len(t.children) == 4:
376                     compile(t.children[3])
377                 else:
378                     print("\t\tPushInt 1")  # If STEP is not used, default is STEP=1
379                 print("\t\tPullVar " + var_name + "STEP")
380                 print(loop_label + ":")                     ### LOOP START
381                 print("\t\tPushVar " + var_name + "END")    ### Exit loop if (I > ↵
    ↳               IEND), same as IF (IEND < I)
382                 print("\t\tPushVar " + var_name)
383                 print("\t\tjsr LT")
384                 print("\t\tjsr PULL")
385                 print("\t\tlda r0L")
386                 #
387                 # BNE below has a range limit of [-128..127], for larger branches, ↵
    ↳               need a JMP hack.
388                 #
389                 #print("\t\tbne " + loop_label + "END")     ### Branch if ZERO flag ↵
    ↳               is CLEAR, thus TRUE (non-ZERO)
390                 print("\t\tbeq " + loop_label + "CONT")
391                 print("\t\tjmp " + loop_label + "END")
392                 print(loop_label + "CONT:")
393
394         ###
395         ### NEXT
396         ###
397         elif t.data == 'next':
398             if len(t.children) > 1:
399                 print(t.children)
400             print("\t\tPushVar " + loop_list[-1])
401             print("\t\tPushVar " + loop_list[-1] + "STEP")
402             print("\t\tjsr ADD")
403             print("\t\tPullVar " + loop_list[-1])
404             print("\t\tjmp LOOP" + str(len(loop_list)-1))
405             print("LOOP" + str(len(loop_list)-1) + "END:")
406             del loop_list[-1]
407
408         ###
409         ### TAB
410         ###
411         elif t.data == 'tab':
412             #print(t.children[0])
413             if isinstance(t.children[0], Token):
414                 if t.children[0].type == 'INT':     # INT, no need for stack usage
415                     col = t.children[0].value
416                     print("\t\tsec")
417                     print("\t\tjsr PLOT")
418                     print("\t\tldy #" + col)
```

```
419                     print("\t\tclc")
420                     print("\t\tjsr PLOT")
421             else:                               # <expression>, need to get result ↵
      ↳         from stack
422                 compile(t.children[0])
423                 print("\t\tjsr PULL")
424                 print("\t\tsec")
425                 print("\t\tjsr PLOT")
426                 print("\t\tldy r0L")
427                 print("\t\tclc")
428                 print("\t\tjsr PLOT")
429
430         ###
431         ### VTAB
432         ###
433         elif t.data == 'vtab':
434             if t.children[0].type == 'INT':
435                 row = t.children[0].value
436             print("\t\tsec")
437             print("\t\tjsr PLOT")
438             print("\t\tldx #" + row)
439             print("\t\tclc")
440             print("\t\tjsr PLOT")
441             #
442             # IMPLEMENT <expression> ASSIGNMENT AS DONE IN TAB
443             #
444
445         ###
446         ### PRINT
447         ###
448         elif t.data == 'print':
449             if not t.children:
450                 #print("print: empty list")
451                 print("\t\tlda #NEWLINE")    # CR
452                 print("\t\tjsr CHROUT")
453             else:
454                 #str_label = "S" + str(str_count)
455                 #str_count = str_count + 1
456                 #if t.children[0].type == 'STRING':
457                     #str_list.append((t.children[0].value).lower())
458                     #print("\t\tLoadAddress " + str_label + "\t\t; to r0")
459                     #print("\t\tjsr PrString")
460                 for i in range(len(t.children)):
461                     #print(t.children[i])
462                     if t.children[i].type == 'STRING':
463                         str_label = "S" + str(str_count)
464                         str_count = str_count + 1
```

```
465                         str_list.append((t.children[i].value).lower())
466                         print("\t\tLoadAddress " + str_label + "\t\t; to r0")
467                         print("\t\tjsr PrString")
468                     if t.children[i] == ',':
469                         #print("\t\tlda #TAB")    # TAB
470                         print("\t\tlda #32\t\t; TAB not implemented, using
    SPACE")    # Use space until TAB is implemented
471                         print("\t\tjsr CHROUT")
472                 print("\t\tPrintNewline")
473 
474             #print(t.children)
475             #print(t.children[0].type)
476 
477         ###
478         ### MUL/DIV/MOD
479         ###
480         elif t.data == 'mul_exp':
481             compile(t.children[0])  # lval
482             compile(t.children[2])  # rval
483             if t.children[1] == '*':
484                 print("\t\tjsr UMUL")
485             elif t.children[1] == '/':
486                 print("\t\tjsr UDIV")
487             else:
488                 print("\t\tjsr UMOD")
489 
490         ###
491         ### ADD/SUB
492         ###
493         elif t.data == 'add_exp':
494             compile(t.children[0])
495             compile(t.children[2])
496             if t.children[1] == '+':
497                 print("\t\tjsr ADD")
498             else:
499                 print("\t\tjsr SUB")
500 
501         ###
502         ### UNKNOWN NODE TYPE
503         ###
504         else:
505             print("\t==>", t.data, t.children)
506 
507     ###
508     ### UNKNOWN OBJECT TYPE
509     ###
510     else:
```

```
511             print("Unknown Object: <not TREE nor TOKEN>:", t)
512
513     ###
514     ### MAIN BODY
515     ###
516     parse_tree = parser.parse(text)
517     print(parse_tree)
518
519     print('\n.include \"macros.inc\"')
520     print('.include \"header.inc\"')
521     print(".code\n")
522
523     for inst in parse_tree.children:
524         compile(inst)
525
526     print()
527     for idx, val in enumerate(str_list):
528         print("S" + str(idx) + ":\t\t.asciiz " + val)
529     for var in var_id_list:
530         print(var + ":\t\t.res 4")
531     for var in dim_list:
532         print("{}:\t\t.res {}".format(var,dim_list[var]))
533     #for var in loop_list:
534     #    print(var + ":\t\t.res 4")
535
536     print('\n.include \"io.asm\"')
537     print('.include \"math.asm\"')
538
```