SPL#

a Forth-style concatenative stack language. The compiler is (currently) written in Python and emits 6502 assembly code.

Originally created by Ron Kneusel for the Apple II, adapted for the Atari 8bit and enhanced by Carsten Strotmann (cas@strotmann.de).

SPL is still work-in-progress, it has some "rough-edges" and might fail in certain places. It has many bugs. However, it works fine for some developers.

Applications in SPL:

* ATR Copy Center ACC - A copy tool for the SIO2USB device

SPL Source (Python)

```
#!/usr/bin/python
#
#  SPL compiler
#  RTK, 03-Sep-2007
#  Last update:  10-Aug-2012
#
#  $Id$
#
#  This code is freeware, do as you please with it.
#
############################################################

import os
import sys
import math
import time
from types import *

#  Modification date
LAST_MOD = "10-Aug-2012"


#---------------------------------------------------------------------
#  Configuration section.  Modify these values to reflect
#  your working environment and target system.
#---------------------------------------------------------------------

#  Assembler name and/or path.  If the executable is in your
#  $PATH just the name is needed.  If on Windows, a good place
#  to put the as6502.exe file is in the C:\WINDOWS\ directory in
#  which case all you need is "as6502" here.
ASSEMBLER = "atasm"

#  Library and include paths.  If you have environment variables set
#  for these, they will be used.  If you do not set environment variables
#  the default value will be used.  Therefore, set the default value to
#  the full pathname, trailing / or \ included, to the place where you
#  set up the directories, if not using environment variables.
LIB_DIR = os.getenv("SPL_LIB_PATH", default="lib/")
INCLUDE_DIR = os.getenv("SPL_INCLUDE_PATH", default="include/")
```

```
#  Pathname of blank ProDOS disk image
DISK_IMAGE = LIB_DIR + "blank.dsk"

#  Default base output file name and type
BASE_NAME = "out"
BASE_TYPE = "bin"

#  Compiler version number
VERSION = "1.0atari"

#  Header for assembly output files, must be a list of strings
HEADER = [
    "",
    "Output from SPL compiler version " + VERSION,
    "Generated on " + time.ctime(time.time()),
    ""
]

#  Code start and stack location in RAM
ORG   = 0x2200

#  Start of variable space, grows downwards
VARTOP= 0x8000  #  doesn't clobber the ProDOS BASIC interpreter

#  Library equates - case matters
EQUATES = {
    "stack"  : ORG,   #  stack address
    "sp"     : 0x80,    #  stack pointer
    "ex"     : 0x81,    #  xreg value
    "ey"     : 0x82,    #  yreg value
    "ea"     : 0x83,    #  areg value
    "ta"     : 0x84,    #  and 7, scratch 1
    "tb"     : 0x86,    #  and 9, scratch 2
    "op1"    : 0x87,    #  1st operand (4 bytes)
    "op2"    : 0x8B,    #  2nd operand (4 bytes)
    "res"    : 0x8F,    #  result (8 bytes)
    "rem"    : 0x96,    #  remainder (4 bytes)
    "tmp"    : 0x9A,    #  scratch (4 bytes)
    "sign"   : 0x9F,    #  sign (1 byte)
    "outbuf" : 0xA0,    #  output text buffer (12 bytes)
    "inbuf"  : 0x400  #  input text buffer
}

#
#  Library words dependency table.  New library words (ie, in lib/)
#  must be added to this table.  What is listed is the name of every
#  routine called (JSR or JMP) by the library word.  This table is
#  used to include only the routines necessary for the program being compiled.
#
DEPENDENCIES = {
    "abs"      : ["get_ab","comp_tb","push"],
    "accept"   : ["get_ta","push"],
    "add1"     : [],
    "add"      : ["get_tb","pop","push"],
    "addstore" : ["get_ab"],
    "and"      : ["get_ab","push"],
    "booland"  : ["get_ab","push"],
    "areg"     : ["pop"],
```

```
"fetch"    : ["get_ta","push"],
"dfetch"   : ["get_ta","push"],
"cfetch"   : ["get_ta", "push"],
"b_and"    : ["get_tb","pop","push"],
"bclr"     : ["get_ab","push"],
"b_or"     : ["get_tb","push"],
"bset"     : ["get_ab","push"],
"btest"    : ["get_ab","push"],
"btoa"     : ["ptrout"],
"btod"     : [],
"b_xor"    : ["get_tb","pop","push"],
"bye"      : [],
"call"     : ["pop"],
"chout"    : [],
"ch"       : ["pop"],
"cls"      : [],
"cmove"    : ["get_ta", "get_tb"],
"cmoveb"   : ["get_ta", "get_op1"],
"comp"     : ["pop","push"],
"comp_ta"  : [],
"comp_tb"  : [],
"copy"     : [],
"count"    : ["get_ta", "push"],
"cprhex"   : [],
"cr"       : [],
"crson"    : [],
"crsoff"   : [],
"ctoggle"  : ["pop", "get_ta"],
"csub"     : [],
"cv"       : ["pop"],
"d32"      : [],
"dabs"     : ["get_op1","push_op1"],
"d_add"    : [],
"dadd"     : ["get_ops","push_res"],
"date"     : ["push"],
"ddivide"  : ["ddiv","push"],
"ddivmod"  : ["ddiv","push"],
"ddiv"     : ["get_ops","d32","neg"],
"depth"    : ["push"],
"d_eq"     : ["zerores"],
"deq"      : ["get_ops","d_eq","push"],
"d_ge"     : ["d_eq","d_gt"],
"dge"      : ["get_ops","d_ge","push"],
"d_gt"     : ["d_sub","iszero","zerores"],
"dgt"      : ["get_ops","d_gt","push"],
"disp"     : ["get_ta","chout","udiv16","push"],
"div"      : ["get_ta","comp_ta","comp_tb","udiv16","push"],
"d_le"     : ["d_eq","d_lt"],
"dle"      : ["get_ops","d_le","push"],
"d_lt"     : ["d_sub","zerores"],
"dlt"      : ["get_ops","d_lt","push"],
"dmod"     : ["ddiv","push"],
"dmult"    : ["get_ops","neg","m32","push_res"],
"dnegate"  : ["get_op1","neg","op1res","push_res"],
"d_ne"     : ["d_eq"],
"dne"      : ["get_ops","d_ne","push"],
"decaddr"  : ["get_ta"],
"deccaddr" : ["pop"],
"dnumber"  : ["pop","neg","push"],
```

```
"dprhex"   : ["get_op1", "cprhex"],
"dprint"   : ["get_op2","chout","neg","pntres"],
"drop2"    : ["pop"],
"drop"     : ["pop"],
"d_sqrt"   : ["d_sub"],
"dsqrt"    : ["get_op1","d_sqrt","push_res"],
"d_sub"    : [],
"dsub"     : ["get_ops","d_sub","push_res"],
"dtos"     : ["pop"],
"dup2"     : ["push"],
"duprint"  : ["get_op2","btod","pntres"],
"dprint"   : ["get_op2","chout","neg","btod","pntres"],
"dup"      : ["push"],
"emit"     : ["pop","chout"],
"eq"       : ["get_ab","push"],
"exit"     : [],
"execute"  : ["pop"],
"erase"    : ["pop","get_ab"],
"fill"     : ["pop","get_ab"],
"fclose"   : ["pop","push"],
"fcreate"  : ["pop","push"],
"fdestroy" : ["pop","push"],
"feof"     : ["pop","push"],
"fetchend" : ["push"],
"fflush"   : ["pop","push"],
"fgetc"    : ["pop","push"],
"fread"    : ["pop","push"],
"fopen"    : ["pop","push","pdos_addr"],
"fputc"    : ["pop","push"],
"fseek"    : ["pop","push"],
"ftell"    : ["pop","push"],
"fwrite"   : ["pop","push"],
"ge"       : ["get_ab","csub","push"],
"get_ab"   : ["get_tb","get_ta"],
"get_op1"  : ["pop"],
"get_op2"  : ["pop"],
"get_ops"  : ["get_op2","get_op1"],
"get_ta"   : ["pop"],
"get_tb"   : ["pop"],
"get_file_info" : ["pop","push"],
"getcwd"   : ["pop","push"],
"gt"       : ["get_ab","csub","push"],
"incaddr"  : ["get_ta"],
"inccaddr" : ["pop"],
"input_s"  : ["push"],
"inverse"  : [],
"iszero"   : [],
"keyp"     : ["rdykey","push"],
"key"      : ["rdkey","push"],
"le"       : ["get_ab","csub","push"],
"lt"       : ["get_ab","csub","push"],
"m32"      : ["zerores"],
"minus1"   : ["pop","push"],
"minus2"   : ["pop","push"],
"mod"      : ["get_ab","comp_ta","comp_tb","udiv16","push"],
"mult"     : ["get_ab","comp_ta","comp_tb","umult16","push"],
"negate"   : ["get_ta","comp_ta","push"],
"neg"      : ["add1"],
"ne"       : ["get_ab","push"],
```

```
"nip"      : ["get_ta","pop","push"],
"normal"   : [],
"not"      : ["get_ta","push"],
"n_str"    : ["get_ta","ptrout","comp_ta","u_str"],
"number"   : ["pop","comp_ta","push"],
"op1res"   : [],
"or"       : ["get_ab","push"],
"over"     : ["push"],
"pad"      : ["push"],
"pdos_addr": [],
"plus1"    : ["get_ta","push"],
"plus2"    : ["get_ta","push"],
"pntres"   : ["chout"],
"pop"      : [],
"pos"      : ["pop"],
"prbuf"    : ["chout"],
"prhex2"   : ["pop", "cprhex"],
"prhex"    : ["pop", "cprhex"],
"primm"    : ["chout"],
"print"    : ["n_str","prbuf"],
"ptrout"   : [],
"push_op1" : ["push"],
"push_op2" : ["push"],
"push_rem" : ["push"],
"push_res" : ["push"],
"push"     : [],
"quit"     : [],
"read_block" : ["pop","push"],
"rename"   : ["pop","push"],
"rdkey"    : [],
"rdykey"   : [],
"reset"    : [],
"rot"      : ["get_ta","get_tb","pop","push"],
"setcwd"   : ["pop","push"],
"space"    : ["chout"],
"set_file_info" : ["pop","push"],
"spfetch"  : ["push"],
"stod"     : ["push"],
"strcmp"   : ["get_ab","push"],
"strcpy"   : ["get_ab"],
"strlen"   : ["get_ta","push"],
"strmatch" : ["get_ab","push"],
"strpos"   : ["get_ab","push"],
"sub1"     : [],
"sub"      : ["get_ab","csub","push"],
"swapcell" : [],
"swap2"    : ["get_ta","get_tb","pop","push"],
"swap"     : ["get_ta","get_tb","push"],
"time"     : ["push"],
"to_r"     : ["pop"],
"from_r"   : ["push"],
"r_fetch"  : ["to_r", "from_r", "dup"],
"tooutbuf" : ["u_str"],
"store16"  : ["get_ta","pop"],
"store32"  : ["get_ta","pop"],
"store8"   : ["get_ta","pop"],
"udiv16"   : [],
"udiv"     : ["get_ab","udiv16","push"],
"ugt"      : ["get_ab","push"],
```

```
    "ult"      : ["get_ab","push"],
    "umod"     : ["get_ab","udiv16","push"],
    "umult16"  : [],
    "umult"    : ["get_ab","umult16","push"],
    "uncount"  : ["pop","push"],
    "uprint"   : ["u_str","prbuf"],
    "ushiftl"  : ["pop","push"],
    "ushiftr"  : ["pop","push"],
    "u_str"    : ["get_ta","btoa"],
    "within"   : ["over", "to_r", "from_r", "ult"],
    "write_block": ["pop","push"],
    "xor"      : ["get_ab","push"],
    "xreg"     : ["pop"],
    "yreg"     : ["pop"],
    "zerores"  : []
}


#
#  Mapping between library words and assembly labels.
#
LIBRARYMAP = {
    "abs"      : "abs",
    "accept"   : "accept",
    "+"        : "add",
    "++"       : "incaddr",
    "c++"      : "inccaddr",
    "+!"       : "addstore",
    "areg"     : "areg",
    "at"       : "pos",
    "@"        : "fetch",
    "d@"       : "dfetch",
    "c@"       : "cfetch",
    ">outbuf"  : "tooutbuf",
    "and"      : "b_and",
    "bclr"     : "bclr",
    "or"       : "b_or",
    "bset"     : "bset",
    "btest"    : "btest",
    "bye"      : "bye",
    "xor"      : "b_xor",
    "call"     : "call",
    "ch"       : "ch",
    "cls"      : "cls",
    "cmove"    : "cmove",
    "cmove>"   : "cmoveb",
    "count"    : "count",
    "~"        : "comp",
    "cr"       : "cr",
    "crson"    : "crson",
    "crsoff"   : "crsoff",
    "ctoggle"  : "ctoggle",
    "cv"       : "cv",
    "dabs"     : "dabs",
    "date"     : "date",
    "d+"       : "dadd",
    "d/"       : "ddivide",
    "d/mod"    : "ddivmod",
    "depth"    : "depth",
```

```
"d="       : "deq",
"d>="      : "dge",
"d>"       : "dgt",
"disp"     : "disp",
"d<="      : "dle",
"d<"       : "dlt",
"dmod"     : "dmod",
"d*"       : "dmult",
"dnegate"  : "dnegate",
"d<>"      : "dne",
"dnumber"  : "dnumber",
"d.$"      : "dprhex",
"d."       : "dprint",
"2drop"    : "drop2",
"drop"     : "drop",
"dsqrt"    : "dsqrt",
"d-"       : "dsub",
"2dup"     : "dup2",
"du."      : "duprint",
"dup"      : "dup",
"emit"     : "emit",
"end@"     : "fetchend",
"exit"     : "exit",
"erase"    : "erase",
"="        : "eq",
"execute"  : "execute",
"fclose"   : "fclose",
"fdestroy" : "fdestroy",
"feof"     : "feof",
"fflush"   : "fflush",
"finfo@"   : "get_file_info",
"finfo!"   : "set_file_info",
"fgetc"    : "fgetc",
"fill"     : "fill",
"fopen"    : "fopen",
"fputc"    : "fputc",
"fread"    : "fread",
"fseek"    : "fseek",
"fwrite"   : "fwrite",
"fcreate"  : "fcreate",
"ftell"    : "ftell",
"getcwd"   : "getcwd",
">="       : "ge",
">"        : "gt",
"input"    : "input_s",
"inverse"  : "inverse",
"keyp"     : "keyp",
"key"      : "key",
"/"        : "div",
"<="       : "le",
"<"        : "lt",
"1-"       : "minus1",
"2-"       : "minus2",
"mod"      : "mod",
"*"        : "mult",
"negate"   : "negate",
"<>"       : "ne",
"nip"      : "nip",
"normal"   : "normal",
```

```
    "not"      : "not",
    "number"   : "number",
    "over"     : "over",
    "pad"      : "pad",
    "1+"       : "plus1",
    "2+"       : "plus2",
    ".2$"      : "prhex2",
    ".$"       : "prhex",
    "."        : "print",
    "pos"      : "pos",
    "quit"     : "quit",
    "read_block" : "read_block",
    "rename"   : "rename",
    "reset"    : "reset",
    "rot"      : "rot",
    ">r"       : "to_r",
    "r>"       : "from_r",
    "r@"       : "r_fetch",
    "setcwd"   : "setcwd",
    "space"    : "space",
    "sp@"      : "spfetch",
    "strcmp"   : "strcmp",
    "strcpy"   : "strcpy",
    "strlen"   : "strlen",
    "strmatch" : "strmatch",
    "strpos"   : "strpos",
    "time"     : "time",
    "-"        : "sub",
    "--"       : "decaddr",
    "c--"      : "deccaddr",
    "2swap"    : "swap2",
    "swap"     : "swap",
    "!"        : "store16",
    "d!"       : "store32",
    "c!"       : "store8",
    "uncount"  : "uncount",
    "u/"       : "udiv",
    "u>"       : "ugt",
    "u<"       : "ult",
    "umod"     : "umod",
    "u*"       : "umult",
    "u."       : "uprint",
    "<<"       : "ushiftl",
    ">>"       : "ushiftr",
    "><"       : "swapcell",
    "within"   : "within",
    "write_block": "write_block",
    "xreg"     : "xreg",
    "yreg"     : "yreg"
}


#-----------------------------------------------------------------------
#  Compiler source code follows.
#-----------------------------------------------------------------------


##########################################################
#  SPLCompiler
```

```
#
class SPLCompiler:
    """Implements a compiler from SPL to 6502 assembly code."""


    #-------------------------------------------------
    #  GetLabel
    #
    def GetLabel(self, name):
        """Return a unique label."""

        label = "A%05d" % self.counter
        self.counter += 1
        return name+"_"+label


    #-------------------------------------------------
    #  Decimal
    #
    def Decimal(self, s):
        """Interpret s as a decimal number."""

        sign = +1
        start = 0

        if (s[0] == "-"):
            sign = -1
            start = 1
        elif (s[0] == "+"):
            start = 1

        i = start
        n = 0
        msg = "Illegal decimal number: "

        while (i < len(s)):
            if (s[i] in "0123456789"):
                n = 10*n + int(s[i])
            elif (s[i] == "#") or (s[i] == "%"):
                if (i != len(s)-1):
                    self.Error(msg + s)
            else:
                self.Error(msg + s)
            i += 1

        return n*sign


    #-------------------------------------------------
    #  Binary
    #
    def Binary(self, s):
        """Interpret s as a binary number."""

        n = i = 0
        msg = "Illegal binary number: "

        while (i < len(s)):
            if (s[i] in "01"):
```

```
            n = 2*n + int(s[i])
        elif (s[i] == "#") or (s[i] == "%"):
            if (i != len(s)-1):
                self.Error(msg + s)
        else:
            self.Error(msg + s)
        i += 1

    return n


#-------------------------------------------------
#  Hexadecimal
#
def Hexadecimal(self, s):
    """Interpret s as a hexadecimal number."""

    n = i = 0
    msg = "Illegal hexadecimal number: "

    while (i < len(s)):
        if (s[i] in "0123456789"):
            n = 16*n + int(s[i])
        elif (s[i].upper() in "ABCDEF"):
            n = 16*n + (ord(s[i].upper()) - ord("A") + 10)
        elif (s[i] == "#") or (s[i] == "%"):
            if (i != len(s)-1):
                self.Error(msg + s)
        else:
            self.Error(msg + s)
        i += 1

    return n


#-------------------------------------------------
#  Number
#
def Number(self, s):
    """Return s as a number, if possible."""

    if (type(s) != StringType):
        n = 0
    elif (s == ""):
        n = 0
    elif (len(s) < 2):
        n = self.Decimal(s)
    elif (s[0:2].upper() == "0X"):
        n = self.Hexadecimal(s[2:])
    elif (s[0] == "$"):
        n = self.Hexadecimal(s[1:])
    elif (s[0:2].upper() == "0B"):
        n = self.Binary(s[2:])
    elif (s[0] == "%"):
        n = self.Binary(s[1:])
    else:
        n = self.Decimal(s)
    return n
```

```
#-----------------------------------------------
#  CheckDecimal
#
def CheckDecimal(self, s):
    """True if string s is a valid decimal number."""

    try:
        n = int(s)
    except:
        return False
    return True


#-----------------------------------------------
#  CheckHexadecimal
#
def CheckHexadecimal(self, s):
    """True if string s is a valid hex number."""

    for t in s.upper():
        if (t not in "0123456789ABCDEF"):
            return False
    return True


#-----------------------------------------------
#  CheckBinary
#
def CheckBinary(self, s):
    """True if string s is a valid binary number."""

    for t in s:
        if (t not in "01"):
            return False
    return True


#-----------------------------------------------
#  isNumber
#
def isNumber(self, s):
    """Return true if string s is a valid number."""

    #  It must be a string
    if (type(s) != StringType):
        return False
    elif (s == ""):
        return False
    elif (len(s) < 2):
        return self.CheckDecimal(s)
    elif (s[0:2].upper() == "0X"):
        return self.CheckHexadecimal(s[2:])
    elif(s[0] == "$"):
        return self.CheckHexadecimal(s[2:])
    elif (s[0:2].upper() == "0B"):
        return self.CheckBinary(s[2:])
    elif (s[0] == "%"):
        return self.CheckBinary(s[2:])
```

```
    else:
        return True


#-------------------------------------------------
#  InvalidName
#
def InvalidName(self, name):
    """Returns true if the given name is not valid."""

    if (type(name) != StringType):
        return True
    if (name == ""):
        return True
    if ((name[0].upper() < "A") or (name[0].upper() > 'Z')) and (name[0] != "_"):
        return True

    for c in name:
        if (c.upper() not in "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_"):
            return True

    return False


#-------------------------------------------------
#  Error
#
def Error(self, msg):
    """Output an error message and quit."""

    raise ValueError("In " + self.funcName + " : " + self.Token + " : " + str(msg)


#-------------------------------------------------
#  ParseCommandLine
#
def ParseCommandLine(self):
    """Parse the command line arguments."""

    #  Defaults
    self.sys = False
    self.warn = False

    #  List of all source code names
    self.files = []

    #  Default output file base filename and type
    self.outname = BASE_NAME
    self.outtype = BASE_TYPE

    next = 0
    for s in sys.argv[1:]:
        if (s == "-o"):
            next = 1
        elif (s == "-t"):
            next = 2
        elif (s == "-org"):
            next = 3
        elif (s == "-var"):
```

```
            next = 4
        elif (s == "-stack"):
            next = 5
        elif (s == "-sys"):
            self.sys = True
        elif (s == "-warn"):
            self.warn = True
        elif (s == "-prodos"):
            self.VARTOP = 0x89ff
        else:
            if (next == 1):
                self.outname = s
                next = 0
            elif (next == 2):
                self.outtype = s
                next = 0
            elif (next == 3):
                self.cmdOrigin = self.Number(s)
                next = 0
            elif (next == 4):
                self.VARTOP = self.Number(s)
                next = 0
            elif (next == 5):
                EQUATES["stack"] = self.Number(s)
                next = 0
            else:
                self.files.append(s)


#-------------------------------------------------
#  LoadFiles
#
def LoadFiles(self):
    """Load the source code on the command line."""

    self.source = ""

    for fname in self.files:

        #  If no .spl extension, add it
        if fname.find(".spl") == -1:
            fname += ".spl"

        try:
            #  Open the file as given on the command line
            f = open(fname,"r")
        except:
            #  Is it in the include directory?
            try:
                f = open(INCLUDE_DIR + fname, "r")
            except:
                self.Error("Unable to locate %s" % fname)

        #  Append to the source string
        self.source += (f.read() + "\n")
        f.close()


#-------------------------------------------------
```

```
#  Tokenize
#
def Tokenize(self):
    """Tokenize the input source code."""

    if self.source == "":
        return

    self.tokens = []
    inToken = inString = inComment = inCode = False
    delim = ""
    t = ""

    for c in self.source:
        if (inString):
            if (c == delim):
                t += c
                self.tokens.append(t)
                t = ""
                inString = False
            else:
                t += c
        elif (inComment):
            if (c == "\n"):
                inComment = False
            elif (c == ")"):
                inComment = False
        elif (inToken):
            if (c <= " "):
                inToken = False
                self.tokens.append(t)
                t = ""
            else:
                t += c
        else:
            if (c == "["):
                inCode = True
            if (c == "]"):
                inCode = False
            if (c == '"') or (c == "'"):
                inString = True
                delim = c
                t += c
            elif (c == "#") and not inCode:
                inComment = True
            elif (c == "(") and not inCode:
                inComment = True
            elif (c == ")") and not inCode:
                inComment = False
            elif (c <= " "):
                pass  # ignore whitespace
            else:
                t += c
                inToken = True


#---------------------------------------------------
#  SetOrigin
#
```

```
    def SetOrigin(self):
        """Set the output origin position."""

        #  Use the command line setting, if given
        if (self.cmdOrigin != -1):
            self.org = self.cmdOrigin
        else:
            self.org = ORG

        next = indef = False

        for i in self.tokens:
            if (i == "org"):
                if (indef):
                    self.Error("Origin statements cannot be within functions.")
                next = True
            elif (i == "def"):
                indef = True
            elif (i == ":"):
                indef = True
            elif (i == "end"):
                indef = False
            elif (i == ";"):
                indef = False
            elif (next):
                self.org = self.Number(i)
                if (self.org < 0) or (self.org > 65535):
                    self.Error("Illegal origin address: " + str(self.org))
                return


    #-------------------------------------------------
    #  AddName
    #
    def AddName(self, name):
        """Add a new name to the names dictionary."""

        if self.names.has_key(name):
            self.Error("Duplicate name found: " + str(name))
#         self.names[name] = self.GetLabel(name)
        self.names[name] = name

    #-------------------------------------------------
    #  Declarations
    #
    def Declarations(self):
        """Locate all defined variables, constants and strings."""

        #  Dictionaries of all defined variables, and numeric and string constants
        #  accessed by name as key.
        self.vars = {}
        self.consts = {}
        self.str = {}
        indef = False
        i = 0

        while (i < len(self.tokens)):
            if (self.tokens[i] == "var"):
                if (indef):
```

```
                self.Error("Variables cannot be defined within functions.")
            if (i+2) >= len(self.tokens):
                self.Error("Syntax error: too few tokens for variable declaration.
            name = self.tokens[i+1]
            bytes= self.Number(self.tokens[i+2])
            i += 2
            if (self.InvalidName(name)):
                self.Error("Illegal variable name: " + str(name))
            if (bytes <= 0) or (bytes >= 65535):
                self.Error("Illegal variable size: " + name + ", size = " + str(by
            self.vars[name] = bytes
            self.symtbl[name] = "VAR"
            self.AddName(name)
        elif (self.tokens[i] == "const"):
            if (indef):
                self.Error("Constants cannot be defined within functions.")
            if (i+2) >= len(self.tokens):
                self.Error("Syntax error: too few tokens for constant declaration.
            name = self.tokens[i+1]
            val = self.Number(self.tokens[i+2])
            i += 2
            if (self.InvalidName(name)):
                self.Error("Illegal constant name: " + str(name))
            if (val < -32768) or (val >= 65535):
                self.Error("Illegal constant value: " + name + ", value = " + str(
            self.consts[name] = val
            self.symtbl[name] = "CONST"
            self.AddName(name)
        elif (self.tokens[i] == "str"):
            if (indef):
                self.Error("Strings cannot be defined within functions.")
            if (i+2) >= len(self.tokens):
                self.Error("Syntax error: too few tokens for string declaration.")
            name = self.tokens[i+1]
            val = self.tokens[i+2]
            i += 2
            if (self.InvalidName(name)):
                self.Error("Illegal string name: " + str(name))
            if (val == ""):
                self.Error("Illegal string value, name = " + str(name))
            self.str[name] = val
            self.symtbl[name] = "STR"
            self.AddName(name)
        elif (self.tokens[i] == "def"):
            indef = True
        elif (self.tokens[i] == ":"):
            indef = True
        elif (self.tokens[i] == "end"):
            indef = False
        elif (self.tokens[i] == ";"):
            indef = False

        i += 1


#-------------------------------------------------
#  ParseDataBlocks
#
def ParseDataBlocks(self):
```

```
    """Get all data blocks."""

    #  Dictionary of all data blocks.  Each entry stores the data values
    #  for that block.
    self.dataBlocks = {}

    indata = False
    i = 0
    name = ""
    data = []

    while (i < len(self.tokens)):
        if (self.tokens[i] == "data"):
            if (indata):
                self.Error("Data blocks may not be nested.")
            indata = True
            data = []
            if (i+1) >= len(self.tokens):
                self.Error("Too few tokens for data block declaration.")
            name = self.tokens[i+1]
            if (self.InvalidName(name)):
                print i
                self.Error("Illegal data block name: " + str(name))
            i += 2
        elif (self.tokens[i] == "end") and (indata):
            indata = False
            self.dataBlocks[name] = data
            self.symtbl[name] = "DATA"
            self.AddName(name)
            i += 1
        elif (indata):
            data.append(self.tokens[i])
            i += 1
        else:
            i += 1

#---------------------------------------------------
#  ParseCodeBlocks
#
def ParseCodeBlocks(self):
    """Get a code block."""

    #  Dictionary of all code blocks.  Each entry stores the code values
    #  for that block.
    self.codeBlocks = {}

    incode = False
    instatement = False
    i = 0
    name = ""
    statement = ""
    code = []

    while (i < len(self.tokens)):
        if (self.tokens[i] == "code"):
            if (incode):
                self.Error("Code blocks may not be nested.")
            incode = True
            code = []
```

```
            if (i+1) >= len(self.tokens):
                self.Error("Too few tokens for code block declaration.")
            name = self.tokens[i+1]
            if (self.InvalidName(name)):
                print i
                self.Error("Illegal code block name: " + str(name))
            i += 2
        elif (self.tokens[i] == "end") and (incode):
            incode = False
            self.codeBlocks[name] = code
            self.symtbl[name] = "CODE"
            self.AddName(name)
            i += 1
        elif (incode):
            if (self.tokens[i] == "["):
                    instatement = True
                    statement = "    "
            elif (self.tokens[i] == "]"):
                    code.append(statement)
            else:
                statement = statement + " " + self.tokens[i]
            i += 1
        else:
            i += 1


#-------------------------------------------------
#  ParseFunctions
#
def ParseFunctions(self):
    """Parse all functions."""

    #  Dictionary of all functions.  Each entry stores the tokens associated with
    #  that function, accessible by function name as key.
    self.funcs = {}

    indef = False
    i = 0
    name = ""
    func = []

    while (i < len(self.tokens)):
        if ((self.tokens[i] == "def") or (self.tokens[i] == ":")):
            if (indef):
                self.Error("Function declarations may not be nested.")
            indef = True
            func = []
            if (i+1) >= len(self.tokens):
                self.Error("Too few tokens for function declaration.")
            name = self.tokens[i+1]
            if (self.InvalidName(name)):
                print i
                self.Error("Illegal function name: " + str(name))
            i += 2
        elif ((self.tokens[i] == "end") or (self.tokens[i] == ";")) and (indef):
            indef = False
            self.funcs[name] = func
            self.symtbl[name] = "FUNC"
            self.referenced[name] = False
```

```
            self.AddName(name)
            i += 1
        elif (indef):
            func.append(self.tokens[i])
            i += 1
        else:
            i += 1


#------------------------------------------------
#  TagFunctions
#
def TagFunctions(self):
    """Mark all functions that are referenced."""

    for f in self.funcs:
        for token in self.funcs[f]:
            if (self.symtbl.has_key(token)):
                if (self.symtbl[token] == "FUNC"):
                    self.referenced[token] = True


#------------------------------------------------
#  isStringConstant
#
def isStringConstant(self, token):
    """Return true if this token is a string constant."""

    return (token[0] == token[-1]) and ((token[0] == '"') or (token[0] == "'"))


#------------------------------------------------
#  StringConstantName
#
def StringConstantName(self):
    """Generate a unique name for this string constant."""

    name = "STR_%04X" % self.stringCount
    self.stringCount += 1
    return name


#------------------------------------------------
#  LocateStringConstants
#
def LocateStringConstants(self):
    """Locate string constants inside of functions and replace them with
       references to STR constants."""

    #  Always reference "main"
    self.referenced["main"] = True

    #  Check all tokens of all defined functions
    for key in self.funcs:
        if (self.referenced[key]):
            i = 0
            while (i < len(self.funcs[key])):
                token = self.funcs[key][i]
                if self.isStringConstant(token):       #  if it is a string consta
```

```
                    name = self.StringConstantName()   #  generate a name for it
                    self.str[name] = token             #  and add it to the string
                    self.symtbl[name] = "STR"          #  plus the symbol table
                    self.funcs[key][i] = name          #  and change the token to
                    self.AddName(name)                 #  add the new name
                i += 1


#-------------------------------------------------
#  Dependencies
#
def Dependencies(self, names):
    """Add all dependencies for the given list of dependencies."""

    for d in names:                                    #  for all dependencies
        if (d not in self.dependencies):               #  if not in global list
            self.dependencies.append(d)                #  add it
            self.Dependencies(DEPENDENCIES[d])         #  and all of its dependencies


#-------------------------------------------------
#  LibraryRoutines
#
def LibraryRoutines(self):
    """Locate all library routines used."""

    #  List of all library routines used
    self.lib = []

    #  Check all tokens of all defined functions
    for key in self.funcs:
        for token in self.funcs[key]:
            if LIBRARYMAP.has_key(token):              #  if it is a library routine
                if (token not in self.lib):            #  and hasn't been added yet
                    self.lib.append(token)             #  add it to the list
                    self.symtbl[token] = "LIB"         #  and the symbol table

    #  Now add all the dependencies
    depends = []
    for routine in self.lib:                           #  for every identified librar
        name = LIBRARYMAP[routine]                     #  get the library file name
        for d in DEPENDENCIES[name]:                   #  check its dependencies
            if (d not in depends):                     #  if not already added
                depends.append(d)                      #  add it to the list

    #  Now add the dependencies of the dependencies
    self.Dependencies(depends)


#-------------------------------------------------
#  CompileNumber
#
def CompileNumber(self, token):
    """Compile a number by pushing it on the stack."""

    n = self.Number(token)

    #  Low 16-bits
    self.fasm.append(["","lda","#<$"+hex(n)[2:]])
```

```
    self.fasm.append(["","ldx","#>$"+hex(n)[2:]])
    self.fasm.append(["","jsr","push"])

    #  If marked as double-precision, push the upper 16-bits, too
    if ("#" in token):
        self.fasm.append(["","lda","#"+hex((n >> 16) & 0xFF)])
        self.fasm.append(["","ldx","#"+hex((n >> 24) & 0xFF)])
        self.fasm.append(["","jsr","push"])

    #  Make sure we include push and pop in the dependencies
    if ("push" not in self.dependencies):
        self.dependencies.append("push")
    if ("pop" not in self.dependencies):
        self.dependencies.append("pop")


#-------------------------------------------------
#  CompileVariableRef
#
def CompileVariableRef(self, token):
    """Compile a variable ref by putting its address on the stack."""

    name = self.names[token]
    self.fasm.append(["","lda","#<"+name])
    self.fasm.append(["","ldx","#>"+name])
    self.fasm.append(["","jsr","push"])


#-------------------------------------------------
#  CompileDataBlockRef
#
def CompileDataBlockRef(self, token):
    """Compile a data block ref by putting its address on the stack."""

    name = self.names[token]
    self.fasm.append(["","lda","#<"+name])
    self.fasm.append(["","ldx","#>"+name])
    self.fasm.append(["","jsr","push"])

#-------------------------------------------------
#  CompileCodeBlockRef
#
def CompileCodeBlockRef(self, token):
    """Compile a code block ref by jsr to its address."""

    name = self.names[token]
    self.fasm.append(["","jsr", name])


#-------------------------------------------------
#  CompileStringConstantRef
#
def CompileStringConstantRef(self, token):
    """Compile a reference to a string constant by putting its
       address on the stack."""

    name = self.names[token]
    self.fasm.append(["","lda","#<"+name])
    self.fasm.append(["","ldx","#>"+name])
```

```
    self.fasm.append(["","jsr","push"])


#-------------------------------------------------
#  CompileLoopBegin
#
def CompileLoopBegin(self):
    """Compile the start of a loop."""

    t = self.GetLabel("loop")
    self.fasm.append([t,"",""])
    self.loop.append([t, self.GetLabel("loop2")])


#-------------------------------------------------
#  CompileLoopEnd
#
def CompileLoopEnd(self):
    """Compile the end of a loop."""

    if self.loop == []:
        self.Error("Loop underflow!")
    t = self.loop[-1]
    self.loop = self.loop[0:-1]
    self.fasm.append(["","jmp",t[0]])
    self.fasm.append([t[1],"",""])


#-------------------------------------------------
#  CompileIf
#
def CompileIf(self):
    """Compile an if statement."""

    t_else = self.GetLabel("else")
    t_then = self.GetLabel("then")
    t      = self.GetLabel("t")
    t_end  = self.GetLabel("tend")
    self.compare.append([t_else, t_then, False, t_end])
    self.fasm.append(["","jsr","pop"])
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","beq", t])
    self.fasm.append(["","bne", t_then])
    self.fasm.append([t, "jmp", t_else])
    self.fasm.append([t_then, "", ""])


#-------------------------------------------------
#  CompileNotIf
#
def CompileNotIf(self):
    """Compile a not if statement."""

    t_else = self.GetLabel("else")
    t_then = self.GetLabel("then")
    t      = self.GetLabel("t")
    t_end  = self.GetLabel("tend")
    self.compare.append([t_else, t_then, False, t_end])
    self.fasm.append(["","jsr","pop"])
```

```
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","bne", t])
    self.fasm.append(["","beq", t_then])
    self.fasm.append([t, "jmp", t_else])
    self.fasm.append([t_then, "", ""])


#-------------------------------------------------
#  CompileElse
#
def CompileElse(self):
    """Compile an else statement."""

    self.compare[-1][2] = True
    t_else, t_then, flag, t_end = self.compare[-1]
    self.fasm.append(["","jmp",t_end])
    self.fasm.append([t_else,"",""])


#-------------------------------------------------
#  CompileThen
#
def CompileThen(self):
    """Compile a then statement."""

    if self.compare == []:
        self.Error("Compare underflow!")
    t_else, t_then, flag, t_end = self.compare[-1]
    self.compare = self.compare[0:-1]
    if not flag:
        self.fasm.append([t_else,"",""])
    else:
        self.fasm.append([t_end,"",""])


#-------------------------------------------------
#  CompileBreak
#
def CompileBreak(self):
    """Comple a break statement."""

    t = self.loop[-1]
    self.fasm.append(["","jmp",t[1]])


#-------------------------------------------------
#  CompileCont
#
def CompileCont(self):
    """Compile a continue statement."""

    t = self.loop[-1]
    self.fasam.append(["","jmp",t[0]])


#-------------------------------------------------
#  CompileIfBreak
#
def CompileIfBreak(self):
```

```
    """Compile a conditional break."""

    t = self.loop[-1]
    q = self.GetLabel("break")
    self.fasm.append(["","jsr","pop"])
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","beq", q])
    self.fasm.append(["","jmp", t[1]])
    self.fasm.append([q,"",""])


#-------------------------------------------------
#  CompileIfCont
#
def CompileIfCont(self):
    """Compile a conditional continue."""

    t = self.loop[-1]
    q = self.GetLabel("ifcont")
    self.fasm.append(["","jsr","pop"])
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","beq", q])
    self.fasm.append(["","jmp", t[0]])
    self.fasm.append([q,"",""])


#-------------------------------------------------
#  CompileNotIfBreak
#
def CompileNotIfBreak(self):
    """Compile a negated conditional break."""

    t = self.loop[-1]
    q = self.GetLabel("notifbreak")
    self.fasm.append(["","jsr","pop"])
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","bne", q])
    self.fasm.append(["","jmp", t[1]])
    self.fasm.append([q,"",""])


#-------------------------------------------------
#  CompileNotIfCont
#
def CompileNotIfCont(self):
    """Compile a negated conditional continue."""

    t = self.loop[-1]
    q = self.GetLabel("notifcont")
    self.fasm.append(["","jsr","pop"])
    self.fasm.append(["","cmp","#0"])
    self.fasm.append(["","bne", q])
    self.fasm.append(["","jmp", t[0]])
    self.fasm.append([q,"",""])


#-------------------------------------------------
#  CompileLibraryRef
#
```

```
def CompileLibraryRef(self, token):
    """Compile a reference to a library word."""

    self.fasm.append(["","jsr", LIBRARYMAP[token]])


#-------------------------------------------------
#  CompileReturn
#
def CompileReturn(self):
    """Compile a return statement."""

    #  Return from subroutine
    self.fasm.append(["","rts",""])


#-------------------------------------------------
#  FunctionAddress
#
def FunctionAddress(self):
    """Mark the next function reference to push the address
       on the stack."""

    self.functionAddress = True


#-------------------------------------------------
#  Keywords
#
def Keywords(self):
    """Place all keywords in the symbol table and
       create the compile helper function dictionary."""

    self.symtbl["{"]       = "KWD"
    self.symtbl["}"]       = "KWD"
    self.symtbl["if" ]     = "KWD"
    self.symtbl["0if"]     = "KWD"
    self.symtbl["else"]    = "KWD"
    self.symtbl["then"]    = "KWD"
    self.symtbl["break"]   = "KWD"
    self.symtbl["cont"]    = "KWD"
    self.symtbl["?break"]  = "KWD"
    self.symtbl["?cont"]   = "KWD"
    self.symtbl["?0break"] = "KWD"
    self.symtbl["?0cont"]  = "KWD"
    self.symtbl["&"]       = "KWD"
    self.symtbl["return"]  = "KWD"

    #  Compile helper dictionary
    self.keywords = {
        "{"       : self.CompileLoopBegin,
        "}"       : self.CompileLoopEnd,
        "if"      : self.CompileIf,
        "0if"     : self.CompileNotIf,
        "else"    : self.CompileElse,
        "then"    : self.CompileThen,
        "break"   : self.CompileBreak,
        "cont"    : self.CompileCont,
        "?break"  : self.CompileIfBreak,
```

```
        "?cont"   : self.CompileIfCont,
        "?0break" : self.CompileNotIfBreak,
        "?0cont"  : self.CompileNotIfCont,
        "return"  : self.CompileReturn,
        "&"       : self.FunctionAddress
    }


#-------------------------------------------------
#  CompileDataBlocks
#
def CompileDataBlocks(self):
    """Create assembly instructions for all data blocks."""

    #  Holds the assembly code for the data blocks
    self.dasm = []

    #  Compile each block
    for f in self.dataBlocks:
        self.dasm.append([self.names[f],"",""])   #  Data block label
        for number in self.dataBlocks[f]:
            n = self.Number(number)
            if ("#" in number):
                if (n < -32768):
                    n += 2**32
                self.dasm.append(["",".byte",str(n & 0xff)])
                self.dasm.append(["",".byte",str((n >> 8) & 0xff)])
                self.dasm.append(["",".byte",str((n >> 16) & 0xff)])
                self.dasm.append(["",".byte",str((n >> 24) & 0xff)])
            elif ("%" in number):
                if (n < 0):
                    n += 2**16
                self.dasm.append(["",".byte",str(n & 0xff)])
                self.dasm.append(["",".byte",str((n >> 8) & 0xff)])
            else:
                if (n >= -32768) and (n < 0):
                    n += 2**16
                if (n < -32768):
                    n += 2**32
                if (n >= 0) and (n < 256):
                    self.dasm.append(["",".byte",str(n)])
                elif (n >= 0) and (n <= 65535):
                    self.dasm.append(["",".byte",str(n & 0xff)])
                    self.dasm.append(["",".byte",str((n >> 8) & 0xff)])
                elif (n > 65535):
                    self.dasm.append(["",".byte",str(n & 0xff)])
                    self.dasm.append(["",".byte",str((n >> 8) & 0xff)])
                    self.dasm.append(["",".byte",str((n >> 16) & 0xff)])
                    self.dasm.append(["",".byte",str((n >> 24) & 0xff)])

#-------------------------------------------------
#  CompileCodeBlocks
#
def CompileCodeBlocks(self):
    """Create assembly instructions for all code blocks."""

    #  Holds the assembly code for the data blocks
    self.casm = []
```

```
    #  Compile each block
    for f in self.codeBlocks:
        self.casm.append([self.names[f],"",""])   #  Code block label
        for line in self.codeBlocks[f]:
            self.casm.append(["",line,""])


#-------------------------------------------------
#  CompileFunctions
#
def CompileFunctions(self):
    """Compile all defined functions."""

    #  Holds the assembly code for the functions
    self.fasm = []

    #  Compile each function
    for f in self.funcs:
        if (self.referenced[f]):
            self.loop = []
            self.compare = []
            self.funcName = f  #  Currently compiling
            self.fasm.append([self.names[f],"",""])  #  Subroutine label
            for token in self.funcs[f]:
                self.Token = token  #  Current token
                if (self.symtbl.has_key(token)):
                    if (self.symtbl[token] == "FUNC"):
                        if (self.functionAddress):
                            #  Push the function address
                            self.fasm.append(["","lda", "#<"+self.names[token]])
                            self.fasm.append(["","ldx", "#>"+self.names[token]])
                            self.fasm.append(["","jsr", "push"])
                            self.functionAddress = False
                        else:
                            #  Compile a subroutine call
                            self.fasm.append(["","jsr",self.names[token]])
                    elif (self.symtbl[token] == "VAR"):
                        self.CompileVariableRef(token)          #  Compile a varia
                    elif (self.symtbl[token] == "DATA"):
                        self.CompileDataBlockRef(token)         #  Compile a refer
                    elif (self.symtbl[token] == "CODE"):
                        self.CompileCodeBlockRef(token)
                    elif (self.symtbl[token] == "CONST"):
                        self.CompileNumber(str(self.consts[token]))  #  Compile a
                    elif (self.symtbl[token] == "STR"):
                        self.CompileStringConstantRef(token)    #  Compile a refer
                    elif (self.symtbl[token] == "KWD"):
                        self.keywords[token]()                  #  Compile a keywo
                    elif (self.symtbl[token] == "LIB"):
                        if (self.functionAddress):
                            #  Push the address of the library routine
                            self.fasm.append(["","lda", "#<"+token])
                            self.fasm.append(["","ldx", "#>"+token])
                            self.fasm.append(["","jsr", "push"])
                            self.functionAddress = False
                        else:
                            #  Compile a library word call
                            self.CompileLibraryRef(token)           #  Compile a l
                    else:
```

```
                        self.Error("Unknown symbol table type: " + str(token) + ",
                                   str(self.symtbl[token]) + ", function = " + f)
                elif (self.isNumber(token)):
                    self.CompileNumber(token)
                else:
                    self.Error("Unknown token: " + str(token) + ", function = " +
            self.fasm.append(["","rts",""])  #  Ending RTS instruction


#-------------------------------------------------
#  pp
#
def pp(self, f, t):
    """Write an instruction to the assembly output file."""

    f.write(t[0])
    f.write("\t")
    f.write(t[1])
    f.write(" ")
    f.write(t[2])
    f.write("\n")


#-------------------------------------------------
#  AssemblyOutput
#
def AssemblyOutput(self):
    """Generate the output assembly code."""

    #  Check for a main function
    if not self.symtbl.has_key('main'):
        self.Error("No main function defined.")
    if (self.symtbl["main"] != "FUNC"):
        self.Error("No main function defined.")

    #  Open the output assembly source code file
    f = open(self.outname + ".s", "w")

    #  Write the header
    for s in HEADER:
        f.write("; "+s+"\n")

    #  Origin
    if (self.sys):
        self.org = 0x2000
    self.pp(f, ["","*=","$%04X" % self.org])
    f.write("\n")

    #  Library equates
    for s in EQUATES:
        self.pp(f, [s,"=","$%04X" % EQUATES[s]])
    f.write("\n")

    #  Variables
    offset = self.VARTOP
    for v in self.vars:
        offset -= self.vars[v]
        self.pp(f, [self.names[v], "=", "$%04X" % offset])
    f.write("\n")
```

```
    #  Main call
    self.pp(f, ["", "lda", "#0"])
    self.pp(f, ["", "sta", "sp"])  #  zero stack pointer
    self.pp(f, ["", "jmp", self.names["main"]])
    f.write("\n")

    #  Data blocks
    for s in self.dasm:
        self.pp(f, s)
    f.write("\n")

    #  Code blocks
    for s in self.casm:
        self.pp(f, s)
    f.write("\n")

    #  String constants
    for s in self.str:
        self.pp(f, [self.names[s], ".byte", str(len(self.str[s])-2) +","+ self.str
    f.write("\n")

    #  Dependencies
    for s in self.dependencies:
        g = open(LIB_DIR+s+".s", "r")
        f.write(g.read())
        g.close()
    f.write("\n")

    #  Library routines
    for s in self.lib:
        g = open(LIB_DIR+LIBRARYMAP[s]+".s", "r")
        f.write(g.read())
        g.close()
    f.write("\n")

    #  Functions
    for s in self.fasm:
        self.pp(f, s)
    f.write("\n")

    # end label
    f.write("_end\n")
    f.close()


#-------------------------------------------------
#  ConvertToAppleII
#
def ConvertToAppleII(self):
    """Convert the binary output file to an Apple II
       text file of EXEC-ing into memory."""

    #  Get the binary data
    f = open(self.outname + ".bin", "rb")
    d = f.read()
    f.close()

    #  Write to a new output file
```

```
    f = open(self.outname + ".txt", "w")
    f.write("CALL -151")

    #  Write each byte
    i = 0
    while (i < len(d)):
        if ((i % 8) == 0):
            f.write("\n%04X: " % (self.org + i))
        f.write("%02X " % ord(d[i]))
        i += 1

    #  Close the file
    if (self.sys):
        f.write("\nBSAVE %s, A%d, L%d, TSYS\n" % (self.outname.upper(), self.org,
    else:
        f.write("\nBSAVE %s, A%d, L%d\n" % (self.outname.upper(), self.org, len(d)


#---------------------------------------------------
#  BlockToTrackSector
#
def BlockToTrackSector(self, blk):
    """Convert a block number to the corresponding track and
       sector numbers."""

    trk = int(blk/8)
    sec = ([[0,14],[13,12],[11,10],[9,8],[7,6],[5,4],[3,2],[1,15]])[blk % 8]
    return [trk, sec]


#---------------------------------------------------
#  TrackSectorOffset
#
def TrackSectorOffset(self, trk, sec):
    """Convert a given track and sector to an offset in
       the disk image."""

    return 256*(16*trk + sec)


#---------------------------------------------------
#  ReadBlock
#
def ReadBlock(self, dsk, blk):
    """Return a 512 byte block from dsk at blk."""

    #  Get the track and sectors for this block
    trk, sec = self.BlockToTrackSector(blk)

    #  Get the offsets
    off1 = self.TrackSectorOffset(trk, sec[0])
    off2 = self.TrackSectorOffset(trk, sec[1])

    #  Get the 256 bytes at each of these locations
    #  as a single list and return it
    return dsk[off1:(off1+256)] + dsk[off2:(off2+256)]


#---------------------------------------------------
```

```
#  WriteBlock
#
def WriteBlock(self, dsk, blk, data):
    """Write 512 bytes of data to dsk at block blk."""

    #  Data must be exactly 512 long
    if (len(data) != 512):
        self.Error("Illegal block length.")

    #  Get the track and sectors for this block
    trk, sec = self.BlockToTrackSector(blk)

    #  Get the points into the dsk image
    off1 = self.TrackSectorOffset(trk, sec[0])
    off2 = self.TrackSectorOffset(trk, sec[1])

    #  Update the 256 values at off1 with the first
    #  256 values of data and the values at off2 with
    #  the next 256 values of data
    for i in xrange(256):
        dsk[off1+i] = data[i]
        dsk[off2+i] = data[i+256]


#------------------------------------------------
#  Date
#
def Date(self):
    """Return the date in ProDOS format."""

    t = time.localtime(time.time())
    dlow = ((t[1] & 0x07) << 5) | t[2]
    dhigh = ((t[0] - 2000) << 1) | ((t[1] & 0x08) >> 3)
    return [dlow, dhigh]


#------------------------------------------------
#  Time
#
def Time(self):
    """Return the time in ProDOS format."""

    t = time.localtime(time.time())
    return [t[4], t[3]]


#------------------------------------------------
#  MarkBlock
#
def MarkBlock(self, blk, blocknum):
    """Mark blocknum as being used."""

    #  Byte number containing blocknum
    bytenum = blocknum / 8

    #  Bit number containing blocknum
    bitnum = 7 - (blocknum % 8)

    #  Set to used
```

```
    blk[bytenum] = blk[bytenum] & ~(1 << bitnum)


#------------------------------------------------
#  ConvertToDSK
#
def ConvertToDSK(self):
    """Convert the binary output file to a .dsk file."""

    #  Get the binary data
    f = open(self.outname + ".bin", "rb")
    d = f.read()
    f.close()
    data = []
    for c in d:
        data.append(ord(c))

    #  Read the blank disk image file as list of bytes.
    #  It is assumed that this file is a blank ProDOS 140k
    #  floppy image stored in DOS 3.3 order.
    f = open(DISK_IMAGE, "rb")
    t = f.read()
    f.close()
    dsk = []
    for c in t:
        dsk.append(ord(c))

    #
    #  Create the directory entry for this file
    #

    #  File storage type and name length
    blk = self.ReadBlock(dsk, 2)
    offset = 0x2b  #  Offset to start of second directory entry
    if (len(data) <= 512):
        blk[offset] = 0x15  #  seedling file
        seedling = True
    else:
        blk[offset] = 0x25  #  sapling file
        seedling = False

    #  File name, always "A.OUT"
    blk[offset+1] = ord("A")
    blk[offset+2] = ord(".")
    blk[offset+3] = ord("O")
    blk[offset+4] = ord("U")
    blk[offset+5] = ord("T")

    #  File type
    if (self.sys):
        blk[offset+0x10] = 255
    else:
        blk[offset+0x10] = 6

    #  Key pointer, block 7
    blk[offset+0x11] = 7
    blk[offset+0x12] = 0

    #  File size
```

```
blocks = int(math.ceil(len(data)/512.0))
if (blocks == 0):
    blocks = 1
blk[offset+0x13] = blocks % 256
blk[offset+0x14] = blocks / 256
blk[offset+0x15] = len(data) % 256
blk[offset+0x16] = len(data) / 256
blk[offset+0x17] = 0

#  Date and time
dlow, dhigh = self.Date()
blk[offset+0x18] = dlow
blk[offset+0x19] = dhigh
m, h = self.Time()
blk[offset+0x1a] = m
blk[offset+0x1b] = h

#  ProDOS versions
blk[offset+0x1c] = 0
blk[offset+0x1d] = 0

#  Access code
blk[offset+0x1e] = 0xc3

#  Aux type code (program start in memory)
blk[offset+0x1f] = self.org % 256
blk[offset+0x20] = self.org / 256

#  Last accessed time
blk[offset+0x21] = dlow
blk[offset+0x22] = dhigh
blk[offset+0x23] = m
blk[offset+0x24] = h

#  Key block for directory
blk[offset+0x25] = 2
blk[offset+0x26] = 0

#  Number of files in this directory
blk[0x25] = blk[0x25] + 1

#  Write the block
self.WriteBlock(dsk, 2, blk)

#
#  Index block
#
blk = self.ReadBlock(dsk, 7)

if (seedling):
    #  Write the data
    i = 0
    while (i < len(data)):
        blk[i] = data[i]
        i += 1
    #  Block used
    used = [7]
else:
    #  Reserve the necessary number of blocks
```

```
        used = []
        for i in xrange(blocks):
            used.append(i+8)
        j = 0
        for i in used:
            blk[j] = i % 256
            blk[j+256] = i / 256
            j += 1

    #  Write the index block
    self.WriteBlock(dsk, 7, blk)

    #  Write the file data if not a seedling file
    if (not seedling):
        #  Pad data to a multiple of 512
        data = data + [0]*(512 - (len(data) % 512))
        j = 0
        for b in used:
            blk = self.ReadBlock(dsk, b)
            for i in xrange(512):
                blk[i] = data[j+i]
            self.WriteBlock(dsk, b, blk)
            j += 512

        #  Include block 7, the index block, so it will
        #  be marked as used in the volume bitmap
        used.append(7)

    #
    #  Volume bitmap updates
    #
    blk = self.ReadBlock(dsk, 6)
    for i in used:
        self.MarkBlock(blk, i)
    self.WriteBlock(dsk, 6, blk)

    #
    #  Write the entire image to disk
    #
    f = open(self.outname + ".dsk", "wb")
    for i in dsk:
        f.write(chr(i))
    f.close()


#------------------------------------------------
#  ConvertToR65
#
def ConvertToR65(self):
    """Make an r65 file."""

    #  Get the binary data
    f = open(self.outname + ".bin", "rb")
    d = f.read()
    f.close()
    data = []
    for c in d:
        data.append(ord(c))
```

```
    #  Create the output file
    f = open(self.outname + ".r65", "wb")
    f.write(chr(self.org % 256))
    f.write(chr(self.org / 256))
    for c in data:
        f.write(chr(c))
    f.close()

#-------------------------------------------------
#  ConvertToAtariCom
#
def ConvertToCom(self):
    """Make an Atari COMpound file."""

    #  Get the binary data
    f = open(self.outname + ".bin", "rb")
    d = f.read()
    f.close()
    data = []
    for c in d:
        data.append(ord(c))

    #  Create the output file
    f = open(self.outname + ".com", "wb")
    f.write(chr(0xff)) # compound marker $FFFF
    f.write(chr(0xff))
    f.write(chr(self.org % 256))  # startaddress
    f.write(chr(self.org / 256))
    f.write(chr((self.org + len(data)+1)% 256))  # endaddress
    f.write(chr((self.org + len(data)+1)/ 256))

    for c in data:
        f.write(chr(c))

    f.write(chr(0xff))
    f.write(chr(0xff))
    f.write(chr(0xe0)) # RUNAD $02E0
    f.write(chr(0x02))
    f.write(chr(0xe1))
    f.write(chr(0x02))
    f.write(chr(self.org % 256)) # startaddress
    f.write(chr(self.org / 256))

    f.close()


#-------------------------------------------------
#  GenerateFinalOutput
#
def GenerateFinalOutput(self):
    """Create the final output file."""

    #  Assemble the .s file
    if (self.outtype != "asm"):
        rc = os.system(ASSEMBLER + " -r " + self.outname + ".s")
        if (rc != 0):
            self.Error("Error during assembly!")

    #  Convert to desired output file format
```

```
    if (self.outtype == "bin"):
        return
    elif (self.outtype == "a2t"):
        self.ConvertToAppleII()
    elif (self.outtype == "dsk"):
        self.ConvertToDSK()
    elif (self.outtype == "r65"):
        self.ConvertToR65()
    elif (self.outtype == "atari"):
        self.ConvertToCom()


#-------------------------------------------------
#  Compile
#
def Compile(self):
    """Compile the files on the command line."""

    self.funcName = "$MAIN$"      #  Name of currently compiling function
    self.Token = "<na>"           #  Currently compiling token
    self.VARTOP = VARTOP          #  Variable starting address
    self.cmdOrigin = -1           #  -org value, if any
    self.counter = 0              #  Start labels from zero
    self.stringCount = 0          #  String constant name counter
    self.symtbl = {}              #  Ready the symbol table
    self.names = {}               #  Map names to labels
    self.dependencies = []        #  Library word dependencies
    self.functionAddress = False  #  If true, push a function address
    self.referenced = {}          #  True if that function referenced by another
                                  #  only functions actually referenced are compil

    self.ParseCommandLine()       #  Parse the command line arguments
    self.Keywords()               #  Put all keywords into the symbol table
    self.LoadFiles()              #  Load all source code into self.src
    self.Tokenize()               #  Break up into tokens in self.tokens
    self.SetOrigin()              #  Determine the origin for the output code
    self.Declarations()           #  Find all variables and constants
    self.ParseDataBlocks()        #  Get all data blocks and their data
    self.ParseCodeBlocks()        #  Get all code blocks
    self.ParseFunctions()         #  Get all the functions and their tokens
    self.TagFunctions()           #  Tag used functions
    self.LocateStringConstants()  #  Locate string constants in functions
    self.LibraryRoutines()        #  Determine all library routines used
    self.CompileDataBlocks()      #  Generate assembly code for all data blocks
    self.CompileCodeBlocks()      #  Generate assembly code for all code blocks
    self.CompileFunctions()       #  Generate assembly code for all functions
    self.AssemblyOutput()         #  Output all assembly code and assemble it
    self.GenerateFinalOutput()    #  Convert the assembler output into the final d


#-------------------------------------------------
#  __init__
#
def __init__(self):
    """Build the compiler object."""

    #  Help message
    if (len(sys.argv) == 1):
        print "\nSPL Compiler (" + LAST_MOD + ")\n"
```

```
            print "Use: spl file1 [file2 ...] [-o outbase] [-t type] [-sys] [-org n] [
                  "[-stack p] [-prodos]\n"
            print "Where:\n"
            print "    file1     =  first source code file (.spl extension assumed)"
            print "    file2...  =  additional source code files"
            print "    outbase   =  base file name for output files (NO extension)"
            print "    type      =  output file type:"
            print "                     bin = compiled binary"
            print "                     asm = assembly source only"
            print "                     a2t = text file for EXEC on Apple II"
            print "                     dsk = a disk file for Apple II emulators"
            print "                     r65 = source file for the r65 65C02 simulator"
            print "                   atari = compound file for Atari XL/XE DOS"
            print "    sys       =  create a ProDOS system file (-t a2t and -t dsk onl
            print "    org       =  set the origin address to n"
            print "    var       =  set the VARTOP value (variables grow down from her
            print "    stack     =  set the stack address (up to 256 bytes)"
            print "    prodos    =  set VARTOP to allow room for 3 file buffers (Apple
            sys.exit(1)

        #  Otherwise, compile the files
        self.Compile()


#  Run if not imported
if __name__ == '__main__':
    app = SPLCompiler()

#
#  end spl.py
#
```