
A SMALL C COMPILER

FOR THE 8080's

BY RON CAIN

811 Eleventh Ave.
Redwood City, CA 94063

I had to have a compiler for my home computer.

There was no doubt about it: after programming nearly every day in BLISS and then reading about C, PASCAL, LISP, and all the other various languages becoming available, it made no sense at all to continue programming in assembly language.

However, the question arose: Which language?

Surprisingly, the decision was not difficult. Since I do mostly system type programming (editors, music-board drivers, modem talkers, faster versions of LIFE, etc.) rather than application programming (accounting programs, solar eclipse predictors, etc.), the choice boiled down to two: C or BLISS. Besides having the most esoteric names of the group, both possess the strengths of system programming languages. Which is to say they both have a slightly better notion of computer architectures than strictly application oriented languages in terms of accessing bytes and words, and neither wastes time doing "garbage collection" or keeping track of legal array subscripting. Both are intended to produce fairly fast (usually inline) code to get the job done with the least amount of overhead.

After looking around the marketplace, I decided C was the appropriate choice if I expected to produce code usable by others. However, another thing also became obvious: the C compilers available either cost a lot of money, ran only on CP/M, or both. Having neither a lot of money nor CP/M (the second by choice), I had to think of an alternate approach.

The first step came by way of the Tiny-C Interpreter offered by Tiny-C Associates. For a mere \$40, I was able to buy the source code of a working C interpreter. I intended to use it to run the C programs I would write until something bigger and better came along. I sent off the money, got the huge white notebook in the mail, and after a couple weeks of typing, had it up and running. Let me be another of many praising the product. The documentation, clarity of print, and ease of implementation of the thing were all exceedingly good.

It gave me a few weeks to evaluate the C language as well as the interpreter. It became obvious after very little time that the language was very easy to use and to follow. It also became obvious that the interpreter, though an excellent implementation, was too slow for my needs. Modem to disk programs like to think in terms of microseconds, not milliseconds, and the former was not to be had with an interpreter. Clearly, a third approach was needed.

The solution was obvious. I had to write the compiler myself. I said obvious, not easy.

So, after gathering together a thick stack of paper and getting ready to produce the (hopefully) last vast 8080 programming effort, I wrote a few lines, and, . . . reconsidered.

Hmmm. What better use of a systems programming language could there be than to write a compiler? And what better language could be used than the one the compiler would ultimately support? Ideas began to gel, and pieces began to fit together. I already had the interpreter. Even though it was slow, it might be just the ticket to bootstrap my way onto the machine. And so it began.

Considering the fact I had never even seen a compiler before and had not done any lengthy programs in C, the thing went together in a remarkably short period of time. Chalk that up to the ease of use of the language. The ability to have variables local to individual routines made the recursive-descent expression parser fit together literally in hours. A few weeks of a couple hours a night saw the compiler begin to take shape. A few obstacles appeared along the way; probably the worst being the point where I discovered I didn't have enough memory for the entire compiler to reside in memory along with the interpreter. One demonic session with the editor deleting all comments (gulp), and shortening all names, brought it to size and solved that problem. The compiler's still recovering from that early trauma. Eventually, there came an evening when all such little problems were solved, and the compiler, running under the interpreter, accepted a small C program and produced an 8080 program.

At that point, I refined the compiler, removed subtle errors, and discovered I still didn't have enough memory for the compiler, the interpreter, and the symbol table a program as large as the compiler itself required. So I asked around, eventually got access to a UNIX system, and spent a couple more weeks refining the compiler without regard to memory or interpreters. This exercise helped remove some of the less obvious syntax errors not caught by the interpreter and necessitated modifying the compiler to accept the true C syntax rather than the slightly-modified syntax used by the interpreter. After this upgrade, the compiler was beginning to look like a true C compiler, and there came a time it ran under UNIX and accepted as input itself.

So, finally, I was able to take the equivalent 8080 code for the compiler back to my machine at home to complete it. The generated code was assembled, run, and once again the com-

piler was submitted to itself. The results of this inbreeding highlighted errors in the generated 8080 code which had thus far been hidden. I set those aright, compiled the compiler once again, then used the output to compile the compiler again, and then one more time (sound like an infinite loop?) compiled the compiler. When the code generated by this great-grandson was byte-for-byte identical to the code generated by UNIX way back when, and all subsequent offspring produced such twins, I decided the thing could be considered working.

Once it was working, I decided it would save people in a similar situation a lot of parallel effort if I made the thing available. And if I sold it, I would probably get caught up in supporting it on a number of machines, which was not my intention behind writing it. Giving it away for free seemed like an excellent way to make C available to hobbyists who might otherwise not have been able to get it. What better way to add users to the C community? So here it is.

For those of you who are interested in getting a compiler for a structured language on your home machines, I'd feel flattered if you'd consider the one published here. And if you are the sort who is looking for a systems programming language, you are undoubtedly the sort who would modify the thing to fit your own needs. Therefore, a few vital statistics are in order.

In a nutshell, the compiler:

1. Is written in C.
2. Accepts as input a text file written in C.
3. Produces as output a text file of 8080 mnemonics.

The syntax it accepts is a subset of the standard C language. Within this subset, it does not depart from the standard syntax, which means the listing shown here will compile and run under UNIX. Although the subset is limited to make the compiler simpler and will therefore not accept just any pre-existing C program, the programs you can write with it are authentic and will be compatible with more complete compilers.

Right out front, so that UNIX freaks can groan and keep their hopes from getting too high, let me tell you the features of standard C this compiler does NOT support:

1. Floating-point data types.
2. Structures (or unions).
3. Multiple dimension arrays.

The aim was not to support the full C language, but rather to support enough of a subset to be able to create C programs which would be compatible with standard C. Then, as the compiler expanded, more and more features could be added to bring it closer to the real thing.

Currently, the allowable data types are:

1. char—8 bit data element.
2. int—16 bit data element.

Obviously, this means the compiler is an integer-only subset of the language. Which means it will not handle floating-point numbers. Actually, all internal arithmetic operations assume 16-bit integers, meaning 8-bit character elements are sign-extended prior to use.

Allowable modifiers of the two basic types are:

1. type *name—declares name to be a pointer to an element of the specified type.
2. type name []—syntactically identical to the above pointer declaration.
3. type name[constant]—declares an array of “constant” size where each array element is of the specified type.

If you've worked with C before, you know just about everything is done with pointers. It lets you exploit the architecture of the cpu by giving access to all addressable memory. Unlike standard C, you can't use more than one modifier per declaration, meaning it will not accept something like “int (*name) []”. This does not present a terrible restriction, but it must be mentioned. Since no runtime checking is made on the legality of pointer usage, it is a trivial matter to access any byte in memory, and a not-at-all-trivial problem finding which routine is clobbering some random location. You can see why pointers have been lumped with “goto's” as the bane of the code maintainers and modifiers.

Allowable primary expressions are:

1. **A local or global variable name.** Any name which has been declared previously may be used as an expression. The name can be any length, but only the first 8 characters are used by the compiler. Presently, variable names are not folded to upper-case prior to being used in the output file. Therefore, the variable “x” would expand in the expression “x = 0” to:

```
LXI H,0
SHLD x
```

If the assembler you intend to use cannot handle lower-case names, do not use lower-case names in your C programs.

2. **Constants.** These can be either:
 - a. A decimal number.
 - b. A single or pair of ASCII characters enclosed in single quotes, such as 'a' or 'TX'.
 - c. A string enclosed in double quotes, such as “this is a string”. The value such a constant yields is a pointer to the first character of the string which the compiler stores in memory.
3. **Function calls.** These are defined as any expression followed by an open paren. Thus, a function can be to a named routine, such as “print()”, or to the results of some expression, such as “1000()” (which calls location 1000 decimal), or “array[i] ()” which calls the location whose value is found in array [i]. Functions always return a value in HL, though it does not have to be used. Arguments to functions appear between the parentheses and are pushed onto the stack in the order they appear. Hence, “print(x,y,z);” would effectively do:


```
PUSH x
PUSH y
PUSH z
CALL print
```

 Any number of arguments may be passed.
4. **Subscripted elements.** Either an array name or a pointer may be subscripted to refer to the appropriate element. Subscripts are assumed to start from zero. Therefore, legal

expressions are:

- array [0] – the first element in array.
- array [x + 31] – the element at the address given by adding x to 31 and then to array.
- pointer [i] – the element at the address given by adding i to the contents of pointer.

Only single dimensions are allowed. Subscripting either an integer array or a pointer to an integer will cause the subscript expression to be doubled. Therefore, if you declare "int *ptr", the expression "ptr [3]" refers to the element at ptr+6.

Allowable unary expression operators are:

1. "-" – forms the two's complement of the expression (minus).
2. "*" – refers to the element pointed to by the expression (providing the expression is a pointer).
3. "&" – evaluates the address of the given expression, providing it has one. Hence, &count yields the address of the element "count". &1000 is an error.
4. "++" – increments the expression by one. If this appears before the expression, it increments before using it. If it appears after it, it will increment it after. Only lvalues (expressions which can appear on the left-hand side of an equal sign) are allowed. Hence, assuming "count" contains a 5, ++count would evaluate to a 6, and "count" would contain a 6. Count++ would evaluate to a 5, and count would contain a 6. 1000++ is illegal. If this operator is applied to an integer pointer, it will increment by 2.
5. "--" – decrements the expression by one. This works just like ++ but subtracts one rather than adding.

Allowable binary operators are:

1. "+" – adds the two expressions (i.e. count + total)
2. "-" – subtracts the two expressions.
3. "*" – multiplies the two expressions.
4. "/" – divides the first expression by the second.
5. "%" – yields the remainder after dividing the first expression by the second (modulo).
6. "|" – yields the logical inclusive "or" of the two expressions.
7. "^" – yields the logical exclusive "or".
8. "&" – yields the logical "and".
9. "=" – assigns the value of the expression on the right to the one on the left. Since evaluation is done right to left in this case, syntaxes like:
x = y = z = 0;
are legal.

Comparison operators compare two expressions and yield either a zero or a one depending whether the result of the compare is false or true, respectively. These include:

1. "==" – tests for equality.
2. "!=" – tests for inequality.
3. "<" – tests for less than.
4. ">" – tests for greater than.
5. "<=" – tests for less than or equal to.

Comparisons involving a pointer (which is an address) are done as unsigned compares. All other compares are signed.

Program control is accomplished by the following allowable statements:

1. **expression**; An expression, no matter how complex, is considered a simple statement.
2. **if (expression) statement**; If the expression is nonzero, the statement is executed, otherwise it isn't.
3. **if (expression) statement; else statement**; This form of the "if" statement allows the "else" clause. As is the case with most "dangling else" ambiguities, all "else's" pair with the nearest unmatched "if".
4. **while (expression) statement**; The statement is performed until the expression becomes zero. Since the test is made before the statement is executed the first time, it need not be executed at all.
5. **break**; This statement will cause control to be transferred out of the inner-most "while" loop.
6. **continue**; This statement, used within a "while" loop, will transfer control back to the top of the loop.
7. **return**; This statement does an immediate return from the current function. If a function does not end with this statement, one is performed regardless.
8. **return expression**; This statement allows a function to return a value explicitly.
9. ; A semicolon by itself is considered a null statement which does nothing but take the place of a statement. You see this in forms such as: "while (*iptr++ = *jptr++);" where the test itself contains all the necessary parts of the statement.
10. { **statement; statement; . . . ; statement**; } The use of curly brackets (" { } ") around any group of simple statements is considered a compound statement. A compound statement can be used anywhere a simple statement can. For example:

```
while (1) { x = 3; y = 10; funct(33); }
or
if (x < y)
{ print(x);
  total(x);
  --x;
}
else
{ type("all done");
  x = y;
}
```

The compiler also accepts some pseudo-op type statements. These are:

1. **#include filename** Anywhere this statement appears in the program, the indicated filename will be opened and inserted. The "included" file may not contain an "#include" statement.
2. **#define name string** This statement will cause the given name to be replaced by the string throughout the entire program. Normally, it is used to define constants, such as:
#define tablesize 1000
#define maxlength 8
But it can also be used for any sort of text:
#define jprint 3crs print (12); print(12); print(12);

The replacement is purely on a text level, and error checking will be performed only after the replacement.

3. `#asm ... #endasm` This structure is not supported by standard C, but it was a feature I felt I needed. It may appear anywhere a statement would, but it passes everything between the word “`#asm`” and the word “`#endasm`” right through the parser without intervention. It is intended to be used to pass assembly language code through the parsing mechanism. Since it counts as a single statement, allowable (and expected) forms are:

```
if (x < y)
  #asm
  LHL TOTAL
  CALL ADD
  CNC ERROR
  #endasm
else return;
```

This pseudo-op conceivably allows an entire assembly language program to be passed through the compiler. Its intent is to allow machine dependent features (like the 8080's “IN” and “OUT” instructions to be used without writing separate programs).

In parsing the input, the compiler never tries to be overly clever, which simplifies the compiler quite a bit and makes the generated code quite predictable, if somewhat redundant.

The input file is scanned character-by-character left to right. Only a single 80 character line at a time is buffered. This reduces the amount of memory used by the compiler and makes any kind of look-ahead improbable. The expression parser (namely the routines `heir1()` through `heir11()` and `primary()`) uses the method of recursive-descent to evaluate an expression, which means the routines call themselves when necessary (such as encountering an open paren). All expressions result in a value in the HL register pair. Character values are sign-extended through the H register. This means the line of code:

```
x = 5;
generates the code:
LXI H, 5
SHLD x
```

Notice, HL still contains the 5. The compiler relies upon this convention in the forms:

```
x = y = z = 0;
```

which generates the code:

```
LXI H,0
SHLD x
SHLD y
SHLD z
```

In fact, using an expression as a statement is a good way to load the HL pair using the compiler. The expression:

```
100;
```

puts 100 into the HL pair. Of more use might be the expression:

```
&local;
```

which would load the address of the variable “local” into HL.

Whenever a binary operator is seen, and a second value is needed, the HL pair is pushed onto the stack, and the second operand is loaded into HL. Therefore, the expression:

```
x = x + 5;
```

would generate the code:

```
LHLD x
PUSH H
LXI H,5
POP D
DAD D
SHLD x
```

Obviously, this shows the excessive caution used by the compiler. It need not actually have pushed the HL pair, but could merely have put it into DE with an “XCHG” instruction. Except for the one fact the compiler had no idea how involved the second expression would be. Consider:

```
x = x + (5 * x) / funct(x * 3);
```

Obviously, in such a case, pushing H would be wise. The compiler makes no distinctions, but generates bullet-proof, rather than optimal, code.

Global variables (those declared outside of a function) are assigned storage and can be referenced by name. This means assembly language code can refer to the global variables using the same name as the C program. Local variables, however, like function arguments, occupy a spot on the stack, and are referenced by some offset from the current stack pointer. The simple routine:

```
main(arg)
char arg;
{ int localword;
  localword = arg;
}
```

would generate the code:

```
main:
```

```
PUSH B
LXI H,0
DAD SP
PUSH H
LXI H,4
DAD SP
CALL ccgchar
POP D
CALL ccpint
POP B
RET
```

Notice a few important items. First, the address of “localword” is evaluated as it is encountered. Then it is pushed, since HL must be used for something else. The address of “arg” is evaluated, and an outside routine “ccgchar” is called to read one byte from the specified address and to sign-extend it. Then the address of “localword” is popped, and another routine “ccpint” is called to store a 16-bit value in HL at the address in DE. This means there is a runtime library needed for the C programs produced by this compiler. It is not listed here, since it is as long as the compiler itself (it is in 8080 code) and all the routines seemed fairly obvious, since they are all invoked by the code generating routines at the end of the compiler. If there is enough response, perhaps Dr. Dobbs can print that too one day.

I guess the biggest question remaining is “How can I get it on my system?”

Good question, and I'm afraid you know the answer better than I do.

Obviously, the C listing printed here is not directly usable on your home system. If you had access to a UNIX system,

you could simply type it in as shown, compile it with UNIX's C compiler, and have a running version. At that point, you could run the compiler under UNIX, submit it to itself for compilation, and the output would be a copy of the compiler in 8080 mnemonics. That output is directly usable on any 8080 machine with some mass-storage and an assembler. This is just what I did in bootstrapping the compiler, so naturally I have all the intermediate files on floppy disk.

However, I am not a distributor. I can't handle, nor do I want to handle, requests to put the files onto some other media. I originally released the first version of the compiler some months ago and have not been able to deal with even the moderate number of requests.

It is my hope some of you will either go to the effort yourselves or get some distributor to go to the trouble of getting the files onto the necessary floppies or cassettes. Such group efforts I will assist where possible.

Already, a couple of distributors or private parties have the thing working on other machines or are planning it.

Walt Bilofsky, Software Consultant, 14478 Glorietta Drive, Sherman Oaks, Ca. 91423, has been in on this thing from early on, and has an extremely improved version running on his Heath-kit. His compiler supports much more of the standard C language, and his runtime library sports the ability to do I/O redirection like the UNIX shell. I am told he is considering a CP/M version, and will eventually support others. What is published here is free, if difficult to transport directly to your system. But for a small fee, I'm sure he'll be able to supply you with a working compiler.

Also, The Code Works has obtained a version and was considering making it work on other machines. Their address is Box 550, Goleta, CA 93017.

Which brings up interesting points. If you haven't already made the intuitive leap about the power behind writing the compiler in the same language it supports, it lies in the ability to compile itself. This means a user with extra memory can add additional features to the basic compiler, compile it with the old compiler, and voila! A new and more powerful version will exist.

If you have a working compiler on one cpu and want to bootstrap your way to another kind of processor, you need only change the code generating portions of the compiler (all grouped into the final section of the listing) to make code for the new machine, compile that compiler, and once again, you have a new beastly. Sort of like cloning.

Personally, I've developed the thing about as far as I need to begin using it for the things I originally intended. After all, I did write it for a reason. However, I am still interested in hearing what modifications are found useful, what machines it eventually wanders onto, and any other interesting paths this thing takes.

If you get this thing running on your home system, make mods to it, make it run on another cpu, or are in a position to make copies of your work for others with similar cpu's, I would appreciate hearing about it in Dr. Dobb's.

I thing it's an excellent opportunity to learn how a compiler works and at the same time establish the necessary groundwork for a C community. It's already here, with the Tiny-C interpreter and the other C compilers now available. I hope to see alot more C programs in the future.

PROGRAM

```

/*****
/*
/*          small-c compiler          */
/*          rev. 1.1                  */
/*          by Ron Cain               */
/*
/*****

/*      Define system dependent parameters      */
/*
/*      Stand-alone definitions                */

/* #define NULL 0          */
/* #define eol 13         */
/*      UNIX definitions (if not stand-alone) */

#include <stdio.h>
#define eol 10
/*      Define the symbol table parameters      */

#define symsiz 14
#define symtbsz 5040
#define numglbs 300
#define startglb symtab
#define endglb startglb+numglbs*symsiz
#define startloc endglb+symsiz
#define endloc symtab+symtbsz-symsiz

/*      Define symbol table entry format      */

#define name 0
#define ident 9
#define type 10
#define storage 11
#define offset 12

/*      System wide name size (for symbols)  */

#define namesize 9
#define namemax 8

/*      Define possible entries for "ident"  */

#define variable 1
#define array 2
#define pointer 3
#define function 4

/*      Define possible entries for "type"   */
#define cchar 1
#define cint 2

/*      Define possible entries for "storage" */

#define statik 1
#define stkloc 2

/*      Define the "while" statement queue  */

#define wqtabsz 100
#define wqsiz 4
#define wqmax wq+wqtabsz-wqsiz

/*      Define entry offsets in while queue  */

#define wqsym 0
#define wqsp 1
#define wqloop 2
#define wqlab 3

/*      Define the literal pool              */

#define litabsz 2000
#define litmax litabsz-1

/*      Define the input line                */

#define linesize 80
#define linemax linesize-1
#define mpmax linemax

/*      Define the macro (define) pool      */

#define macqsize 1000
#define macmax macqsize-1

/*      Define statement types (tokens)     */

#define stif 1
#define stwhile 2
#define streturn 3
#define stbreak 4
#define stcont 5
#define stasm 6
#define stexp 7

/*      Now reserve some storage words      */

```

```

char symtab[symtbsz]; /* symbol table */
char *glbptr,*locptr; /* ptrs to next entries */
int wq[wqtabsz]; /* while queue */
int *wqptr; /* ptr to next entry */

char litq[litabsz]; /* literal pool */
int litptr; /* ptr to next entry */

char macq[macqsize]; /* macro string buffer */
int macptr; /* and its index */

char line[linesize]; /* parsing buffer */
char mline[linesize]; /* temp macro buifer */
int lptr,mptr; /* ptrs into each */

/* Misc storage */
int nxtlab, /* next avail label # */
litlab, /* label # assigned to literal pool */
sp, /* compiler relative stk ptr */
argstk, /* function arg sp */
ncmp, /* # open compound statements */
errcnt, /* # errors in compilation */
eof, /* set non-zero on final input eof */
input, /* iob # for input file */
output, /* iob # for output file (if any) */
input2, /* iob # for "include" file */
glbflag, /* non-zero if internal globals */
ctext, /* non-zero to intermix c-source */
cmode, /* non-zero while parsing c-code */
lastst; /* zero when passing assembly code */

char quote[2]; /* literal string for "" */
char *cptr; /* work ptr to any char buffer */
int *iptr; /* work ptr to any int buffer */
/* >>>> start ccl <<<<<< */
/* Compiler begins execution here */
/*
main()
(
glbptr=startglb; /* clear global symbols */
locptr=startloc; /* clear local symbols */
wqptr=wq; /* clear while queue */
macptr= /* clear the macro pool */
litptr= /* clear literal pool */
sp = /* stack ptr (relative) */
errcnt= /* no errors */
eof= /* not eof yet */
input= /* no input file */
input2= /* or include file */
output= /* no open units */
ncmp= /* no open compound states */
lastst= /* no last statement yet */
quote[1]=
0; /* ...all set to zero.... */
quote[0]=""; /* fake a quote literal */
cmode=1; /* enable preprocessing */
/* compiler body */
/*
ask(); /* get user options */
openout(); /* get an output file */
openin(); /* and initial input file */
header(); /* intro code */
parse(); /* process ALL input */
dumplits(); /* then dump literal pool */
dumpglbs(); /* and all static memory */
errorssummary(); /* summarize errors */
trailer(); /* follow-up code */
closeout(); /* close the output (if any) */
return; /* then exit to system */
)
/*
/* Process all input text
/*
/* At this level, only static declarations,
/* defines, includes, and function
/* definitions are legal...
parse()
(
while (eof==0) /* do until no more input */
(
if(amatch("char",4)){declglb(cchar);ns();}
else if(amatch("int",3)){declglb(cint);ns();}
else if(match("#asm"))doasm();
else if(match("#include"))doinclude();
else if(match("#define"))addmac();
else newfunc();
blanks(); /* force eof if pending */
)
)
/*
/* Dump the literal pool
/*
dumplits()
(int j,k;
if (litptr==0) return; /* if nothing there, exit...*/
printlabel(litlab);col(); /* print literal label */
k=0; /* init an index... */
while (k<litptr) /* to loop with */
{defbyte(); /* pseudo-op to define byte */
j=10; /* max bytes per line */

```



```

while(1)
  (if(argstk==0)return; /* no more args */
  if(match(")")j=pointer;
  else j=variable;
  if(symname(n)=0) illname();
  if(findloc(n))multidef(n);
  if(match("[") /* pointer ? */
  /* it is a pointer, so skip all */
  /* stuff between "[ ]" */
  (while(inbyte()!=" "]
    if(endst())break;
    j=pointer;
    /* add entry as pointer */
  )
  addloc(n,j,t,argstk);
  argstk=argstk-2; /* cnt down */
  if(endst())return;
  if(match(",")==0)error("expected comma");
  )
)
/*
/* Statement parser
/* called whenever syntax requires
/* a statement.
/* this routine performs that statement
/* and returns a number telling which one
statement()
(
  if((ch()==0) & (eof)) return;
  else if(amatch("char",4))
    (dealloc(cchar);ns());
  else if(amatch("int",3))
    (dealloc(cint);ns());
  else if(amatch("(")compound());
  else if(amatch("if",2))
    (doif();lastst=stif);
  else if(amatch("while",5))
    (dowhile();lastst=stwhile);
  else if(amatch("return",6))
    (doreturn();ns();lastst=streturn);
  else if(amatch("break",5))
    (dobreak();ns();lastst=stbreak);
  else if(amatch("continue",8))
    (docont();ns();lastst=stcont);
  else if(match(";"))
    (doasm();lastst=stasm);
  else if(match("#asm"))
    (doasm();lastst=stasm);
  /* if nothing else, assume it's an expression */
  else(expression());ns();lastst=stexp);
  return lastst;
)
/*
/* Semicolon enforcer
/* called whenever syntax requires a semicolon
ns() (if(match(";")==0)error("missing semicolon");)
/*
/* Compound statement
/* allow any number of statements to fall between "{}"
compound()
(
  ++ncmp; /* new level open */
  while(match("}")==0)
    if(eof) return;
    else statement();
  --ncmp; /* close current level */
)
/*
/* "if" statement
doif()
(
  int flcv, fsp, flab1, flab2;
  flcv=locptr; /* record current local level */
  fsp=sp; /* record current stk ptr */
  flab1=getlabel(); /* get label for false branch */
  test(flcv); /* get expression, and branch false */
  statement(); /* if true, do a statement */
  sp=modstk(fsp); /* then clean up the stack */
  locptr=flcv; /* and deallocate any locals */
  if (amatch("else",4)==0) /* if...else ? */
    /* simple "if...print false label */
    (printlabel(flab1);col();nl();
    return; /* and exit */
    )
  /* an "if...else" statement. */
  jump(flcv2=getlabel()); /* jump around false code */
  printlabel(flab1);col();nl(); /* print false label */
  statement(); /* and do "else" clause */
  sp=modstk(fsp); /* then clean up stk ptr */
  locptr=flcv; /* and deallocate locals */
  printlabel(flab2);col();nl(); /* print true label */
)
/*
/* "while" statement
dowhile()
(
  int wq[4]; /* allocate local queue */
  wq[wqsym]=locptr; /* record local level */
  wq[wqsp]=sp; /* and stk ptr */
  wq[wqloop]=getlabel(); /* and looping label */
  wq[wqlab]=getlabel(); /* and exit label */
  addwhile(wq); /* add entry to queue */
  printlabel(wq[wqloop]);col();nl(); /* loop label */
  test(wq[wqlab]); /* see if true */
  statement(); /* if so, do a statement */
  jump(wq[wqloop]); /* loop to label */
  printlabel(wq[wqlab]);col();nl(); /* exit label */
  locptr=wq[wqsym]; /* deallocate locals */
  sp=modstk(wq[wqsp]); /* clean up stk ptr */
  delwhile(); /* delete queue entry */
)
/*
/* "return" statement
doreturn()
(
  /* if not end of statement, get an expression */
  if(endst()==0)expression();
  modstk(0); /* clean up stk */
  ret(); /* and exit function */
)
/*
/* "break" statement
dobreak()
(
  int *ptr;
  /* see if any "whiles" are open */
  if ((ptr=readwhile())==0) return; /* no */
  modstk((ptrfwqsp)); /* else clean up stk ptr */
  jump(ptrfwqlab); /* jump to exit label */
)
/*
/* "continue" statement
docont()
(
  int *ptr;
  /* see if any "whiles" are open */
  if ((ptr=readwhile())==0) return; /* no */
  modstk((ptrfwqsp)); /* else clean up stk ptr */
  jump(ptrfwqloop); /* jump to loop label */
)
/*
/* "asm" pseudo-statement
doasm()
(
  cmode=0; /* mark mode as "asm" */
  while (1)
    (inline(); /* get and print lines */
    if (match("#endasm")) break; /* until... */
    if(eof)break;
    outstr(line);
    nl();
    )
  kill(); /* invalidate line */
  cmode=1; /* then back to parse level */
)
/*
/* >>>> start of cc3 <<<<<<<<<
/*
/* Perform a function call
/* called from heir11, this routine will either call
/* the named function, or if the supplied ptr is
/* zero, will call the contents of HL
callfunction(ptr)
(
  char *ptr; /* symbol table entry (or 0) */
  int nargs;
  nargs=0;
  blanks(); /* already saw open paren */
  if(ptr==0)push(); /* calling HL */
  while(streq(line+ptr,"")==0)
    (if(endst())break;
    expression(); /* get an argument */
    if(ptr==0)swapstk(); /* don't push addr */
    push(); /* push argument */
    nargs=nargs+2; /* count args*2 */
    if (match(",")==0) break;
    )
  needbrack("");
  if(ptr)call(ptr);
  else callstk();
  sp=modstk(sp+nargs); /* clean up arguments */
)
junk()
(
  if(an(inbyte()))
    while(an(ch()))gch();
  else while(an(ch())==0)
    (if(ch()==0)break;
    gch();
    )
  blanks();
)
endst()
(
  blanks();
  return ((streq(line+ptr,"")|(ch()==0)));
)
illname()
(
  error("illegal symbol name");junk();
  multidef(sname)
  char *sname;
  error("already defined");
)

```



```

        comment();
        outstr(sname);nl();
    }
    needbrack(str)
    {
        char *str;
        if (match(str)==0)
            (error("missing bracket");
             comment();outstr(str);nl();
            )
    }
    needlval()
    {
        error("must be lvalue");
    }
    findglb(sname)
    {
        char *sname;
        char *ptr;
        ptr=startglb;
        while(ptr!=glbptr)
            (if(astreq(sname,ptr,namemax))return ptr;
             ptr=ptr+symsiz;
            )
        return 0;
    }
    findloc(sname)
    {
        char *sname;
        char *ptr;
        ptr=startloc;
        while(ptr!=locptr)
            (if(astreq(sname,ptr,namemax))return ptr;
             ptr=ptr+symsiz;
            )
        return 0;
    }
    addglb(sname,id,typ,value)
    {
        char *sname,id,typ;
        int value;
        char *ptr;
        if(cptr=findglb(sname))return cptr;
        if(glbptr==endglb)
            (error("global symbol table overflow");
             return 0;
            )
        cptr=ptr=glbptr;
        while(an(*ptr++ = *sname++)); /* copy name */
        cptr[ident]=id;
        cptr[stype]=typ;
        cptr[storage]=statik;
        cptr[offset]=value;
        cptr[offset+1]=value>>8;
        glbptr=glbptr+symsiz;
        return cptr;
    }
    addloc(sname,id,typ,value)
    {
        char *sname,id,typ;
        int value;
        char *ptr;
        if(cptr=findloc(sname))return cptr;
        if(locptr==endloc)
            (error("local symbol table overflow");
             return 0;
            )
        cptr=ptr=locptr;
        while(an(*ptr++ = *sname++)); /* copy name */
        cptr[ident]=id;
        cptr[stype]=typ;
        cptr[storage]=stkloc;
        cptr[offset]=value;
        cptr[offset+1]=value>>8;
        locptr=locptr+symsiz;
        return cptr;
    }
    /* Test if next input string is legal symbol name */
    symname(sname)
    {
        char *sname;
        int k;char c;
        blanks();
        if(alpha(ch())==0)return 0;
        k=0;
        while(an(ch()))sname[k++]=gch();
        sname[k]=0;
        return 1;
    }
    /* Return next avail internal label number */
    getlabel()
    {
        return(++nxtlab);
    }
    /* Print specified number as label */
    printlabel(label)
    {
        int label;
        outstr("cc");
        outdec(label);
    }
    /* Test if given character is alpha */
    alpha(c)
    {
        char c;
        c=c&127;
        return(((c>="a")&(c<="z"))|
                ((c>="A")&(c<="Z"))|
                (c=="_"));
    }
    /* Test if given character is numeric */
    numeric(c)
    {
        char c;
        c=c&127;
        return((c>="0")&(c<="9"));
    }
    /* Test if given character is alphanumeric */
    an(c)
    {
        char c;
        return((alpha(c))|(numeric(c)));
    }
    /* Print a carriage return and a string only to console */
    pl(str)
    {
        char *str;
        int k;
        k=0;
        putchar(eol);
        while(str[k])putchar(str[k++]);
    }
    addwhile(ptr)
    {
        int ptr[];
        {
            int k;
            if (wqptr==wqmax)
                (error("too many active whiles");return;);
            k=0;
            while (k<wqsiz)
                (*wqptr++ = ptr[k++]);
        }
    }
    delwhile()
    {
        (if(readwhile()) wqptr=wqptr-wqsiz;
         )
    }
    readwhile()
    {
        if (wqptr==wq) (error("no active whiles");return 0;
                       else return (wqptr-wqsiz);
        )
    }
    ch()
    {
        return(line[lptr]&&127);
    }
    nch()
    {
        if(ch()==0)return 0;
        else return(line[lptr+1]&&127);
    }
    gch()
    {
        if(ch()==0)return 0;
        else return(line[lptr++]&&127);
    }
    kill()
    {
        lptr=0;
        line[lptr]=0;
    }
    inbyte()
    {
        while(ch()==0)
            (if (eof) return 0;
             inline();
             preprocess();
            )
        return gch();
    }
    inchar()
    {
        if(ch()==0)inline();
        if(eof)return 0;
        return(gch());
    }
    inline()
    {
        int k,unit;
        while(1)
            (if (input==0)openin();
             if(eof)return;
             if((unit=input2)==0)unit=input;
             kill();
             while((k=getc(unit))>0)
                 (if((k==eol)|(lptr>=linemax))break;
                  line[lptr++]=k;
                  )
             line[lptr]=0; /* append null */
             if(k<=0)
                 (fclose(unit);
                  if(input2)input2=0;
                  else input=0;
                  )
             if(lptr)
                 (if((c=text)&(c=mode))
                  (comment();
                   outstr(line);
                   nl();
                  )
                 )
            )
    }
    /* >>>>> start of cc4 <<<<<<< */
    keepch(c)
    {
        char c;
        mline[mptr]=c;
        if(mptr<mpmax)mptr++;
        return c;
    }
    preprocess()
    {
        int k;
        char c,sname[namesize];
        if(cmode==0)return;
    }

```

```

mptr=lpstr=0;
while(ch())
  {if((ch()==" ")|(ch()==9))
    {keepch(" ");
    while((ch()==" ")|
          (ch()==9))
      gch();
    }
  else if(ch()=="")
    {keepch(ch());
    gch();
    while(ch()!="")
      {if(ch()==0)
        {error("missing quote");
        break;
        }
      keepch(gch());
      }
    gch();
    keepch("");
    }
  else if(ch()==39)
    {keepch(39);
    gch();
    while(ch()!=",")
      {if(ch()==0)
        {error("missing apostrophe");
        break;
        }
      keepch(gch());
      }
    gch();
    keepch(39);
    }
  else if((ch()=="/*")&(nch()=="**"))
    {inchar();inchar();
    while(((ch()=="/*")&
          (nch()=="/*"))==0)
      {if(ch()==0)inline();
      else inchar();
      if(eof)break;
      }
    inchar();inchar();
    }
  else if(an(ch()))
    {k=0;
    while(an(ch()))
      {if(k<namemax)sname[k++]=ch();
      gch();
      }
    sname[k]=0;
    if(k=findmac(sname))
      while(c=macq[k++])
        keepch(c);
    else
      {k=0;
      while(c=sname[k++])
        keepch(c);
      }
    else keepch(gch());
    }
  keepch(0);
  if(mptr>=mpmax)error("line too long");
  lpstr=mptr=0;
  while(line[lpstr++]=mline[mptr++]);
  lpstr=0;
}
addmac()
{
char sname[namesize];
int k;
if(symname(sname)==0)
  {fillname();
  kill();
  return;
  }
k=0;
while(putmac(sname[k++]));
while(ch()==" " | ch()==9) gch();
while(putmac(gch()));
if(macptr>=macmax)error("macro table full");
}
putmac(c)
{
char c;
macq[macptr]=c;
if(macptr<macmax)macptr++;
return c;
}
findmac(sname)
{
char *sname;
int k;
k=0;
while(k<macptr)
  {if(astreq(sname,macq[k],namemax))
    {while(macq[k++]);
    return k;
    }
  while(macq[k++]);
  while(macq[k++]);
  }
return 0;
}
outbyte(c)
char c;

```

```

{
if(c==0)return 0;
if(output)
  {if(putc(c,output)<=0)
    {closeout();
    error("Output file error");
    }
  }
else putchar(c);
return c;
}
outstr(ptr)
char ptr[];
{
int k;
k=0;
while(outbyte(ptr[k++]));
}
nl()
{outbyte(eol);}
tab()
{outbyte(9);}
col()
{outbyte(58);}
error(ptr)
char ptr[];
{
int k;
comment();outstr(line);nl();comment();
k=0;
while(k<lpstr)
  {if(line[k]==9) tab();
  else outbyte(" ");
  ++k;
  }
outbyte("-");
nl();comment();outstr("***** ");
outstr(ptr);
outstr(" *****");
nl();
++errcnt;
}
ol(ptr)
char ptr[];
{
ot(ptr);
nl();
}
ot(ptr)
char ptr[];
{
tab();
outstr(ptr);
}
streq(str1,str2)
char str1[],str2[];
{
int k;
k=0;
while (str2[k])
  {if ((str1[k])!=(str2[k])) return 0;
  k++;
  }
return k;
}
astreq(str1,str2,len)
char str1[],str2[];int len;
{
int k;
k=0;
while (k<len)
  {if ((str1[k])!=(str2[k]))break;
  if(str1[k]==0)break;
  if(str2[k]==0)break;
  k++;
  }
if (an(str1[k]))return 0;
if (an(str2[k]))return 0;
return k;
}
match(lit)
char *lit;
{
int k;
blanks();
if (k=streq(line+lpstr,lit))
  {lpstr=lpstr+k;
  return 1;
  }
return 0;
}
amatch(lit,len)
char *lit;int len;
{
int k;
blanks();
if (k=astreq(line+lpstr,lit,len))
  {lpstr=lpstr+k;
  while(an(ch())) inbyte();
  return 1;
  }
return 0;
}
blanks()

```

```

        (while(1)
            (while(ch()==0)
                (inline();
                 preprocess();
                 if(eof)break;
                )
                if(ch()==" ")gch();
                else if(ch()==9)gch();
                else return;
            )
        )
    outdec(number)
    int number;
    (
        int k,zs;
        char c;
        zs = 0;
        k=10000;
        if (number<0)
            (number=(-number);
             outbyte("-");
            )
        while (k>=1)
            (
                c=number/k + "0";
                if ((c!="0")|(k==1)|(zs))
                    (zs=1;outbyte(c));
                number=number%k;
                k=k/10;
            )
    )
    /* >>>>>> start of cc5 <<<<<<< */
expression()
(
    int lval[2];
    if(heir1(lval))rvalue(lval);
)
heir1(lval)
int lval[];
(
    int k,lval2[2];
    k=heir2(lval);
    if (match("="))
        (if(k==0)(needlval());return 0;
         if (lval[1])push();
         if(heir1(lval2))rvalue(lval2);
         store(lval);
         return 0;
        )
    else return k;
)
heir2(lval)
int lval[];
(
    int k,lval2[2];
    k=heir3(lval);
    blanks();
    if(ch()!="|")return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("|"))
            (push();
             if(heir3(lval2)) rvalue(lval2);
             pop();
             or();
            )
        )
        else return 0;
    )
)
heir3(lval)
int lval[];
(
    int k,lval2[2];
    k=heir4(lval);
    blanks();
    if(ch()!="^")return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("^"))
            (push();
             if(heir4(lval2))rvalue(lval2);
             pop();
             xor();
            )
        )
        else return 0;
    )
)
heir4(lval)
int lval[];
(
    int k,lval2[2];
    k=heir5(lval);
    blanks();
    if(ch()!="&")return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("&"))
            (push();
             if(heir5(lval2))rvalue(lval2);
             pop();
             and();
            )
        )
        else return 0;
    )
)
heir5(lval)
int lval[];

```

```

(
    int k,lval2[2];
    k=heir6(lval);
    blanks();
    if((streq(line+lptr,"=")==0)&
        (streq(line+lptr,"|")==0))return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("=|"))
            (push();
             if(heir6(lval2))rvalue(lval2);
             pop();
             eq();
            )
        )
        else if (match("&"))
            (push();
             if(heir6(lval2))rvalue(lval2);
             pop();
             ne();
            )
        )
        else return 0;
    )
)
heir6(lval)
int lval[];
(
    int k,lval2[2];
    k=heir7(lval);
    blanks();
    if((streq(line+lptr,"<")==0)&
        (streq(line+lptr,">")==0)&
        (streq(line+lptr,"<=")==0)&
        (streq(line+lptr,">=")==0))return k;
    if(streq(line+lptr,">>"))return k;
    if(streq(line+lptr,"<<<"))return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("<="))
            (push();
             if(heir7(lval2))rvalue(lval2);
             pop();
             if(cpctr=lval[0])
                 if(cpctr[ident]==pointer)
                     (ule();
                      continue;
                     )
                 if(cpctr=lval2[0])
                     if(cpctr[ident]==pointer)
                         (ule();
                          continue;
                         )
            )
            le();
        )
        else if (match(">="))
            (push();
             if(heir7(lval2))rvalue(lval2);
             pop();
             if(cpctr=lval[0])
                 if(cpctr[ident]==pointer)
                     (uge();
                      continue;
                     )
                 if(cpctr=lval2[0])
                     if(cpctr[ident]==pointer)
                         (uge();
                          continue;
                         )
            )
            ge();
        )
        else if((streq(line+lptr,"<<"))&
                (streq(line+lptr,"<<=")==0))
            (inbyte();
             push();
             if(heir7(lval2))rvalue(lval2);
             pop();
             if(cpctr=lval[0])
                 if(cpctr[ident]==pointer)
                     (ult();
                      continue;
                     )
                 if(cpctr=lval2[0])
                     if(cpctr[ident]==pointer)
                         (ult();
                          continue;
                         )
            )
            lt();
        )
        else if((streq(line+lptr,">>"))&
                (streq(line+lptr,">>=")==0))
            (inbyte();
             push();
             if(heir7(lval2))rvalue(lval2);
             pop();
             if(cpctr=lval[0])
                 if(cpctr[ident]==pointer)
                     (ugt();
                      continue;
                     )
                 if(cpctr=lval2[0])
                     if(cpctr[ident]==pointer)
                         (ugt();
                          continue;
                         )
            )
            gt();
        )
    )
)

```

```

        else return 0;
    }
}
/* >>>>> start of cc6 <<<<<< */
heir7(lval)
int lval[];
{
    int k, lval2[2];
    k=heir8(lval);
    blanks();
    if((streq(line+lptr,">>")==0)&
        (streq(line+lptr,"<<")==0))return k;
    if(k)rvalue(lval);
    while(1)
        (if (match(">>"))
            (push();
             if(heir8(lval2))rvalue(lval2);
             pop();
             asr();
            )
         else if (match("<<"))
            (push();
             if(heir8(lval2))rvalue(lval2);
             pop();
             asl();
            )
         else return 0;
        )
}
heir8(lval)
int lval[];
{
    int k, lval2[2];
    k=heir9(lval);
    blanks();
    if((ch()!="+")&(ch()!="-"))return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("+"))
            (push();
             if(heir9(lval2))rvalue(lval2);
             if(cptr=lval[0])
                 if((cptr[ident]==pointer)&
                    (cptr[type]==cint))
                     doublereg();
             pop();
             add();
            )
         else if (match("-"))
            (push();
             if(heir9(lval2))rvalue(lval2);
             if(cptr=lval[0])
                 if((cptr[ident]==pointer)&
                    (cptr[type]==cint))
                     doublereg();
             pop();
             sub();
            )
         else return 0;
        )
}
heir9(lval)
int lval[];
{
    int k, lval2[2];
    k=heir10(lval);
    blanks();
    if((ch()!="*")&(ch()!="/")&
        (ch()!="%"))return k;
    if(k)rvalue(lval);
    while(1)
        (if (match("*"))
            (push();
             if(heir9(lval2))rvalue(lval2);
             pop();
             mult();
            )
         else if (match("/"))
            (push();
             if(heir10(lval2))rvalue(lval2);
             pop();
             div();
            )
         else if (match("%"))
            (push();
             if(heir10(lval2))rvalue(lval2);
             pop();
             mod();
            )
         else return 0;
        )
}
heir10(lval)
int lval[];
{
    int k;
    char *ptr;
    if(match("**"))
        (if((k=heir10(lval))==0)
            (needlval();
             return 0;
            )
         if(lval[1])push();
         rvalue(lval);
        )
}
        inc();
        ptr=lval[0];
        if((ptr[ident]==pointer)&
            (ptr[type]==cint))
            inc();
        store(lval);
        return 0;
    }
}
else if(match("--"))
    (if((k=heir10(lval))==0)
        (needlval();
         return 0;
        )
     if(lval[1])push();
     rvalue(lval);
     dec();
     ptr=lval[0];
     if((ptr[ident]==pointer)&
        (ptr[type]==cint))
         dec();
     store(lval);
     return 0;
    )
}
else if (match("-"))
    (k=heir10(lval);
     if (k) rvalue(lval);
     neg();
     return 0;
    )
}
else if (match("*"))
    (k=heir10(lval);
     if(k)rvalue(lval);
     lval[1]=cint;
     if(ptr=lval[0])lval[1]=ptr[type];
     lval[0]=0;
     return 1;
    )
}
else if (match("&"))
    (k=heir10(lval);
     if(k=0)
         (error("illegal address");
          return 0;
         )
     else if (lval[1])return 0;
     else
         (immed();
          outstr(ptr=lval[0]);
          nl();
          lval[1]=ptr[type];
          return 0;
         )
    )
}
else
    (k=heir11(lval);
     if(match("++"))
         (if(k=0)
             (needlval();
              return 0;
             )
          if(lval[1])push();
          rvalue(lval);
          inc();
          ptr=lval[0];
          if((ptr[ident]==pointer)&
             (ptr[type]==cint))
              inc();
          store(lval);
          dec();
          if((ptr[ident]==pointer)&
             (ptr[type]==cint))
              dec();
          return 0;
          )
         else if (match("--"))
             (if(k=0)
                 (needlval();
                  return 0;
                 )
              if(lval[1])push();
              rvalue(lval);
              dec();
              ptr=lval[0];
              if((ptr[ident]==pointer)&
                 (ptr[type]==cint))
                  dec();
              store(lval);
              inc();
              if((ptr[ident]==pointer)&
                 (ptr[type]==cint))
                  inc();
              return 0;
              )
             else return k;
            )
        )
}
/* >>>>> start of cc7 <<<<<< */
heir11(lval)
{
    int *lval;
    int k;char *ptr;
    k=primary(lval);
    ptr=lval[0];
    blanks();
    if((ch()=="["|ch()=="("))
        while(1)

```



```

/*      static memory cell */
putmem(sym)
char *sym;
(
  if((sym[ident1]=pointer)&(sym[type]==cchar))
    (ol("MOV A,L");
     ot("STA ");
    )
  else ot("SHLD ");
  outstr(sym+name);
  nl();
)
/* Store the specified object type in the primary register */
/* at the address on the top of the stack */
putstk(typeobj)
char typeobj;
(
  pop();
  if(typeobj==cchar)call("ccpchar");
  else call("ccpint");
)
/* Fetch the specified object type indirect through the */
/* primary register into the primary register */
indirect(typeobj)
char typeobj;
(
  if(typeobj==cchar)call("ccgchar");
  else call("ccgint");
)
/* Swap the primary and secondary registers */
swap()
(
  ol("XCHG");
)
/* Print partial instruction to get an immediate value */
/* into the primary register */
immed()
(
  ol("LXI H,");
)
/* Push the primary register onto the stack */
push()
(
  ol("PUSH H");
  sp=sp-2;
)
/* Pop the top of the stack into the secondary register */
pop()
(
  ol("POP D");
  sp=sp+2;
)
/* Swap the primary register and the top of the stack */
swapstk()
(
  ol("XTHL");
)
/* Call the specified subroutine name */
call(sname)
char *sname;
(
  ot("CALL ");
  outstr(sname);
  nl();
)
/* Return from subroutine */
ret()
(
  ol("RET");
)
/* Perform subroutine call to value on top of stack */
callstk()
(
  immed();
  outstr("$+5");
  nl();
  swapstk();
  ol("FCHL");
  sp=sp+2;
)
/* Jump to specified internal label number */
jump(label)
(
  int label;
  ot("JMP ");
  printlabel(label);
  nl();
)
/* Test the primary register and jump if false to label */
testjump(label)
(
  int label;
  ol("MOV A,H");
  ol("ORA L");
  ot("JZ ");
  printlabel(label);
  nl();
)
/* Print pseudo-op to define a byte */
defbyte()
(
  ot("DB ");
)
/* Print pseudo-op to define storage */
defstorage()
(
  ot("DS ");
)
/* Print pseudo-op to define a word */
defword()
(
  ot("DW ");
)
/* Modify the stack pointer to the new value indicated */
modstk(newsp)
(
  int newsp;
  int k;
  k=newsp-sp;
  if(k==0)return newsp;
  if(k>=0)
    (if(k<7)
     (if(k&1)
      (ol("INX SP");
       k--;
      )
     while(k)
      (ol("POP B");
       k=k-2;
      )
     return newsp;
    )
   )
   if(k<0)
    (if(k>=-7)
     (if(k&1)
      (ol("DCX SP");
       k++;
      )
     while(k)
      (ol("PUSH B");
       k=k+2;
      )
     return newsp;
    )
   )
   swap();
   immed();outdec(k);nl();
   ol("DAD SP");
   ol("SPHL");
   swap();
   return newsp;
  )
  /* Double the primary register */
  doublereg()
  (
    ol("DAD H");
  )
  /* Add the primary and secondary registers */
  /* (results in primary) */
  add()
  (
    ol("DAD D");
  )
  /* Subtract the primary register from the secondary */
  /* (results in primary) */
  sub()
  (
    call("ccsub");
  )
  /* Multiply the primary and secondary registers */
  /* (results in primary) */
  mult()
  (
    call("ccmult");
  )
  /* Divide the secondary register by the primary */
  /* (quotient in primary, remainder in secondary) */
  div()
  (
    call("ccdiv");
  )
  /* Compute remainder (mod) of secondary register divided */
  /* by the primary */
  /* (remainder in primary, quotient in secondary) */
  mod()
  (
    div();
    swap();
  )
  /* Inclusive "or" the primary and the secondary registers */
  /* (results in primary) */
  or()
  (call("ccor");)
  /* Exclusive "or" the primary and secondary registers */
  /* (results in primary) */
  xor()
  (call("ccxor");)
  /* "And" the primary and secondary registers */
  /* (results in primary) */
  and()
  (call("ccand");)
  /* Arithmetic shift right the secondary register number of */
  /* times in primary (results in primary) */
  asr()
  (call("ccasr");)
  /* Arithmetic left shift the secondary register number of */
  /* times in primary (results in primary) */
  asl()
  (call("ccasl");)
  /* Form two's complement of primary register */
  neg()
  (call("ccneg");)
  /* Form one's complement of primary register */
  com()
  (call("cccocom");)
  /* Increment the primary register by one */
  inc()
  (ol("INX H");)
  /* Decrement the primary register by one */
  dec()
  (ol("DCX H");)

  /* Following are the conditional operators */
  /* They compare the secondary register against the primary */
  /* and put a literal 1 in the primary if the condition is */
  /* true, otherwise they clear the primary register */

  /* Test for equal */
  eq()
  (call("cceq");)

```

```

/* Test for not equal */
ne()
    (call("ccne"));
/* Test for less than (signed) */
lt()
    (call("cclt"));
/* Test for less than or equal to (signed) */
le()
    (call("ccle"));
/* Test for greater than (signed) */
gt()
    (call("ccgt"));
/* Test for greater than or equal to (signed) */
ge()
    (call("ccge"));
/* Test for less than (unsigned) */
ult()
    (call("ccult"));
/* Test for less than or equal to (unsigned) */
ule()
    (call("ccule"));
/* Test for greater than (unsigned) */
ugt()
    (call("ccugt"));
/* Test for greater than or equal to (unsigned) */
uge()
    (call("ccuge"));

/* <<<< End of compiler >>>> */

```

