CMSC 430
Spring 2007

NIP COMPILER for HAC 430 Computer – version 2007-2

1. Language Requirements

Compiler Structure

NIP will process user files and produce an output file of machine language for the HAC 430 computer (see HAC 430 Principles of Operations)

NIP Input: NIP will read via standard input a file containing a NIP program. On the NIP command can appear options. The syntax is:

nip -options

The only option specified so far is the s option.

nip -s

means to write the source program onto the standard output file.

Additional options will be specified later. Any NIP implementation may have additional options beyond those required.

NIP Output: NIP will produce a listing of the program to the standard output file (if the s option is specified). In addition, a file of machine language for the HAC 430 will be created in the file nip.out.

Error messages will appear mixed with the program listing.

Examples:

(1) A standard compilation with the program in *FileName* and listing to appear on terminal:

nip < FileName

(2) A compilation with the s option from file *FileName* with listing to go to *FileName.lst*:

nip -s <FileName >FileName.lst

Compiler Design: NIP will consist of 4 parts:

- **Scanner** - using lex or flex to produce a list of tokens. Specification to follow.
- **Parser**- using yacc or bison to produce a list of quads. Specification to follow.
- **Symbol table**- Produce a symbol table of references. Specification to follow.
- **Code Gen**- Produce a file of HAC 430 machine code. Specification to follow.
- **Optimizer** – Produce a list of quads that represents the optimized program. Specifications to follow.

NIP Language

NIP contain the following syntax. Lower case terms are monterminals, upper case terms are tokens in the language. *start_here* is the start symbol. This grammar must be modified to work correctly through bison.

SYNTAX:

(1) start_here --> program
(2) program --> procs START ID
(3) program → procs START ID (exprlist)
(4) procs --> procedure
(5) procs --> procs procs
(6) procedure --> proc_head declist proclist { stmntlist }
(7) proc_head --> ID FUNCTION parmlist
(8) proc_head --> ID PROCEDURE parmlist
(9) parmlist -->
(10) parmlist --> (plist)
(11) plist --> type_name : idlist
(12) plist --> type_name ARRAY : alist
(13) plist --> plist; type_name: idlist
(14) plist --> plist; type_name ARRAY : alist
(15) type_name --> CHAR
(16) type_name --> INTEGER
(17) declist -->
(18) declist --> decl ;
(19) proclist -->
(20) proclist --> procs
(21) idlist --> ID
(22) idlist --> idlist , idlist
(23) alist --> ID (CONSTANT)
(24) alist --> alist , alist
(25) decl --> type_name idlist
(26) decl --> type_name ARRAY alist
(27) decl --> decl ; type_name idlist
(28) decl --> decl ; type_name ARRAY alist
(29) stmntlist --> stmntlist ; stmntlist
(30) stmntlist --> statement
(31) statement --> ID := expression
(32) statement --> ID (expression) := expression
(33) statement --> IF lexpression { stmntlist }
(34) statement --> IF lexpression { stmntlist ELSE stmntlist }
(35) statement --> WHILE lexpression { stmntlist }
(36) statement --> REPEAT lexpression { stmntlist }
(37) statement --> CALL ID
(38) statement --> CALL ID (exprlist)
(39) statement --> READ

(40) statement --> READ (idlist)
(41) statement --> WRITE
(42) statement --> WRITE (exprlist)
(43) statement --> RETURN (expression)
(44) exprlist --> expression
(45) exprlist --> exprlist , exprlist
(46) expression --> expression addop expression
(47) expression --> term
(48) lexpression --> expression relop expression
(49) lexpression --> expression
(50) relop --> >
(51) relop --> <
(52) relop --> =
(53) relop --> <>
(54) term --> term mulop term
(55) term --> primary
(56) primary --> ID
(57) primary --> CONSTANT
(58) primary --> LITERAL
(59) primary --> ID (exprlist)
(60) primary --> ID ( )
(61) primary --> (expression)
(62) addop --> +
(63) addop --> -
(64) mulop --> *
(65) mulop --> /

Additional syntax rules:

(1) ID represents an identifier - from one to 20 letters or digits starting with a letter. Upper and lower case are the same (e.g., *ABC, abc,* and *aBc* are the same identifier).

(2) CONSTANT represents an integer constant from 1 to 8 digits.

(3) LITERAL represents a string of the form "list" where *list* contains up to 100 characters. The character " is represented in the list as "". Any character except *new-line* may appear within a LITERAL.

(4) All other special tokens represent themselves: START, FUNCTION, PROCEDURE, ARRAY, CHAR, INTEGER, IF, ELSE, WHILE, REPEAT, CALL, READ, WRITE, RETURN. Case is independent (e.g. *IF* and *if* are the same.) These are all reserved words.

(5) Comments shall be any string of symbols that begins // and is terminated by either // or the end of the current line, whichever comes first.

(6) Blanks and end of line characters terminate tokens. That is, blanks may not be embedded within tokens.

Semantic Rules:

(1) RETURN is only valid in a function. All functions must return a value of type INTEGER.

(2) Functions with no arguments are invoked as *FunctionName()*.

(3) *Read* of a CHAR gets the next character from the input. Data wraps from the end of one line onto the next. *Read* of an INTEGER skips blanks and gets the next integer value. If *Read* has no identifier list, then the next line of input is read.

.

(4) *Write* a LITERAL writes it on the output without enclosing " symbols. *Write* of an INTEGER formats it and prints it. *Write* with no expression list writes a *newline* character.

(5) Standard block structured scope rules apply. All variables must be declared.

(6) Execution begins with the procedure named ID in START ID. Arguments to this procedure may be expressions involving constants.

(7) Semantics of IF and WHILE statements: > is greater than, < is less than, = is equality and <> is inequality. A positive expression value is true, all others are false.

(8) All parameters are call by reference. It must not be possible to change the value of a constant or literal via a parameter.

(9) Procedures may appear in any order and may be called before defined, but usual scope rules apply. All procedures and functions may be recursive. Indirect recursion is allowed.

(10) Assigning a CHAR value to an INTEGER assigns the internal ASCII representation to the integer. Similarly, assigning an INTEGER variable to a CHAR variable assigns the CHAR as the ASCII character represented by the integer value.

(11) Operations on literals: A:="abc" assigns "a" to CHAR variable A. Similarly for passing parameters.

(12) CHAR and LITERAL data may not appear in arithmetic expressions. They only may appear as right hand side operands in an assignment statement, as arguments in a parameter list, or as arguments in a *write* statement.

(13) Array A(n) goes from 0 through n (i.e., n+1 elements in array).

2. HAC 430 Principles of Operation

The Hypothetical Abstract Computer (HAC) 430 runs as an interpreter on the grace machines. NIP is a compiler that generates code for this machine. **Note: As of now, the hac430 only runs on the solaris system and not under linux. So if you want to use the interpreter, you have to log onto grace as solaris.grace.umd.edu. It will fail if you are one of the linux machines.**

Accessing the HAC 430

In order to run the HAC 430, use the command:

```
hac430 FileName
```

where *FileName* is the output from the nip compiler. Thus to compile and execute a program from file *MyProgram*, use the following:

```
nip < MyProgram
hac430 nip.out
```

Output from the machine will appear on your terminal and input to the machine will follow the hac command line.

The HAC 430 interpreter will be in the file: ~mvz/hac430. Add the following command to your .login file:

```
alias hac430,'~mvz/hac430'
```

This will cause your command *hac430* to correctly access the interpreter.

Machine architecture

The HAC 430 is a decimal computer with 5,000 words of memory. Each word contains 8 digits. There are also 10 registers, numbered 0 through 9, that contain data. Register 0 has a special purpose, which is described below. Most instructions contain an operation, a register number and a memory address and apply data from memory to that register.

Instructions: Each instruction contains 8 digits and is of the following format:

```
OPRIXXXX
```

where:

OP is an operation code. These are given in a later section of this document.

IR is a register number.

I is a register number used as an index register. This is used in computing the effective address (EA). A zero (0) means no index register, thus register 0 cannot be used as an index register.

XXXX is the relative offset, and is used in computing the effective address (EA). If XXXX is between 0 and 4,999, then the INDIRECT flag is set to false and OFFSET is set to XXXX. If XXXX is between 5,000 and 9,999, then the INDIRECT flag is set to true and OFFSET is set to

XXXX-5000 (e.g., a relative offset between 0 and 4,999).

Effective Address Calculation: Let c(M) be the contents of memory location M and let r(R) be the contents of register R. The effective address (EA) for most instructions is computed as follows:

(1) If XXXX is the relative offset of some instruction, let INDIRECT and OFFSET be determined as specified above.
(2) Let I be the index register of that instruction.
(a) If I<>0 then RELADDR := OFFSET+r(I) else RELADDR := OFFSET.
(b) IF INDIRECT is true then EA := c(RELADDR) else EA := RELADDR.

Loader Format

Information passed to the HAC 430 computer (which is also the output from the NIP compiler) has the following format.

(1) The file contains a sequence of records, each record is an address and contents or a start address. There is only one start address record in a file.

(2) Data (instruction) records have the format:

XXXX YY ZZZZZZZZ

where:

XXXX is an address
YY is the size of the data for this address (in number of characters)
ZZZZZZZZ is from one to 8 digits. The number is specified by YY.

For example, to store the constant 123 at location 4356, the following record can be used:

4356 03 123

or the following does the same thing:

4356 08 00000123

The instruction L 2,3,2468 (Load reg. 2 with offset 2468 indexed by reg. 3) at address 501 is given by the record:
.ce
0501 08 01232468
(Operation code 01 is the load instruction.)
.sp
(3) The start address record has the format:
.ce
XXXX 00
where XXXX is the location where execution is to begin. It must be the last record of the file.

Instructions

For each of the following instructions, the format is:

OP R I XXXX

and EA refers to the effective address, described previously.

00 - SO	Set Option - The EA is computed. EA=0 - Do nothing. EA=1 - Debug mode. Print current address. When machine halts, print stop address. EA=2 - Debug mode 2. Print address and instruction before each instruction executed. EA=3 - Turn off all options. EA=9 - Halt machine.
01 - L	Load - c(EA) --> r(R)
02 - ST	Store - r(R) --> c(EA)
03 - A	Add - c(EA) + r(R) --> r(R). Overflow flag is set if overflow occurs.
04 - S	Subtract - r(R) - c(EA) --> r(R). Overflow flag is set if overflow occurs.
05 - M	Multiply - r(R) * c(EA) --> r(R,R+1). R must be an even numbered register, and results take 16 digits.
06 - D	Divide - r(R) / c(EA) --> r(R,R+1). R must be even register. Quotient in R, Remainder in R+1.
07 - IO	I/O - Compute EA. EA=1. Read next character into low order 3 digits of register 0. ASCII code is used. EA=2. Write low order 3 digits as ASCII character from register 0 to terminal. r(0) must be between 0 and 127. EA=3. Read next string of characters and convert to integer and store in register 0. EA=4. Print out contents of register 0 as an integer. EA=5. Go to next line of input. EA=6. Go to next line of output.
1X - SK	Skip instructions. Skip next instruction if in comparing r(R) to c(EA), the condition is true. X=1 SKLT - Skip less than (i.e., r(R) less than c(EA)) X=2 SKLE - Skip less than or equal X=3 SKGT - Skip greater than X=4 SKGE - Skip greater than or equal to X=5 SKEQ - Skip equal to X=6 SKNE - Skip not equal to X=7 SKOV - Skip if overflow flag set X=8 SKNO - Skip is no overflag set Overflow only changed by A, S, AI and SI instructions.
21 - LI	Load immediate - EA --> r(R)
23 - AI	Add immediate - EA + r(R) --> r(R)
24 - SI	Subtract immediate - r(R) - EA --> r(R)
25 - MI	Multiply immediate - r(R) * EA --> r(R,R+1). Same as M.
26 - DI	Divide immediate- r(R) / EA --> r(R,R+1). Same as D.
27 - J	Jump. Next instruction executed will be c(EA).

	This instruction following one of the skip instructions is used to conditional jumps.
28 - JSR	Jump to subroutine: Next location --> r(R), Next instruction executed will be c(EA). If JSR R,I,XXXX is used to go to a subroutine, then J 0,R,0000 is used to return.
37 - JI	Jump immediate. The same as the Jump instruction (J - 27) except that the next instruction address is EA.
38 - JSRI	Jump to subroutine immediate - The same as the JSR instruction (28) except that the next instruction address is EA.

HAC 430 error messages

If the HAC430 interpreter finds an error in a program, it will display a three digit message and halt. The following messages are defined. If any other message appears, then please inform Dr. Zelkowitz since this will probably mean an error in the interpreter itself.

601	Not even register in D instruction (6)
701	Operand not in range 1..6 for IO (7) instruction
702	ASCII character not in range 0..127 in register 0 for IO 2 instruction
800	Invalid argument for operation SO (0)
802	Address not in range 0..4999 for J (27) or JI (37) instruction
803	Address not in range 0..4999 for JSR (28) or JSRI (38) instruction
808	Invalid operation code
991	Not even register in instruction M (5)
992	Invalid instruction address (Not in range 0..4999)
993	Invalid effective address (Not in range 0..4999)
995	Loader error - Instruction length > 8 digits
996	Loader error - Instruction size field not followed by blank
997	Loader error - Address location not followed by blank
998	Loader error - Insufficient digits in instruction field
999	Read past end of file