THE UNIX PROGRAMMING ENVIRONMENT

Brian W. Kernighan
Rob Pike

Contents

Chapter 1

UNIX for Beginners

What is "UNIX"? In the narrowest sense, it is a time-sharing operating system kernel: a program that controls the resources of a computer and allocates them among its users. It lets users run their programs; it controls the peripheral devices (discs, terminals, printers, and the like) connected to the machine; and it provides a file system that manages the long-term storage of information such as programs, data, and documents.

In a broader sense, "UNIX" is often taken to include not only the kernel, but also essential programs like compilers, editors, command languages, programs for copying and printing files, and so on.

Still more broadly, "UNIX" may even include programs developed by you or other users to be run on your system, such as tools for document preparation, routines for statistical analysis, and graphics packages.

Which of these uses of the name "UNIX" is correct depends on which level of the system you are considering. When we use "UNIX" in the rest of this book, context should indicate which meaning is implied.

The UNIX system sometimes looks more difficult than it is — it's hard for a newcomer to know how to make the best use of the facilities available. But fortunately it's not hard to get started — knowledge of only a few programs should get you off the ground. This chapter is meant to help you to start using the system as quickly as possible. It's an overview, not a manual; we'll cover most of the material again in more detail in later chapters. We'll talk about these major areas:

- basics — logging in and out, simple commands, correcting typing mistakes, mail, inter-terminal communication.
- day-to-day use — files and file system, printing files, directories, commonly-used commands.
- the command interpreter or shell — filename shorthands, redirecting intput and output, pipes, setting erase and kill characters, and defining your own search path for commands.

If you've used a UNIX system before, most of this chapter should be familiar; you might want to skip straight to Chapter 2.

You will need a copy of the *UNIX Programmer's Manual*, even as you read this chapter; it's often easier for us to tell you to read about something in the manual than to repeat its contents here. This book is not supposed to replace it, but to show you how to make best use of the commands described in it. Furthermore, there may be differences between what we say here and what is true on your system. The manual has a permuted index at the beginning that's indispensable for finding the right programs to apply to a problem; learn to use it.

Finally, a word of advice: don't be afraid to experiment. If you are a beginner, there are very few accidental things you can do to hurt yourself or other users. So learn how things work by trying them. This is a long chapter, and the best way to read it is a few pages at a time, trying things out as you go.

1.1 Getting Started

1.1.1 Some prerequisites about terminals and typing

To avoid explaining everything about using computers, we must assume you have some familiarity with computer terminals and how to use them. If any of the following statements are mystifying, you should ask a local expert for help.

The UNIX system is *full duplex*: the characters you type on the keyboard are sent to the system, which sends them back to the terminal to be printed on the screen. Normally, this *echo* process copies the characters directly to the screen, so you can see what you are typing, but sometimes, such as when you are typing a secret password, the echo is turned off so the characters do not appear on the screen.

Most of the keyboard characters are ordinary printing characters with no special significance, but a few tell the computer how to interpret your typing. By far the most important of these is the RETURN key. The RETURN key signifies the end of a line of input; the system echoes it by moving the terminal's cursor to the beginning of the next line on the screen. RETURN must be pressed before the system will interpret the characters you have types.

RETURN is an example of a *control character* — an invisible character that controls some aspect of input and output on the terminal. On any reasonable terminal, RETURN has a key of its own, but most control characters do not. Instead, they must be typed by holding down the CONTROL key, sometimes called CTL or CNTL or CTRL, then pressing another key, usually a letter. For example, RETURN may be typed by pressing the RETURN key or, equivalently, holding down the CONTROL key and typing an 'm'. RETURN might therefore be called a control-m, which we will write as *ctl*-m. Other control characters include *ctl*-d, which tells a program that there is no more input; *ctl*-g, which rings the bell on the terminal; *ctl*-h, often called backspace, which can be used to correct typing mistakes; and *ctl*-i, often called tab, which advances the cursor to the next tab stop, much as on a regular typewriter. Tab stops on UNIX systems are eight spaces apart. Both the backspaces and tab characters have their keys on the terminals.

Two other keys have special meaning: DELETE, sometimes called RUBOUT or some abbreviation, and BREAK, sometimes called INTERRUPT. On most UNIX systems, the DELETE key stops a program immediately, without waiting for it to finish. On some systems, *ctl*-c provides this device. And on some systems, depending on how the terminals are connected, BREAK is synonym for DELETE or *ctl*-c.

1.1.2 A Session with UNIX

Let's begin with an annotated dialog between you and your UNIX system. Throughout the examples in this book, what you type is printed in slanted letters, computer responses are in typewriter-style characters, and explanations are in *italics*.

```
Establish a connection: dial a phone or turn on a switch as necessary.
Your system should say
login: you                 Type your name, then press RETURN
Password:                  Your password won't be echoed as you type it
You have mail.             There's mail to be read after you log in
$                          The system is now ready for your commands
$                          Press RETURN a couple of times
$ date                     What is the date and time?
Sun Sep 25 23:02:57 EDT 1983
$ who                      Who's using the computer?
jlb      tty0    Sep 25 13:59
you      tty2    Sep 25 23:01
mary     tty4    Sep 25 19:03
doug     tty5    Sep 25 19:22
egb      tty7    Sep 25 17:17
bob      tty8    Sep 25 20:48
$ mail                     Read your mail
From doug Sun Sep 25 20:53 EDT 1983
give me a call sometime monday

?                          RETURN moves on to the next message
From mary Sun Sep 25 19:07 EDT 1983     Next message

? d                        Delete this message
$                          No more mail
$ mail mary                Send mail to mary
lunch at 12 is fine
ctl-d                      End of mail
$                          Hang up phone or turn off terminal
                           and that's the end
```

Sometimes that's all there is to a session, though occasionally people do some work too. The rest of this section will discuss the session above, plus other programs that make it possible to do useful things.

1.1.3 Logging in

You must have a login name and password, which you can get from your system administrator. The UNIX system is capable of dealing wit a wide variety of terminals, but it is strongly oriented towards devices with lower case; case distinctions matter! If your terminal produces only upper case (like some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure the switches are set appropriately on your device: upper and lower case, full duplex, and any other settings that local experts advise, such as the speed, or *baud rate*. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone or merely flipping a switch. In either case, the system should type

```
login:
```

If it types garbage, you may be at the wrong speed; check the speed setting and other switches. If that fails, press the BREAK or INTERRUPT key a few times, slowly. If nothing produces a login message, you will have to get help.

When you get the `login:` message, type your login name *in lower case*. Follow it by pressing RETURN. If a password is required, you will be asked for it, and printing will be turned off while you type it.

The culmination of your login efforts is a *prompt*, usually a single character, indicating that the system is ready to accept commands from you. The prompt is mostly likely to be a dollar sign `$` or a percent sign `%`, but you can change it to anything you like; we'll show you how a little later. The prompt is actually printed by a program called the *command interpreter* or *shell*, which is your main interface to the system.

There may be a message of the day just before the prompt, or a notification that you have mail. You may also be asked what kind of terminal you are using; your answer helps the system to use any special properties the terminal might have.

1.1.4 Typing commands

Once you receive the prompt, you can type commands, which are requests that the system do something. We will use *program* as a synonym for *command*. When you see the prompt (let's assume it's `$`), type date and press RETURN. the system should replay with the date and time, then print another prompt, so the whole transaction will look like this on your terminal:

```
$ date
Mon Sep 26 12:20:57 EDT 1983
$
```

Don't forget RETURN, and don't type the `$`. If you think you're being ignored, press RETURN; something should happen. RETURN won't be mentioned again, but you need it at the end of every line.

The next command to try is who, which tells you everyone who is currently logged in:

```
$ who
rim     tty0     Sep 26 11:37
pjw     tty4     Sep 26 11:30
gerard  tty7     Sep 26 10:27
mark    tty9     Sep 26 07:59
you     ttya     Sep 26 12:20
$
```

The first column is the user name. The second is the system's name for the connection being used ("tty" stands for "teletype," an archaic synonym for "terminal"). The rest tells when the user logged on. You might also try

```
$ who am i
you     ttya     Sep 26 12:20
$
```

If you make a mistake typing the name of a command, and refer to a non-existent command, you will be told that no command of that name can be found:

```
$ whom                       Misspelled command name ...
whom: not found              ... so system didn't know how to run it
$
```

Of course, if you inadvertently type the name of an actual command, it will run, perhaps with mysterious results.

1.1.5 Strange terminal behavior

Sometimes your terminal will act strangely, for example, each letter may be typed twice, or RETURN may not put the cursor at the first column of the next line. You can usually fix this by turning the terminal off and on, or by logging out and logging back in. Or you can read the description of the command `stty` (“set terminal options”) in Section 1 of the manual. To get intelligent treatment of tab characters if your terminal doesn’t have tabs, type the command

```
$ stty -tabs
```

and the system will convert tabs into the right number of spaces. If your terminal does have computer-settable tab stops, the command tabs will set them correctly for you. (You may actually have to say

```
$ tabs terminal-type
```

to make it work — see the `tabs` command description in the manual.)

1.1.6 Mistakes in typing

1.1.7 Type-ahead

1.1.8 Stopping a program

1.1.9 Logging out

1.1.10 Mail

1.1.11 Writing to other users

1.1.12 News

1.1.13 The manual

1.1.14 Computer-aided instruction

1.1.15 Games

1.2 Day-to-day use: files and common commands

1.3 More about files: directories

1.4 The shell

1.5 The rest of the UNIX system

Chapter 2

The File System

2.1 The basics of files

2.2 What’s in a file?

2.3 Directories and filenames

2.4 Permissions

2.5 Inodes

2.6 The directory hierarchy

2.7 Devices

Chapter 3

Using the Shell

The shell — the program that interprets your requests to run programs — is the most important program for most UNIX users; with the possible exception of your favorite text editor, you will spend more time working with the shell than any other program. In this chapter and in Chapter 5, we will spend a fair amount of time on the shell's capabilities. The main point we want to make is that you can accomplish a lot without much hard work, and certainly without resorting to programming in a conventional language like C, if you know how to use the shell.

We have divided our coverage of the shell into two chapters. This chapter goes one step beyond the necessities covered in Chapter 1 to some fancier but commonly used shell features, such as metacharacters, quoting, creating new commands, passing arguments to them, the use of shell variables, and some elementary control flow. These are topics you should know for your own use of the shell. The material in Chapter 5 is heavier going — it is intended for writing serious shell programs, ones that are bullet-proofed for use by others. The division between the two chapters is somewhat arbitrary, of course, so both should be read eventually.

3.1 Command line structure

To proceed, we need a slightly better understanding of just what a command is, and how it is interpreted by the shell. This section is more formal coverage, with new information, of the shell basics introduced in the first chapter.

The simplest command is a single *work*, usually naming a file for execution (later we will see some other types of commands):

```
$ who                              Execute the file /bin/who
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$
```

A command usually ends with a new line, but semicolon ; is also a *command terminator*:

```
$ date;
Wed Sep 28 09:07:15 EDT 1983
$ date; who
Wed Sep 28 09:07:23 EDT 1983
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$
```

Although semicolons can be used to terminate commands; as usual nothing happens until you type RETURN. Notice that the shell only prints one prompt after multiple commands, but except for the prompt,

```
$ date; who
```

is identical to typing the two commands on different lines. In particular, `who` doesn't run until `date` has finished.

Try sending the output of "date; who" through a pipe:

```
$ date; who | wc
Wed Sep 28 09: 08:48 EDT 1983
      2     10      60
$
```

This might not be what expected, because only the output of `who` goes to `wc`. Connecting `who` and `wc` with a pipe forms a single command, called a *pipleline*, that runs after `date`. The precedence of `|` is higher than that of '`:`' as the shell parses your command line.

Parentheses can be used to group commands:

```
$ (date; who) | tee save | wc
      3     16     89                       output from wc
$ cat save
Wed Sep 28 09:13:22 EDT 1983
you      tty2    Sep 28 07:51
jpl      tty4    Sep 28 08:32
$ wc <save
3 16 48
$
```

`tee` copies its input to the named file or files, as well as to its output, so `wc` receives the same data as if `tee` weren't in the pipeline.

Another command terminator is the ampersand `&`. It's exactly like the semicolon or newline, except that it tells the shell not to wait for the command to complete. Typically, `&` is used to run a long-running command "in the background" while you continue to type interactive commands:

```
$ long-running-command &
5273                            Process-id of long-running-command
$                               Prompt appears immediately
```

Given the ability to group commands, there are some more interesting uses of background process. The command `sleep` waits the specified number of seconds before exiting:

```
$ sleep 5
$                               Five seconds pass before prompt
$ (sleep 5; date) & date
5278
Wed Sep 28 09:18:20 EDT 1983    Ouput from second date
$ Wed Sep 28 09:18:25 EDT 1983  Prompt appears, then date 5 etc. later
```

The background process starts but immediately sleeps; meanwhile, the second `date` command prints the current time and shell prompts for new command. Five seconds later, the `sleep` exits and the first `date` prints the new time. It's hard to represent the passage of time on paper, so you should try this example. (Depending on how busy your machine is and other such details, the difference between the two times might not be exactly five seconds.) This is an easy way to run a command in the future; consider

```
$ (sleep 300; echo Tea is ready) &     Tea will be ready in 5 minutes
5291
$
```

as a handy reminder mechanism. (A *ctl*-g in the string to be `echo`ed will ring the terminal's bell when it's printed.) The parenthese are needed in these examples, since the precedence of `&` is higher than that of '`;`'.

The `&` terminator applies to the commands, and since pipelines are commands you don't need parentheses to run pipelines in the background:

```
$ pr file | lpr &
```

arranges to print the file on the line printer without making you wait for command to finish. Parenthesizing the pipeline has the same effect, but requires more typing:

```
$ (pr file | lpr) &                     Same as last example
```

Most programs accept *arguments* on the command line, such as `file` (an argument to `pr`) in the above example. Arguments are words, separated by blanks and tabs, that typically name files to be processed by the command, but they are strings that may be interpreted any way the programs see fit. For example, `pr` accepts names of the files to print, `echo` echoes its arguments without interpretation, and `grep`'s first argument specifies a test pattern to search for. And, of course, most programs also have options, indicated by arguments beginning with minus sign.

The various special characters interpreted by the shell, such as `<`, `>`, `|`, `;` and `&`, are *not* arguments to the programs the shell runs. They instead control how the shell runs them. For example,

```
$ echo Hello >junk
```

tells the shell to run `echo` with the single argument `Hello`, and place the output in the file `junk`. The string `>junk` is not an argument to `echo`; it is interpreted by the shell and never seen by `echo`. In fact, it need not be the last string in the command:

```
$ >junk echo hello
```

is identical, but less obvious.

Exercise 3-1 What are the differences among the following three commands?

```
$ cat file | pr
$ pr <file
$ pr file
```

(Over the years the redirection operator `<` has lost some ground to pipes; people seem to fine "`cat file|`" more natural than "`<file`".)

3.2 Metacharacters

The shell recognizes a number of other characters as special; the most commonly used is the asterisk `*` which tells the shell to search the directory for filenames in which any string of characters occurs in the position of the `*`. For example,

```
$ echo *
```

is a poor facsimile of `ls`. Something we didn't mention in Chapter 1 is that the filename-matching characters do not look at filenames beginning with dot, to avoid problems with the names '.' and ".." that are in every directory. The rule is: the filename-matching characters only match filenames beginning with a period if the period is explicitly supplied in the pattern. As usual, a judicious `echo` or two will clarify what happens:

```
$ ls
.profile
junk
temp
$ echo *
junk temp
$ echo .*
. .. .profile
$
```

Characters like * that have special properties are known as *metacharacters*. There are a lot of them: Table 3.1 is the complete list, although a few of them won't be discussed untile Chapter 5.

Given the number of shell metacharacters, there has to e some way to say to the shell, "leave it alone." The easiest and best way to protect special characters from being interpreted is to enclose them in single quote charaters:

```
$ echo '* * *'
* * *
$
```

It's also possible to use the double quotes `"..."`, but the shell actually peeks inside these quotes to look for `$`, `` `...` `` and \, so don't use `"..."` unless you intend some processing of the quoted string.

A third possibility is to put a backslash \ in front of *each* character that you want to protect from the shell, as in

```
$ echo \*\*\*
```

Although `\*\*\*` isn't much like English, the shell terminology for it is still a *word*, which is any single string the shell accepts as a unit, including blanks if they are quoted.

Quotes of one kind protect quotes of the other kind:

```
$ echo "don't do that!"
Don't do that
$
```

and they don't have to surround the whole argument:

```
$ echo x'*'y
x*y
$ echo '*'A'?'
*A?
$
```

Table 3.1: Shell Metacharacters

`>`	`prog >file direct standard output to file`
`>>`	`prog >>file append standard output to file`
`<`	`prog <file take standard input from file`
`|`	`p1|p2 connect standard output p1 to standard input p2`
`<<str`	`here document: standard input follows, up to next str on a line by itself`
`*`	`match any string of zero of more characters in filenames`
`?`	`match any single character in filenames`
`[ccc]`	`match any single character from ccc in filenames; ranges like 0-9 or a-z are legal`
`;`	`command terminator: p1;p2 does p1, then p2`
`&`	`like ; but doesn't wait for p1 to finish`
`` `...` ``	``run command(s) in ...; output replaces `...` ``
`(...)`	`run command(s) in ... in sub-shell`
`{...}`	`run command(s) in ... in current shell (rarely used)`
`$1, $2 etc.`	`$0...$9 replaced by arguments to shell file`
`$var`	`value of shell variable var`
`${var}`	`value of var; avoids confusion when concatenated with text; see also Table 5.3`
`\`	`\c take character c literally, \newline discarded`
`'...'`	`take ... literally`
`"..."`	``take ... literally after $, `...`, and \ interpreted``
`#`	`if # starts word, rest of line is a comment (not in 7th Ed.)`
`var=value`	`assign to variable var`
`p1 && p2`	`run p1; if successful, run p2`
`p1 || p2`	`run p1; if unsuccessful, run p2`

In this example, because the quotes are discarded after they've done their job, `echo` sees a single argument containing no quotes.

Quoted strings can contain new lines:

```
$ echo 'hello
> world'
hello
world
$
```

The string '> ' is *secondary prompt* printed by the shell when it expects you to type more input to complete a command. In this example the quote on the first line has to be balanced with another. The secondary prompt string is stored in the shell variable `PS2`, and can be modified to taste.

In all of these examples, the quoting of a metacharacter prevents the shell from trying to interpret it. The command

3.3 Creating new commands

3.4 Command arguments and arameters

3.5 Program output as arguments

3.6 Shell variables

3.7 More on I/O redirection

3.8 Looping in shell program

3.9 bundle: putting it all together

3.10 Why a programmable shell

Chapter 4

Filters

4.1 The `grep` family

4.2 Other filters

4.3 The stream editor sed

4.4 The awk pattern scanning and processing language

4.5 Good files and good filters

Chapter 5

Shell Programming

Chapter 6

Programming with Standard I/O

Chapter 7

UNIX System Calls

Chapter 8

Program Development

Chapter 9

Document preparation

9.1 The `ms` macro package

9.2 The `troff` level

9.3 The `tbl` and `eqn` preprocessors

9.4 The manual page

9.5 Other document preparation tools

Chapter 10

Epilog

Appendix A

Editor Summary

Appendix B

hoc Manual

Hoc - An Interactive Language For Floating Point Arithmetic

Brian Kernighan
Rob Pike

ABSTRACT

Hoc is a simple programmable interpreter for floating point expressions. It has C-style control flow, function definition and the usual numerical built-in functions such as cosine and logarithm.

1. Expressions

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns the value of its right operand to its left operand, and yields the value, so multiple assignments work. The expression grammar is:

```
    expr:   number
        |   variable
        |   ( expr )
        |   expr binop expr
        |   unop expr
        |   function ( arguments )
```

Numbers are floating point. The input format is that recognized by scanf(3): digits, decimal point, digits, e or E, signed exponent. At least one digit or a decimal point must be present; the other components are optional.

Variable names are formed from a letter followed by a string of letters and numbers. binop refers to binary operators such as addition or logical comparison; unop refers to the two negation operators, `!' (logical negation, `not') and `-' (arithmetic negation, sign change). Table 1 lists the operators.

Table 1: Operators, in decreasing order of precedence

^	exponentiation (FORTRAN **), right associative
! -	(unary) logical and arithmetic negation
* /	multiplication, division
+ -	addition, subtraction
> >=	relational operators: greater, greater or equal,
< <=	less, less or equal,
== !=	equal, not equal (all sameprecedence)
&&	logical AND (both operands always evaluated)
||	logical OR (bothoperands always evaluated)
=	assignment, right associative

Functions, as described later, may be defined by the user. Function arguments are expressions separated by commas. There are also a number of built-in functions, all of which take a single argument, described in Table 2.

Table 2: Built-in Functions

abs(x)	|x|, absolute value of x
atan(x)	arc tangent of x
cos(x)	cos(x), cosine of x
exp(x)	e^x, exponential of x
int(x)	integer part of x, truncated towards zero
log(x)	log(x), logarithm base e of x
log10(x)	log_10(x), logarithm base 10 of x
sin(x)	sin(x), sine of x
sqrt(x)	sqrt(x), x^(1/2)

Logical expressions have value 1.0 (true) and 0.0 (false). As in C, any non-zero value is taken to be true. As is always the case with floating point numbers, equality comparisons are inherently suspect.

Hoc also has a few built-in constants, shown in Table 3.

Table 3: Built-in Constants

DEG	57.29577951308232087680	180/pi, degrees per radian
E	2.71828182845904523536	e, base of natural logarithms
GAMMA	0.57721566490153286060	gamma, Euler-Mascheroni constant
PHI	1.61803398874989484820	(sqrt(5)+1)/2, the golden ratio
PI	3.14159265358979323846	pi, circular transcendental number

----

2. Statements and Control Flow

Hoc statements have the following grammar:

```
    stmt:       expr
        |       variable = expr
        |       procedure ( arglist )
        |       while ( expr ) stmt
        |       if ( expr ) stmt
        |       if ( expr ) stmt else stmt
        |       { stmtlist }
        |       print expr-list
        |       return optional-expr

    stmtlist:   (nothing)
        |       stmlist stmt
```

An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their value.

Note that semicolons are not special to hoc: statements are terminated by newlines. This causes some peculiar behavior. The following are legal statements:

```
    if (x < 0) print(y) else print(z)

    if (x < 0) {
        print(y)
    } else {
        print(z)
    }
```

In the second example, the braces are mandatory: the newline after the if would terminate the statement and produce a syntax error were the brace omitted.

The syntax and semantics of hoc control flow facilities are basically the same as in C. The while and if statements are just as in C, except there are no break or continue statements.

3. Input and Output: read and print

The input function read, like the other built-ins, takes a single argument. Unlike the built-ins, though, the argument is not an expression: it is the name of a variable. The next number (as defined above) is read from the standard input and assigned to the named variable. The return value of read is 1 (true) if a value was read, and 0 (false) if read encountered end of file or an error.

Output is generated with the print statement. The arguments to print are a comma-separated list of expressions and strings in double quotes, as in C. Newlines must be supplied; they are never provided automatically by print.

Note that read is a special built-in function, and therefore takes a single parenthesized argument, while print is a statement that takes a comma-separated, unparenthesized list:

```
    while (read(x)) {
        print "value is ", x, "\n"
    }
```

4. Functions and Procedures

Functions and procedures are distinct in hoc, although they are defined by the same mechanism. This distinction is simply for run-time error checking: it is an error for a procedure to return a value, and for a function not to return one.

The definition syntax is:

```
    function: func name() stmt

    procedure: proc name() stmt
```

name may be the name of any variable --- built-in functions are excluded. The definition, up to the opening brace or statement, must be on one line, as with the if statements above.

Unlike C, the body of a function or procedure may be any statement, not necessarily a compound (brace-enclosed) statement. Since semicolons have no meaning in hoc, a null procedure body is formed by an empty pair of braces.

Functions and procedures may take arguments, separated by commas, when invoked. Arguments are referred to as in the shell: refers to the third (1-indexed) argument. They are passed by value and within functions are semantically equivalent to variables. It is an error to refer to an argument numbered greater than the number of arguments passed to the routine. The error checking is done dynamically, however, so a routine may have variable numbers of arguments if initial arguments affect the number of arguments to be referenced (as in C's printf).

Functions and procedures may recurse, but the stack has limited depth (about a hundred calls). The following shows a hoc definition of Ackermann's function:

```
    $ hoc
    func ack() {
            if ($1 == 0) return $2+1
            if ($2 == 0) return ack($1-1, 1)
            return ack($1-1, ack($1, $2-1))
    }
    ack(3, 2)
        29
    ack(3, 3)
        61
    ack(3, 4)
    hoc: stack too deep near line 8
    ...
```

5. Examples

Stirling's formula:

```
    ...

    $ hoc
    func stirl() {
        return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
    }
    stirl(10)
            3628684.7
    stirl(20)
            2.4328818e+18
```

Factorial function, n!:

```
    func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
```

Ratio of factorial to Stirling approximation:

```
    i = 9
    while ((i = i+1) <= 20) {
            print i, "  ", fac(i)/stirl(i), "\n"
    }
    10 1.0000318
    11 1.0000265
    12 1.0000224
    13 1.0000192
    14 1.0000166
    15 1.0000146
    16 1.0000128
    17 1.0000114
    18 1.0000102
```

```
19 1.0000092
20 1.0000083
```

Appendix C

hoc Listing

```
***** hoc.y *************************************************************

%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
        Symbol  *sym;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
        int     narg;   /* number of arguments */
}
%token  <sym>   NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token  <sym>   FUNCTION PROCEDURE RETURN FUNC PROC READ
%token  <narg>  ARG
%type   <inst>  expr stmt asgn prlist stmtlist
%type   <inst>  cond while if begin end
%type   <sym>   procname
%type   <narg>  arglist
%right  '='
%left   OR
%left   AND
%left   GT GE LT LE EQ NE
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS NOT
%right  '^'
%%
list:     /* nothing */
        | list '\n'
        | list defn '\n'
        | list asgn '\n'  { code2(pop, STOP); return 1; }
        | list stmt '\n'  { code(STOP); return 1; }
        | list expr '\n'  { code2(print, STOP); return 1; }
        | list error '\n' { yyerrok; }
        ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
        | ARG '=' expr
            { defnonly("$"); code2(argassign,(Inst)$1); $$=$3;}
        ;
stmt:     expr  { code(pop); }
        | RETURN { defnonly("return"); code(procret); }
        | RETURN expr
                { defnonly("return"); $$=$2; code(funcret); }
        | PROCEDURE begin '(' arglist ')'
                { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
        | PRINT prlist  { $$ = $2; }
        | while cond stmt end {
                ($1)[1] = (Inst)$3;     /* body of loop */
                ($1)[2] = (Inst)$4; }   /* end, if cond fails */
        | if cond stmt end {    /* else-less if */
                ($1)[1] = (Inst)$3;     /* thenpart */
                ($1)[3] = (Inst)$4; }   /* end, if cond fails */
        | if cond stmt end ELSE stmt end {      /* if with else */
                ($1)[1] = (Inst)$3;     /* thenpart */
                ($1)[2] = (Inst)$6;     /* elsepart */
                ($1)[3] = (Inst)$7; }   /* end, if cond fails */
        | '{' stmtlist '}'      { $$ = $2; }
        ;
cond:     '(' expr ')'  { code(STOP); $$ = $2; }
        ;
while:    WHILE { $$ = code3(whilecode,STOP,STOP); }
        ;
if:       IF    { $$ = code(ifcode); code3(STOP,STOP,STOP); }
        ;
begin:    /* nothing */         { $$ = progp; }
        ;
end:      /* nothing */         { code(STOP); $$ = progp; }
        ;
stmtlist: /* nothing */         { $$ = progp; }
        | stmtlist '\n'
        | stmtlist stmt
        ;
expr:     NUMBER { $$ = code2(constpush, (Inst)$1); }
        | VAR    { $$ = code3(varpush, (Inst)$1, eval); }
        | ARG    { defnonly("$"); $$ = code2(arg, (Inst)$1); }
        | asgn
        | FUNCTION begin '(' arglist ')'
                { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
        | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
        | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
        | '(' expr ')'  { $$ = $2; }
        | expr '+' expr { code(add); }
        | expr '-' expr { code(sub); }
        | expr '*' expr { code(mul); }
        | expr '/' expr { code(div); }
        | expr '^' expr { code (power); }
        | '-' expr   %prec UNARYMINUS   { $$=$2; code(negate); }
        | expr GT expr  { code(gt); }
        | expr GE expr  { code(ge); }
        | expr LT expr  { code(lt); }
        | expr LE expr  { code(le); }
        | expr EQ expr  { code(eq); }
        | expr NE expr  { code(ne); }
        | expr AND expr { code(and); }
        | expr OR expr  { code(or); }
        | NOT expr      { $$ = $2; code(not); }
        ;
prlist:   expr                  { code(prexpr); }
        | STRING                { $$ = code2(prstr, (Inst)$1); }
        | prlist ',' expr       { code(prexpr); }
        | prlist ',' STRING     { code2(prstr, (Inst)$3); }
        ;
defn:     FUNC procname { $2->type=FUNCTION; indef=1; }
            '(' ')' stmt { code(procret); define($2); indef=0; }
        | PROC procname { $2->type=PROCEDURE; indef=1; }
            '(' ')' stmt { code(procret); define($2); indef=0; }
        ;
procname: VAR
        | FUNCTION
        | PROCEDURE
        ;
arglist:  /* nothing */         { $$ = 0; }
        | expr                  { $$ = 1; }
        | arglist ',' expr      { $$ = $1 + 1; }
        ;
%%
        /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int     indef;
char    *infile;        /* input file name */
FILE    *fin;           /* input file pointer */
char    **gargv;        /* global argument list */
int     gargc;
int c;  /* global for use by warning() */
yylex()         /* hoc6 */
{
        while ((c=getc(fin)) == ' ' || c == '\t')
                ;
        if (c == EOF)
                return 0;
        if (c == '.' || isdigit(c)) {   /* number */
                double d;
                ungetc(c, fin);
                fscanf(fin, "%lf", &d);
                yylval.sym = install("", NUMBER, d);
                return NUMBER;
        }
        if (isalpha(c)) {
                Symbol *s;
                char sbuf[100], *p = sbuf;
                do {
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("name too long", sbuf);
                        }
                        *p++ = c;
                } while ((c=getc(fin)) != EOF && isalnum(c));
                ungetc(c, fin);
                *p = '\0';
                if ((s=lookup(sbuf)) == 0)
                        s = install(sbuf, UNDEF, 0.0);
                yylval.sym = s;
                return s->type == UNDEF ? VAR : s->type;
        }
        if (c == '$') { /* argument? */
                int n = 0;
                while (isdigit(c=getc(fin)))
                        n = 10 * n + c - '0';
                ungetc(c, fin);
                if (n == 0)
                        execerror("strange $...", (char *)0);
                yylval.narg = n;
                return ARG;
        }
        if (c == '"') { /* quoted string */
                char sbuf[100], *p, *emalloc();
                for (p = sbuf; (c=getc(fin)) != '"'; p++) {
                        if (c == '\n' || c == EOF)
                                execerror("missing quote", "");
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("string too long", sbuf);
                        }
                        *p = backslash(c);
                }
                *p = 0;
                yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
                strcpy(yylval.sym, sbuf);
                return STRING;
        }
        switch (c) {
        case '>':       return follow('=', GE, GT);
        case '<':       return follow('=', LE, LT);
        case '=':       return follow('=', EQ, '=');
        case '!':       return follow('=', NE, NOT);
        case '|':       return follow('|', OR, '|');
        case '&':       return follow('&', AND, '&');
        case '\n':      lineno++; return '\n';
        default:        return c;
        }
}

backslash(c)    /* get next char with \'s interpreted */
        int c;
{
        char *index();  /* `strchr()' in some systems */
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if (c != '\\')
                return c;
        c = getc(fin);
        if (islower(c) && index(transtab, c))
                return index(transtab, c)[1];
        return c;
}

follow(expect, ifyes, ifno)     /* look ahead for >=, etc. */
{
        int c = getc(fin);

        if (c == expect)
                return ifyes;
        ungetc(c, fin);
        return ifno;
}

defnonly(s)     /* warn if illegal definition */
        char *s;
{
        if (!indef)
                execerror(s, "used outside definition");
}

yyerror(s)      /* report compile-time error */
        char *s;
{
        warning(s, (char *)0);
}

execerror(s, t) /* recover from run-time error */
        char *s, *t;
{
        warning(s, t);
        fseek(fin, 0L, 2);              /* flush rest of file */
        longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
        execerror("floating point exception", (char *) 0);
}

main(argc, argv)        /* hoc6 */
        char *argv[];
{
        int i, fpecatch();

        progname = argv[0];
        if (argc == 1) {        /* fake an argument list */
                static char *stdinonly[] = { "-" };

                gargv = stdinonly;
                gargc = 1;
        } else {
                gargv = argv+1;
                gargc = argc-1;
        }
        init();
        while (moreinput())
                run();
        return 0;
}

moreinput()
{
        if (gargc-- <= 0)
                return 0;
        if (fin && fin != stdin)
                fclose(fin);
        infile = *gargv++;
        lineno = 1;
        if (strcmp(infile, "-") == 0) {
                fin = stdin;
                infile = 0;
        } else if ((fin=fopen(infile, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n", progname, infile);
                return moreinput();
        }
        return 1;
}

run()   /* execute until EOF */
{
        setjmp(begin);
        signal(SIGFPE, fpecatch);
        for (initcode(); yyparse(); initcode())
                execute(progbase);
}

warning(s, t)   /* print warning message */
        char *s, *t;
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t)
                fprintf(stderr, " %s", t);
        if (infile)
                fprintf(stderr, " in %s", infile);
        fprintf(stderr, " near line %d\n", lineno);
        while (c != '\n' && c != EOF)
                c = getc(fin);  /* flush rest of input line */
        if (c == '\n')
                lineno++;
}


***** hoc.h ****************************************************************

typedef struct Symbol {  /* symbol table entry */
        char    *name;
        short   type;
        union {
                double  val;            /* VAR */
                double  (*ptr)();       /* BLTIN */
                int     (*defn)();      /* FUNCTION, PROCEDURE */
                char    *str;           /* STRING */
        } u;
        struct Symbol   *next;  /* to link to another */
} Symbol;
Symbol  *install(), *lookup();

typedef union Datum {   /* interpreter stack type */
        double  val;
        Symbol  *sym;
} Datum;
extern  Datum pop();
extern  eval(), add(), sub(), mul(), div(), negate(), power();

typedef int (*Inst)();
#define STOP    (Inst) 0

extern  Inst *progp, *progbase, prog[], *code();
extern  assign(), bltin(), varpush(), constpush(), print(), varread();
extern  prexpr(), prstr();
extern  gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern  ifcode(), whilecode(), call(), arg(), argassign();
extern  funcret(), procret();

***** symbol.c **************************************************************

#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0;  /* symbol table: linked list */

Symbol *lookup(s)       /* find s in symbol table */
        char *s;
{
        Symbol *sp;

        for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;
        return 0;       /* 0 ==> not found */
}

Symbol *install(s, t, d)  /* install s in symbol table */
        char *s;
        int t;
        double d;
{
        Symbol *sp;
        char *emalloc();
        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist; /* put at front of list */
        symlist = sp;
        return sp;
}

char *emalloc(n)        /* check return from malloc */
        unsigned n;
{
        char *p, *malloc();

        p = malloc(n);
        if (p == 0)
                execerror("out of memory", (char *) 0);
        return p;
}

***** code.c ****************************************************************

#include "hoc.h"
#include "y.tab.h"
#include <stdio.h>

#define NSTACK  256

static Datum stack[NSTACK];     /* the stack */
static Datum *stackp;           /* next free spot on stack */

#define NPROG   2000
Inst    prog[NPROG];    /* the machine */
Inst    *progp;         /* next free spot for code generation */
Inst    *pc;            /* program counter during execution */
Inst    *progbase = prog; /* start of current subprogram */
int     returning;      /* 1 if return stmt seen */

typedef struct Frame {  /* proc/func call stack frame */
        Symbol  *sp;    /* symbol table entry */
        Inst    *retpc; /* where to resume after return */
        Datum   *argn;  /* n-th argument on stack */
        int     nargs;  /* number of arguments */
} Frame;
#define NFRAME  100
Frame   frame[NFRAME];
Frame   *fp;            /* frame pointer */
```

```
initcode() {
        progp = progbase;
        stackp = stack;
        fp = frame;
        returning = 0;
}

push(d)
        Datum d;
{
        if (stackp >= &stack[NSTACK])
                execerror("stack too deep", (char *)0);
        *stackp++ = d;
}

Datum pop()
{
        if (stackp == stack)
                execerror("stack underflow", (char *)0);
        return *--stackp;
}

constpush()
{
        Datum d;
        d.val = ((Symbol *)*pc++)->u.val;
        push(d);
}

varpush()
{
        Datum d;
        d.sym = (Symbol *)(*pc++);
        push(d);
}

whilecode()
{
        Datum d;
        Inst *savepc = pc;

        execute(savepc+2);      /* condition */
        d = pop();
        while (d.val) {
                execute(*((Inst **)(savepc)));  /* body */
                if (returning)
                        break;
                execute(savepc+2);      /* condition */
                d = pop();
        }
        if (!returning)
                pc = *((Inst **)(savepc+1)); /* next stmt */
}

ifcode()
{
        Datum d;
        Inst *savepc = pc;      /* then part */

        execute(savepc+3);      /* condition */
        d = pop();
        if (d.val)
                execute(*((Inst **)(savepc)));
        else if (*((Inst **)(savepc+1))) /* else part? */
                execute(*((Inst **)(savepc+1)));
        if (!returning)
                pc = *((Inst **)(savepc+2)); /* next stmt */
}

define(sp)      /* put func/proc in symbol table */
        Symbol *sp;
{
        sp->u.defn = (Inst)progbase;    /* start of code */
        progbase = progp;       /* next code starts here */
}

call()          /* call a function */
{
        Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
                                      /* for function */
        if (fp++ >= &frame[NFRAME-1])
                execerror(sp->name, "call nested too deeply");
        fp->sp = sp;
        fp->nargs = (int)pc[1];
        fp->retpc = pc + 2;
        fp->argn = stackp - 1;  /* last argument */
        execute(sp->u.defn);
        returning = 0;
}

ret()           /* common return from func or proc */
{
        int i;
        for (i = 0; i < fp->nargs; i++)
                pop();  /* pop arguments */
        pc = (Inst *)fp->retpc;
        --fp;
        returning = 1;
}

funcret()       /* return from a function */
{
        Datum d;
        if (fp->sp->type == PROCEDURE)
                execerror(fp->sp->name, "(proc) returns value");
        d = pop();      /* preserve function return value */
        ret();
        push(d);
}

procret()       /* return from a procedure */
{
        if (fp->sp->type == FUNCTION)
                execerror(fp->sp->name,
                        "(func) returns no value");
        ret();
}

double *getarg()        /* return pointer to argument */
{
        int nargs = (int) *pc++;
        if (nargs > fp->nargs)
            execerror(fp->sp->name, "not enough arguments");
        return &fp->argn[nargs - fp->nargs].val;
}

arg()   /* push argument onto stack */
{
        Datum d;
        d.val = *getarg();
        push(d);
}

argassign()     /* store top of stack in argument */
{
        Datum d;
        d = pop();
        push(d);        /* leave value on stack */
        *getarg() = d.val;
}

bltin()
{
        Datum d;
        d = pop();
        d.val = (*(double (*)())*pc++)(d.val);
        push(d);
}

eval()          /* evaluate variable on stack */
{
        Datum d;
        d = pop();
        if (d.sym->type != VAR && d.sym->type != UNDEF)
                execerror("attempt to evaluate non-variable", d.sym->name);
        if (d.sym->type == UNDEF)
                execerror("undefined variable", d.sym->name);
        d.val = d.sym->u.val;
        push(d);
}

add()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val += d2.val;
        push(d1);
}

sub()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val -= d2.val;
        push(d1);
}

mul()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val *= d2.val;
        push(d1);
}

div()
{
        Datum d1, d2;
        d2 = pop();
        if (d2.val == 0.0)
                execerror("division by zero", (char *)0);
        d1 = pop();
        d1.val /= d2.val;
        push(d1);
}

negate()
{
        Datum d;
        d = pop();
        d.val = -d.val;
        push(d);
}

gt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val > d2.val);
        push(d1);
}

lt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val < d2.val);
        push(d1);
}

ge()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val >= d2.val);
        push(d1);
}

le()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val <= d2.val);
        push(d1);
}

eq()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val == d2.val);
        push(d1);
}

ne()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != d2.val);
        push(d1);
}

and()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
        push(d1);
}

or()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
        push(d1);
}

not()
{
        Datum d;
        d = pop();
        d.val = (double)(d.val == 0.0);
        push(d);
}
```

```
power()
{
        Datum d1, d2;
        extern double Pow();
        d2 = pop();
        d1 = pop();
        d1.val = Pow(d1.val, d2.val);
        push(d1);
}

assign()
{
        Datum d1, d2;
        d1 = pop();
        d2 = pop();
        if (d1.sym->type != VAR && d1.sym->type != UNDEF)
                execerror("assignment to non-variable",
                        d1.sym->name);
        d1.sym->u.val = d2.val;
        d1.sym->type = VAR;
        push(d2);
}

print() /* pop top value from stack, print it */
{
        Datum d;
        d = pop();
        printf("\t%.8g\n", d.val);
}

prexpr()        /* print numeric value */
{
        Datum d;
        d = pop();
        printf("%.8g ", d.val);
}

prstr()         /* print string value */
{
        printf("%s", (char *) *pc++);
}

varread()       /* read into variable */
{
        Datum d;
        extern FILE *fin;
        Symbol *var = (Symbol *) *pc++;
```

```
  Again:
        switch (fscanf(fin, "%lf", &var->u.val)) {
        case EOF:
                if (moreinput())
                        goto Again;
                d.val = var->u.val = 0.0;
                break;
        case 0:
                execerror("non-number read into", var->name);
                break;
        default:
                d.val = 1.0;
                break;
        }
        var->type = VAR;
        push(d);
}

Inst *code(f)   /* install one instruction or operand */
        Inst f;
{
        Inst *oprogp = progp;
        if (progp >= &prog[NPROG])
                execerror("program too big", (char *)0);
        *progp++ = f;
        return oprogp;
}

execute(p)
        Inst *p;
{
        for (pc = p; *pc != STOP && !returning; )
                (*((++pc)[-1]))();
}

***** init.c ****************************************************************

#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Sqrt(), Exp(), integer();

static struct {         /* Keywords */
        char    *name;
        int     kval;
} keywords[] = {
        "proc",         PROC,
        "func",         FUNC,
        "return",       RETURN,
        "if",           IF,
        "else",         ELSE,
        "while",        WHILE,
        "print",        PRINT,
        "read",         READ,
        0,              0,
};

static struct {         /* Constants */
        char *name;
        double cval;
} consts[] = {
        "PI",    3.14159265358979323846,
        "E",     2.71828182845904523536,
        "GAMMA", 0.57721566490153286060,  /* Euler */
        "DEG",  57.29577951308232087680,  /* deg/radian */
        "PHI",   1.61803398874989484820,  /* golden ratio */
        0,       0
};

static struct {         /* Built-ins */
        char *name;
        double  (*func)();
} builtins[] = {
        "sin",  sin,
        "cos",  cos,
        "atan", atan,
        "log",  Log,    /* checks range */
        "log10", Log10, /* checks range */
        "exp",  Exp,    /* checks range */
        "sqrt", Sqrt,   /* checks range */
        "int",  integer,
        "abs",  fabs,
        0,      0
};

init()  /* install constants and built-ins in table */
{
        int i;
        Symbol *s;
        for (i = 0; keywords[i].name; i++)
                install(keywords[i].name, keywords[i].kval, 0.0);
        for (i = 0; consts[i].name; i++)
                install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
                s = install(builtins[i].name, BLTIN, 0.0);
                s->u.ptr = builtins[i].func;
        }
}

***** math.c ****************************************************************

#include <math.h>
#include <errno.h>
extern  int     errno;
double  errcheck();

double Log(x)
        double x;
{
        return errcheck(log(x), "log");
}
double Log10(x)
        double x;
{
        return errcheck(log10(x), "log10");
}

double Sqrt(x)
        double x;
{
        return errcheck(sqrt(x), "sqrt");
}

double Exp(x)
        double x;
{
        return errcheck(exp(x), "exp");
}

double Pow(x, y)
        double x, y;
{
        return errcheck(pow(x,y), "exponentiation");
}

double integer(x)
        double x;
{
        return (double)(long)x;
}

double errcheck(d, s)   /* check result of library call */
        double d;
        char *s;
{
        if (errno == EDOM) {
                errno = 0;
                execerror(s, "argument out of domain");
        } else if (errno == ERANGE) {
                errno = 0;
                execerror(s, "result out of range");
        }
        return d;
}

***** makefile **********************************************************

CC = lcc
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc6:   $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -lm -o hoc6

hoc.o code.o init.o symbol.o:   hoc.h

code.o init.o symbol.o: x.tab.h

x.tab.h:        y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr:     hoc.y hoc.h code.c init.c math.c symbol.c
        @pr $?
        @touch pr

clean:
        rm -f $(OBJS) [xy].tab.[ch]
```