Objectives

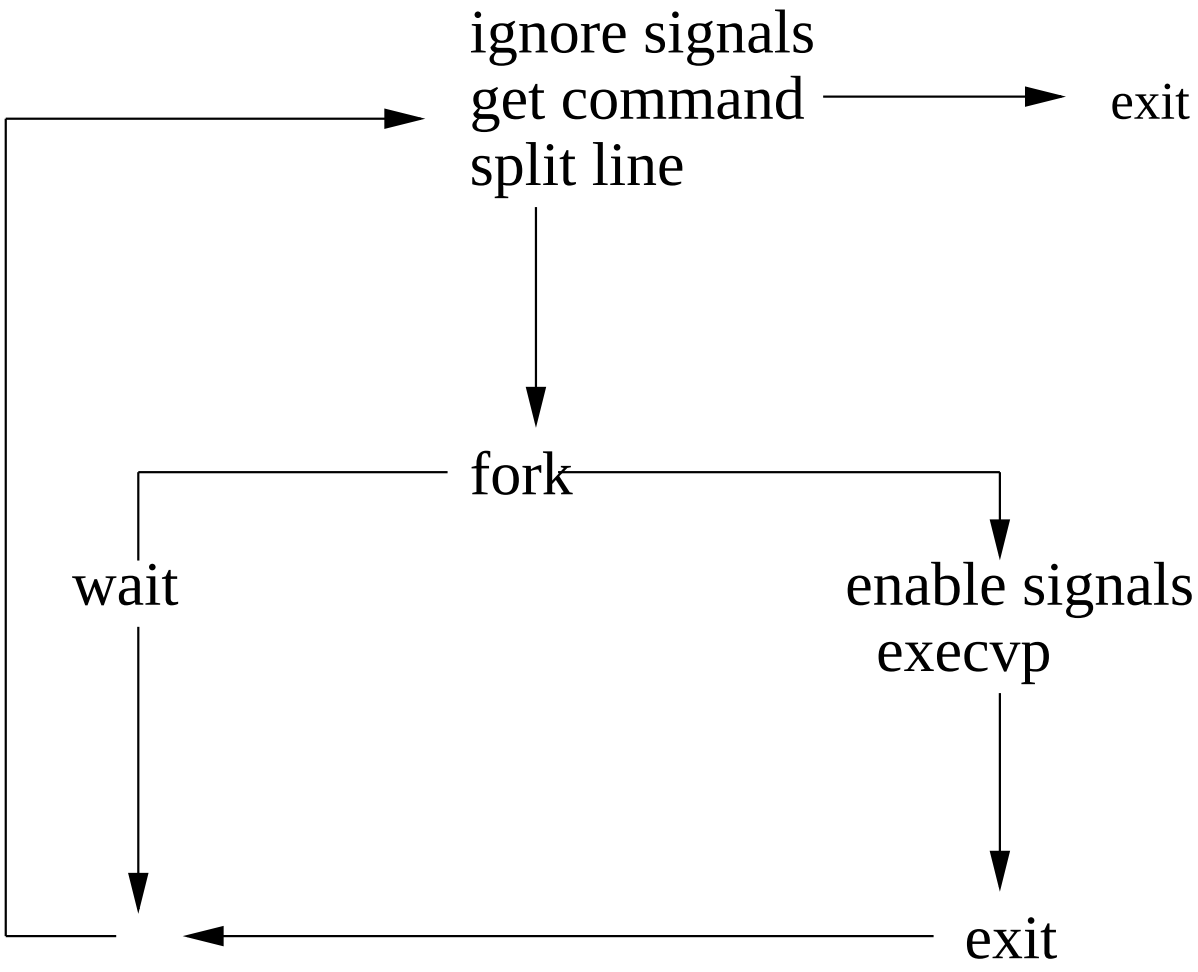
- Ideas and Skills
  - A Unix shell is a programming language
  - What is a shell script? How does a shell process a script?
  - How do shell control structures work? exit(0)=success
  - Shell variables: why and how
  - What is the environment? how does it work?
- System calls and functions
  - exit
  - getenv
- Commands
  - env

Components of a shell script

```
#!/bin/sh
# script2: a real program with variables, input,
#          and control flow

BOOK=$HOME/phonebook.data
echo find what name in phonebook
read NAME
if grep $NAME $BOOK > /tmp/pb.tmp
then
        echo Entries for $NAME
        cat /tmp/pb.tmp
else
        echo No entries for $NAME
fi
rm /tmp/pb.tmp
```

- variable
- user input
- control
- environment

Control flow of a shell



smsh1–Command line parsing

```
/**  smsh1.c  small-shell version 1
 **             first really useful version after prompting shell
 **             this one parses the command line into strings
 **             uses fork, exec, wait, and ignores signals
 **/
#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        "smsh.h"

#define DFL_PROMPT      "> "

int main()
{
        char    *cmdline, *prompt, **arglist;
        int     result;
        void    setup();

        prompt = DFL_PROMPT ;
        setup();

        while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
```

```
                if ( (arglist = splitline(cmdline)) != NULL  ){
                        result = execute(arglist);
                        freelist(arglist);
                }
                free(cmdline);
        }
        return 0;
}

void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
}

void fatal(char *s1, char *s2, int n)
{
        fprintf(stderr,"Error: %s,%s\n", s1, s2);
        exit(n);
}
```

Source code of execute.c

```
/* execute.c - code used by small shell to execute commands */

#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        <sys/wait.h>

int execute(char *argv[])
/*
 * purpose: run a program passing it arguments
 * returns: status returned via wait, or -1 on error
 *  errors: -1 on fork() or wait() errors

 */
{
        int     pid ;
        int     child_info = -1;

        if ( argv[0] == NULL )          /* nothing succeeds     */
                return 0;

        if ( (pid = fork())  == -1 )
```

```
            perror("fork");
        else if ( pid == 0 ){
            signal(SIGINT, SIG_DFL);  /*turn on the default signal handling*/
            signal(SIGQUIT, SIG_DFL);
            execvp(argv[0], argv);
            perror("cannot execute command");
            exit(1);
        }
        else {
            if ( wait(&child_info) == -1 )  /*wait for return of the child*/
                perror("wait");
        }
        return child_info;
}
```

Add if-then-fi statement

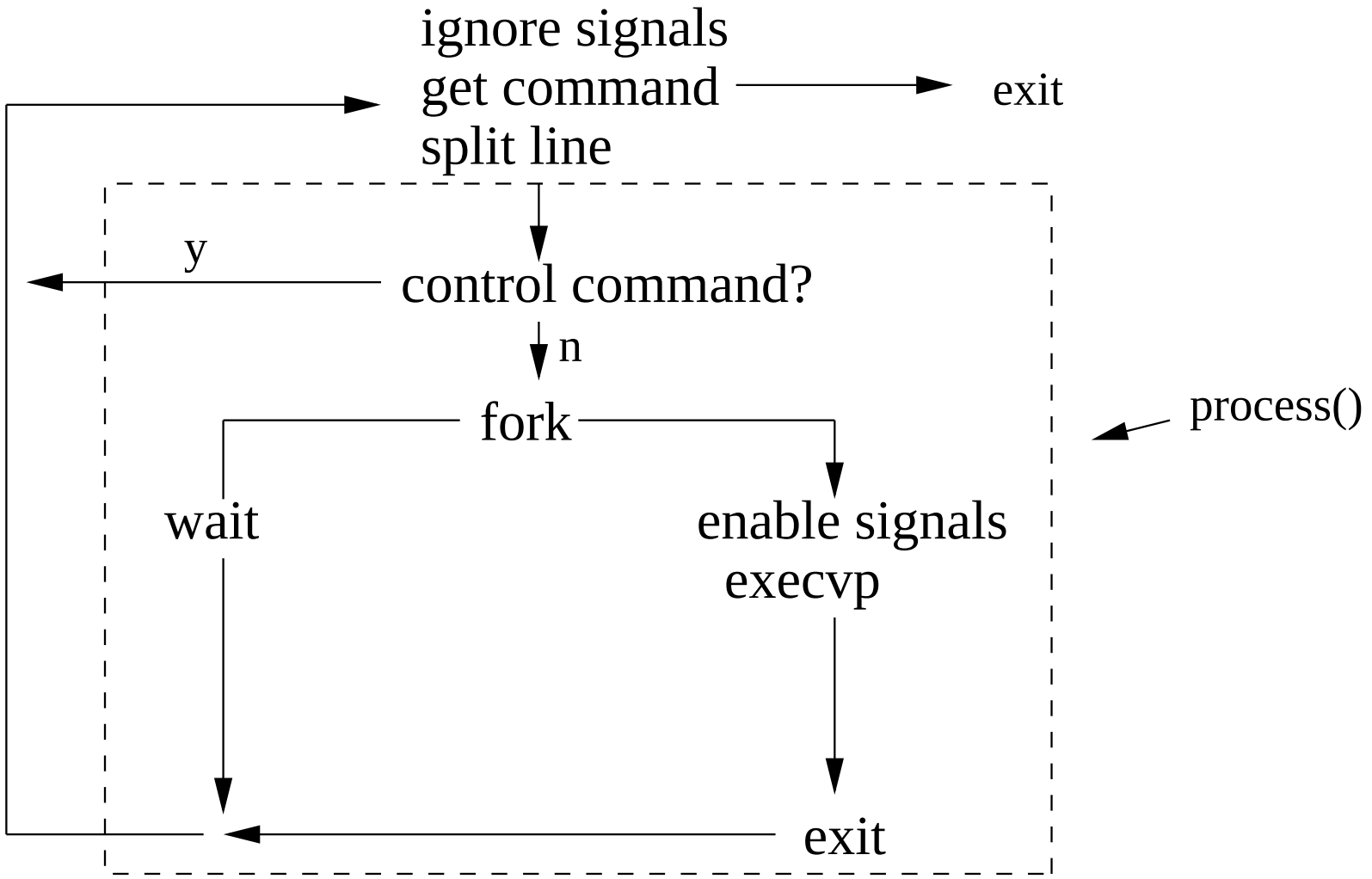
```
ls
who
if diff file1 file.bak
then
     echo no differences found, removing backup
     rm file1.bak
else
     echo backup differs, making it read-only
     chmod -w file1.bak
fi
date
```

We check the exit value of an application. If the exit value is 0, it means success. Otherwise, it indicates failure.

How *if* works

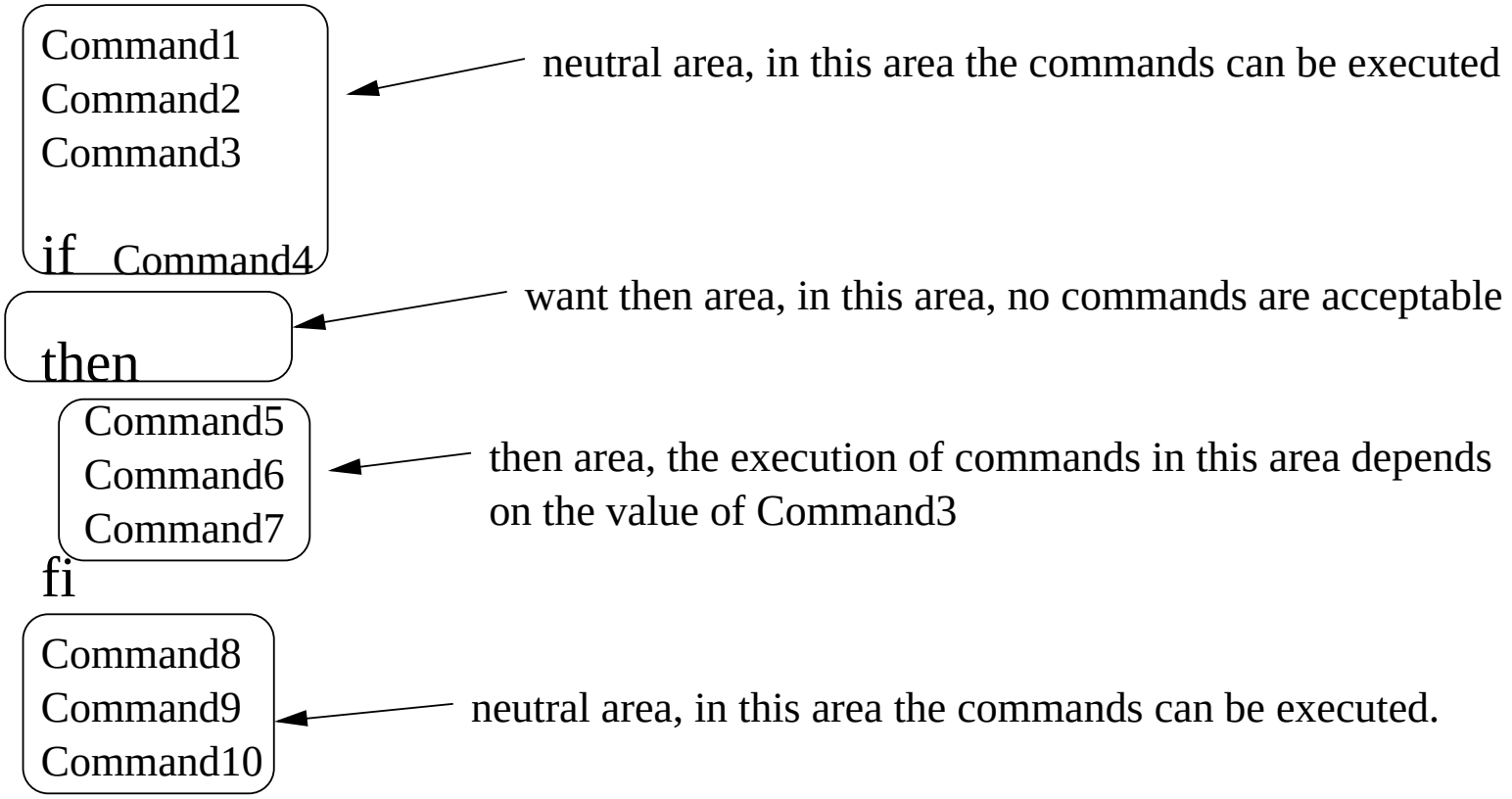
- The shell runs the command that follows the word *if*
- The shell checks the *exit* status of the command
- An *exit* status of 0 means *success*, nonzero means *failure*
- The shell executes commands after the *then* line if success
- The shell executes commands after the *else* line if failure
- The keyword *fi* marks the end of the *if* block.

Adding flow control commands to smsh



A wrapper function called process added

Adding flow control commands to smsh



Read the implementation

Read the source code: smsh2.c, process.c, controlflow.c

```
int process(char **args)
/*
 * purpose: process user command
 * returns: result of processing command
 * details: if a built-in then call appropriate function, if not execute()
 *  errors: arise from subroutines, handled there
 */
{
        int             rv = 0;

        if ( args[0] == NULL )
                rv = 0;
        else if ( is_control_command(args[0]) )
                rv = do_control_command(args);
        else if ( ok_to_execute() )
                rv = execute(args);
        return rv;
}
```

Read the implementation

```
int ok_to_execute()
/*
 * purpose: determine the shell should execute a command
 * returns: 1 for yes, 0 for no
 * details: if in THEN_BLOCK and if_result was SUCCESS then yes
 *          if in THEN_BLOCK and if_result was FAIL    then no
 *          if in WANT_THEN  then syntax error (sh is different)
 */
{
        int     rv = 1;         /* default is positive */

        if ( if_state == WANT_THEN ){
                syn_err("then expected");
                rv = 0;
        }
        else if ( if_state == THEN_BLOCK && if_result == SUCCESS )
                rv = 1;
        else if ( if_state == THEN_BLOCK && if_result == FAIL )
                rv = 0;
        return rv;
}
```

Read the implementation

```
int do_control_command(char **args)
/*
 * purpose: Process "if", "then", "fi" - change state or detect error
 * returns: 0 if ok, -1 for syntax error
 */
{
        char    *cmd = args[0];
        int     rv = -1;

        if( strcmp(cmd,"if")==0 ){
                if ( if_state != NEUTRAL )
                        rv = syn_err("if unexpected");
                else {
                        last_stat = process(args+1);
                        if_result = (last_stat == 0 ? SUCCESS : FAIL );
                        if_state = WANT_THEN;
                        rv = 0;
                }
        }
        else if ( strcmp(cmd,"then")==0 ){
                if ( if_state != WANT_THEN )
                        rv = syn_err("then unexpected");
                else {
```

```
                        if_state = THEN_BLOCK;
                        rv = 0;
                }
        }
        else if ( strcmp(cmd,"fi")==0 ){
                if ( if_state != THEN_BLOCK )
                        rv = syn_err("fi unexpected");
                else {
                        if_state = NEUTRAL;
                        rv = 0;
                }
        }
        else
                fatal("internal error processing:", cmd, 2);
        return rv;
}
```

Test smsh2

cc -o smsh2 smsh2.c process.c controlflow.c splitline.c execute.c

Shell variables

Example of using variables in a Shell script

```
$age=7                          #assigning a value
$echo $age                      #retrieving a value
7
$echo age                       #the $ is required
age
$echo $age + $age               #string operation
7+7
$read name                      #input from stdin
fido
$hello, $name, how are you      #can be interpolated
hello, fido, how are you
$ls > $name.$age                #used as part of a command
food: not found                 #no space allowed in assignment
$age=‘expr $age + $age‘         #numeric evaluation using expr
$echo $age
8
```

Shell variables and Environment variables

- Every UNIX process runs in a specific environment. An environment consists of a table of environment variables, each with an assigned value.
- The shell maintains a set of internal variables known as shell variables. These variables cause the shell to work in a particular way. Shell variables are local to the shell in which they are defined; they are not available to the parent or child shells. To move a local variable to an environment variable, use *export*.
- Bash does not keep two variable tables. It copies its environment variables into its local vaviable table.

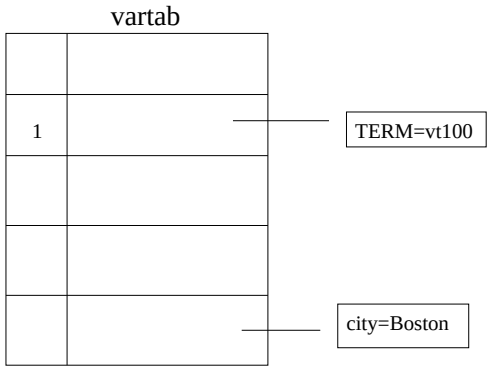
Shell variables: operations on variables

Operation	Syntax	Notes
assignment	var=value	no spaces
reference	$var	
delete	unset var	
stdin input	read var	also, read var1, var2
list vars	set	
make global	export var	

To list variable, use *set* command. To list environment variables, use *env* command.

Shell variables: representation and interface

```
struct var{
        char *str;      /*name=val string*/
        int global;     /*whether it is a global variable*/
}
static struct var tab[MAXVARS];
```



We also need the interface to this representation to support the built-in variable commands. Read varlib.c

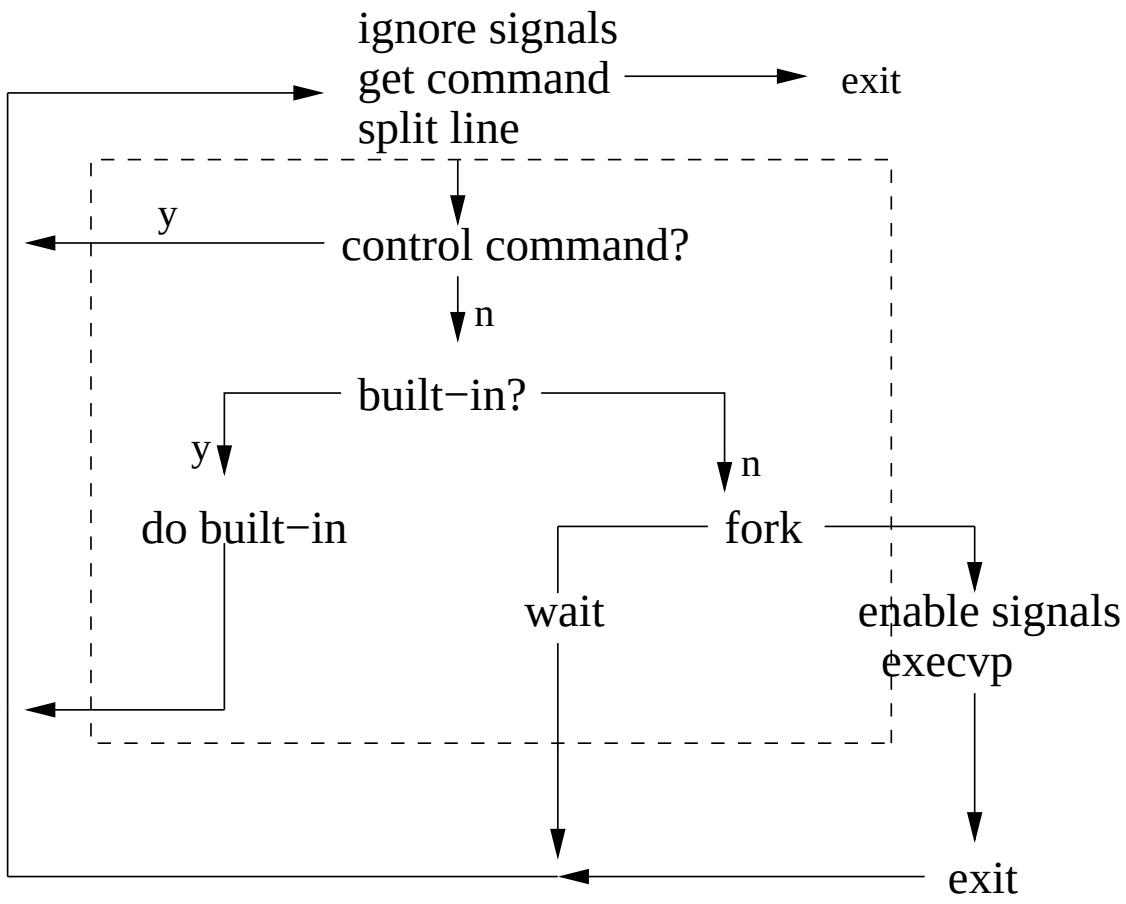
```
int VLstore( char *name, char *val )  add/updates variables
char * VLlookup( char *name )         look up a variable in the table
int VLexport( char *name )            marks a var for export
void VLlist()                         list all variables in the table
```

Shell variables: Add built-in variable commands

We would like to add the following built-in commands to the shell:

- assignment "=": price=12
- list variables: set
- export local variable to global variables: export

Therefore, before we fork a process to execute a command/application, we have to check whether it contains built-in commands for variables.


ignore signals
get command
exit
split line
y
control command?
n
built−in?
y
n
do built−in
fork
wait
enable signals
execvp
exit

```
int process(char **args)
/*
 * purpose: process user command
 * returns: result of processing command
 * details: if a built-in then call appropriate function, if not execute()
 *  errors: arise from subroutines, handled there
 */
{
        int             rv = 0;

        if ( args[0] == NULL )
                rv = 0;
        else if ( is_control_command(args[0]) )
                rv = do_control_command(args);
        else if ( ok_to_execute() )
                if ( !builtin_command(args,&rv) )
                        rv = execute(args);
        return rv;
}
```

builtin_command() in builtin.c

```
 int builtin_command(char **args, int *resultp)
/*
 * purpose: run a builtin command
 * returns: 1 if args[0] is builtin, 0 if not
 *   *resultp keeps the result of the built-in command.
 * details: test args[0] against all known builtins.  Call functions
 */
{
        int rv = 0;
        if ( strcmp(args[0],"set") == 0 ){           /* 'set' command? */
                VLlist();
                *resultp = 0;
                rv = 1;
        }
        else if ( strchr(args[0], '=') != NULL ){   /* assignment cmd */
                *resultp = assign(args[0]);
                if ( *resultp != -1 )               /* x-y=123 not ok */
                        rv = 1;
        }
        else if ( strcmp(args[0], "export") == 0 ){
                if ( args[1] != NULL && okname(args[1]) )
                        *resultp = VLexport(args[1]);
                else
```

```
                        *resultp = 1;
                rv = 1;
        }
        return rv;
}
```

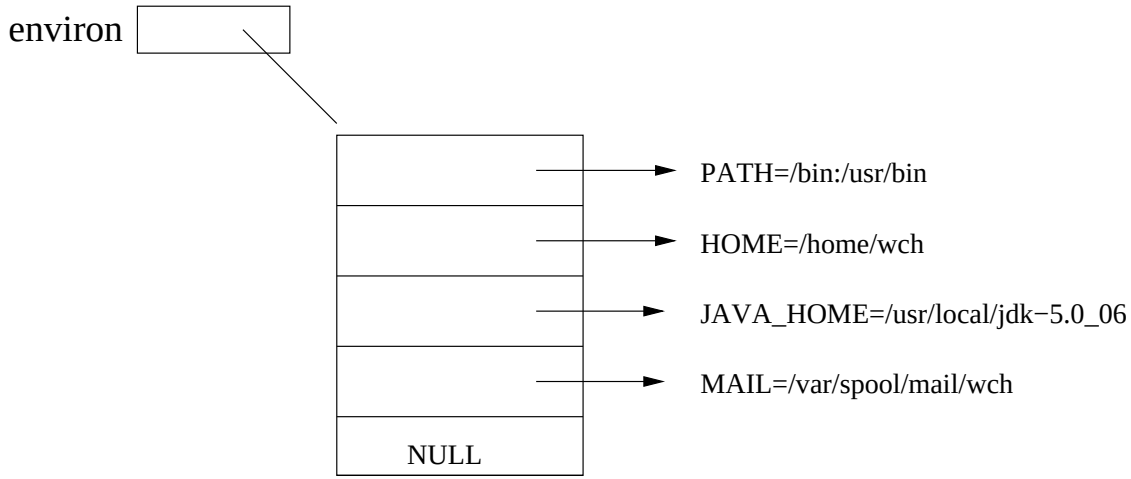
Shell variables

Compile and run the code:

```
cc -o smsh3 smsh2.c splitline.c execute.c \
  process2.c controlflow.c builtin.c varlib.c
```

Shell variables: How to use it?

Each process has a global variable *char ** environ*, which points to an array of strings containing the variables in the format of *var=value*

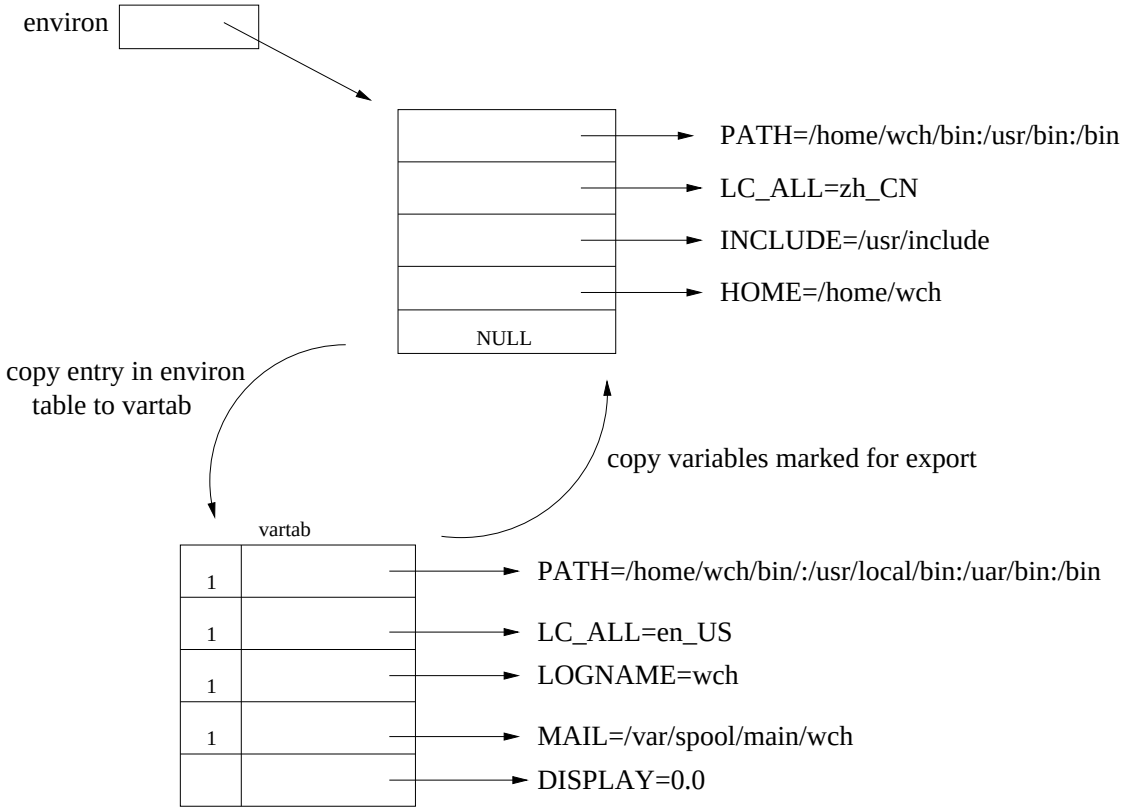


To read a value from environ table, use *char *getenv(const char *name)* function. This function is defined in *stdlib.h*

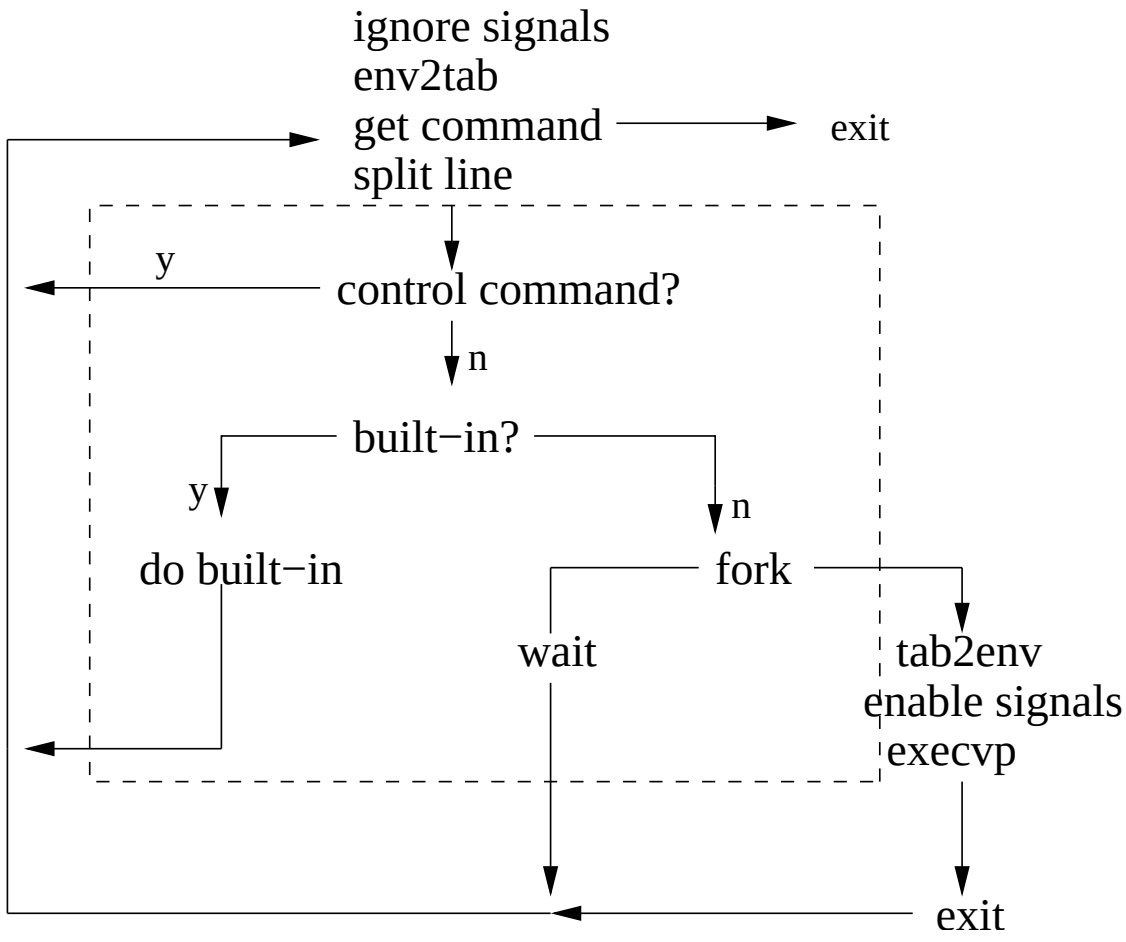
```
/*read and print enviroment variables*/
extern char     **environ;

main()
{
  int     i;

  for( i = 0 ; environ[i] ; i++ )
     printf("%s\n", environ[i] );
}
```

```
/*demo of changing environment variables*/
#include        <stdio.h>

extern char ** environ;

main()
{
        char *table[3];

        table[0] = "TERM=vt100";
        table[1] = "HOME=/on/the/range";
        table[2] = 0;

        environ = table;

        execlp("env", "env", NULL);
}
```

Handle environment variables



Handle environment variables

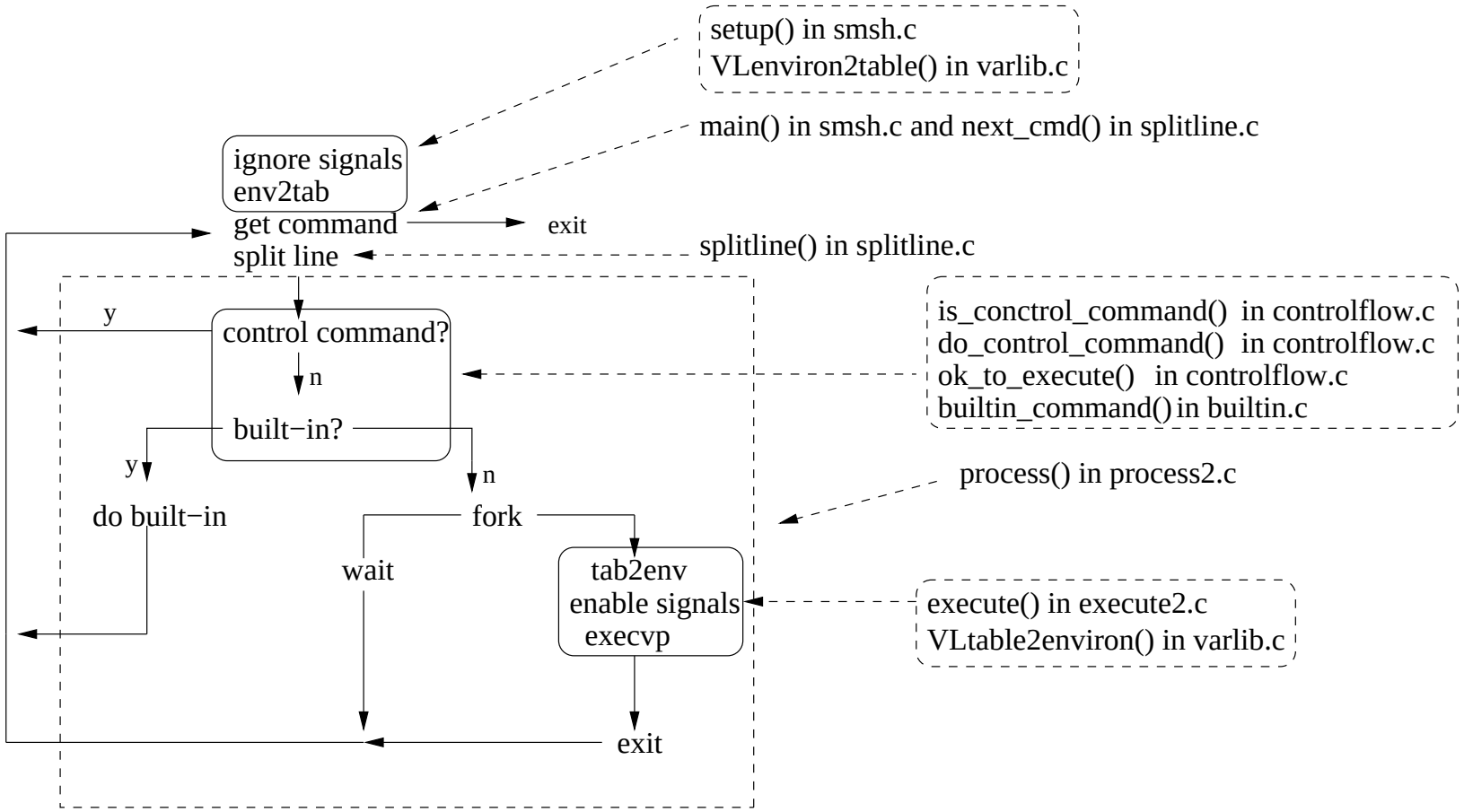


Handle environment variables-Changes in source code

```
void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
        extern char **environ;

        VLenviron2table(environ);
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
}
```

```
int execute(char *argv[])
/*
 * purpose: run a program passing it arguments
 * returns: status returned via wait, or -1 on error
 *  errors: -1 on fork() or wait() errors
 */
{
        extern char **environ;
        int     pid ;
        int     child_info = -1;

        if ( argv[0] == NULL )          /* nothing succeeds*/
                return 0;

        if ( (pid = fork())  == -1 )
                perror("fork");
        else if ( pid == 0 ){
                environ = VLtable2environ();
                signal(SIGINT, SIG_DFL);
                signal(SIGQUIT, SIG_DFL);
                execvp(argv[0], argv);
                perror("cannot execute command");
                exit(1);
        }
        else {
                if ( wait(&child_info) == -1 )
                        perror("wait");
        }
        return child_info;
}
```

Put all source code together



Test the shell

Compile and test the shell we have.

Status of the shell

feature	supports	needs
commands	runs programs	
variables	=,set	read, $var substitution
if	if then	else
environ	all	
exit		exit
cd		cd
$\langle$, $\rangle$, |	none	all