IEEE P1003.2 Draft 11.2 – September 1991

presumably). The implementation must ensure that all effects specified by POSIX.2 resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, etc.) are identical to those resulting from the invocation of an executable file.

Example : Consider three versions of the `ls` utility:

— The application includes a shell function named `ls`.

— The user writes her own utility named `ls` and puts it in `/hsa/bin`.

— The example implementation provides `ls` as a regular shell built-in that will be invoked (either by the shell or directly by *exec*) when the path search reaches the directory `/posix/bin`.

If `PATH=/posix/bin`, various invocations yield different versions of `ls`:

Invocation	Version of `ls`
`ls` (from within application script)	(1) function
`command ls` (from within application script)	(3) built-in
`ls` (from within makefile called by application)	(3) built-in
`system("ls")`	(3) built-in
`PATH="/hsa/bin:$PATH" ls`	(2) user's version

After the *execve*() failure described, the shell normally executes the file as a shell script. Some implementations, however, attempt to detect whether the file is actually a script and not an executable from some other architecture. The method used by the KornShell is allowed by the text that indicates nontext files may be bypassed.

3.9.2 Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator `|`. The standard output of all but the last command shall be connected to the standard input of the next command.

The format for a pipeline is:

`[!]` *command1* `[ |` *command2* `...]`

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see 3.7).

If the pipeline is not in the background (see 3.9.3.1), the shell shall wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

Exit Status

If the reserved word ! does not precede the pipeline, the exit status shall be the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status shall be 1; if the last command returns greater than zero, the exit status is zero.

3.9.2.1 Pipelines Rationale. *(This subclause is not a part of P1003.2)*

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of `command1` to the standard input of `command2`.

The reserved word ! was added to allow more flexible testing using AND and OR lists.

It was suggested that it would be better to return a nonzero value if any command in the pipeline terminates with nonzero status (perhaps the bitwise OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause application breakage if changed. An example of historical (and POSIX.2) behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5                1
$ echo $?                           1
0                                   1
```

3.9.3 Lists

An *AND-OR-list* is a sequence of one or more pipelines separated by the operators

```
&&    ||
```

A *list* is a sequence of one or more AND-OR-lists separated by the operators

```
;    &
```

and optionally terminated by

```
;    &    <newline>
```

The operators `&&` and `||` shall have equal precedence and shall be evaluated from beginning to end.

A `;` or `<newline>` terminator shall cause the preceding AND-OR-list to be executed sequentially; an & shall cause asynchronous execution of the preceding AND-OR-list.

The term *compound-list* is derived from the grammar in 3.10; it is equivalent to a sequence of *lists,* separated by `<newline>`s, that can be preceded or followed by an arbitrary number of `<newline>`s.

3.9.3.0.1 Lists Rationale. *(This subclause is not a part of P1003.2)*

The equal precedence of && and || is historical practice. The developers of the standard evaluated the model used more frequently in high level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but could not in the end allow existing scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the { } or ( ) groupings that are required historically. There are some disadvantages to these groupings:

— The ( ) can be expensive, as they spawn other processes on some systems. This performance concern is primarily an implementation issue.

— The { } braces are not operators (they are reserved words) and require a trailing space after each {, and a semicolon before each }. Most programmers (and certainly interactive users) have avoided braces as grouping constructs because of the irritating syntax required. Braces were not changed to operators because that would generate compatibility issues even greater than the precedence question; braces appear outside the context of a keyword in many shell scripts.

An example reiterates the precedence of the lists as they associate from beginning 1
to end. Both of the following commands write solely `bar` to standard output: 1

```
false && echo foo || echo bar    1
true || echo foo && echo bar     1
```

The following is an example that illustrates `<newline>`s in compound-lists:

```
while
        # a couple of newlines

        # a list
        date && who || ls; cat file
        # a couple of newlines

        # another list
        wc file > output & true

do
        # 2 lists
        ls
        cat file
done
```

3.9.3.1 Asynchronous Lists

If a command is terminated by the control operator ampersand (&), the shell shall execute the command asynchronously in a subshell. This means that the shell shall not wait for the command to finish before executing the next command.

The format for running a command in background is:

command1 `&` [*command2* `&` . . .]

The standard input for an asynchronous list, before any explicit redirections are performed, shall be considered to be assigned to a file that has the same properties as `/dev/null`. If it is an interactive shell, this need not happen. In all cases, explicit redirection of standard input shall override this activity.

When an element of an asynchronous list (the portion of the list ended by an ampersand, such as *command1* , above) is started by the shell, the process ID of the last command in the asynchronous list element shall become known in the current shell execution environment; see 3.12. This process ID shall remain known until:

- — The command terminates and the application waits for the process ID, or
- — Another asynchronous list is invoked before `$!` (corresponding to the previous asynchronous list) is expanded in the current execution environment.

The implementation need not retain more than the {CHILD_MAX} most recent entries in its list of known process IDs in the current shell execution environment.

Exit Status

The exit status of an asynchronous list shall be zero.

3.9.3.1.1 Asynchronous Lists Rationale. *(This subclause is not a part of P1003.2)*

The grammar treats a construct such as

```
foo & bar & bam &
```

as one ‘‘asynchronous list,’’ but since the status of each element is tracked by the shell, the term ‘‘element of an asynchronous list’’ was introduced to identify just one of the `foo`, `bar`, `bam` portions of the overall list.

Unless the implementation has an internal limit, such as {CHILD_MAX}, on the retained process IDs, it would require unbounded memory for the following example:

```
while true
do      foo & echo $!
done
```

The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in 3.11.

Since the connection of the input to the equivalent of `/dev/null` is considered to occur before redirections, the following script would produce no output:

```
exec < /etc/passwd
cat <&0 &
wait
```

3.9.3.2 Sequential Lists

Commands that are separated by a semicolon (`;`) shall be executed sequentially.

The format for executing commands sequentially is:

command1 `[;` *command2* `]`. . .

Each command shall be expanded and executed in the order specified.

Exit Status

The exit status of a sequential list shall be the exit status of the last command in the list.

3.9.3.3 AND Lists

The control operator `&&` shall denote an AND list. The format is:

command1 `[ &&` *command2* `]`. . .

First *command1* is executed. If its exit status is zero, *command2* is executed, and so on until a command has a nonzero exit status or there are no more commands left to execute. The commands shall be expanded only if they are executed.

Exit Status

The exit status of an AND list shall be the exit status of the last command that is executed in the list.

3.9.3.4 OR Lists

The control operator `||` shall denote an OR List. The format is:

command1 `[ ||` *command2* `]`. . .

First, *command1* is executed. If its exit status is nonzero, *command2* is executed, and so on until a command has a zero exit status or there are no more commands left to execute.

Exit Status

The exit status of an OR list shall be the exit status of the last command that is executed in the list.

3.9.4 Compound Commands

The shell has several programming constructs that are *compound commands*, which provide control flow for commands. Each of these compound commands has a reserved word or control operator at the beginning, and a corresponding terminator reserved word or operator at the end. In addition, each can be followed by redirections on the same line as the terminator. Each redirection shall apply to all the commands within the compound command that do not explicitly override that redirection.

3.9.4.1 Grouping Commands

The format for grouping commands is as follows:

`(`*compound-list* `)`	Execute *compound-list* in a subshell environment; see 3.12. Variable assignments and built-in commands that affect the environment shall not remain in effect after the list finishes.
`{` *compound-list* `;}`	Execute *compound-list* in the current process environment.

Exit Status

The exit status of a grouping command shall be the exit status of *list*.

3.9.4.1.1 Grouping Commands Rationale. *(This subclause is not a part of P1003.2)*

The semicolon shown in `{` *compound-list* `;}` is an example of a control operator delimiting the `}` reserved word. Other delimiters are possible, as shown in 3.10; `<newline>` is frequently used.

A proposal was made to use the <do-done> construct in all cases where command grouping performed in the current process environment is performed, identifying it as a construct for the grouping commands, as well as for shell functions. This was not included because the shell already has a grouping construct for this purpose (`{ }`), and changing it would have been counter-productive.

3.9.4.2 `for` Loop

The `for` loop shall execute a sequence of commands for each member in a list of *items*. The `for` loop requires that the *reserved words* `do` and `done` be used to delimit the sequence of commands.

The format for the `for` loop is as follows.

```
for name [ in word ... ]
do
        compound-list
done
```

First, the list of words following `in` shall be expanded to generate a list of items. Then, the variable *name* shall be set to each item, in turn, and the *compound-list* executed each time. If no items result from the expansion, the *compound-list* shall not be executed. Omitting

```
in word ...
```

is equivalent to

```
in "$@"
```

Exit Status

The exit status of a `for` command shall be the exit status of the last command that executes. If there are no items, the exit status shall be zero.

3.9.4.2.1 `for` Loop Rationale. *(This subclause is not a part of P1003.2)*

The format is shown with generous usage of `<newline>`s. See the grammar in 3.10 for a precise description of where `<newline>`s and semicolons can be interchanged.

Some historical implementations support `{` and `}` as substitutes for `do` and `done`. The working group chose to omit them, even as an obsolescent feature. (Note that
these substitutes were only for the `for` command; the `while` and `until` com- 1
mands could not use them historically, because they are followed by compound- 1
lists that may contain `{...}` grouping commands themselves 1

The reserved word pair `do` ... `done` was selected rather than `do` ... `od` (which would have matched the spirit of `if` ... `fi` and `case` ... `esac`) because `od` is a commonly-used utility name and this would have been an unacceptable choice.

3.9.4.3 `case` Conditional Construct

The conditional construct `case` shall execute the *compound-list* corresponding to the first one of several *patterns* (see 3.13) that is matched by the string resulting from the tilde expansion, parameter expansion, command substitution, and arithmetic expansion and quote removal of the given word. The reserved word `in` shall denote the beginning of the patterns to be matched. Multiple patterns with the same *compound-list* are delimited by the `|` symbol. The control operator `)` terminates a list of patterns corresponding to a given action. The *compound-list* for each list of patterns is terminated with `;;`. The `case` construct terminates with the reserved word `esac` (`case` reversed).

The format for the `case` construct is as follows.

```
case word in
     [(]pattern1 )            compound-list ;;          2
     [(]pattern2 |pattern3 )  compound-list ;;          2
     ...
esac
```

The `;;` is optional for the last *compound-list*.

Each pattern in a pattern list shall be expanded and compared against the expansion of *word*. After the first match, no more patterns shall be expanded, and the *compound-list* shall be executed. The order of expansion and comparing of patterns in a multiple pattern list is unspecified.

Exit Status

The exit status of `case` is zero if no patterns are matched. Otherwise, the exit status shall be the exit status of the last command executed in the *compound-list*.

3.9.4.3.1 `case` Conditional Construct Rationale. *(This subclause is not a part of P1003.2)*

An optional open-parenthesis before *pattern* was added to allow numerous historical KornShell scripts to conform. At one time, using the leading parenthesis was required if the `case` statement were to be embedded within a `$( )` command substitution; this is no longer the case with the POSIX shell. Nevertheless, many existing scripts use the open-parenthesis, if only because it makes matching-parenthesis searching easier in `vi` and other editors. This is a relatively simple implementation change that is fully upward compatible for all scripts.

Consideration was given to requiring `break` inside the *compound-list* to prevent falling through to the next pattern action list. This was rejected as being nonexisting practice. An interesting undocumented feature of the KornShell is that using `;&` instead of `;;` as a terminator causes the exact opposite behavior—the flow of control continues with the next *compound-list*.

The pattern `"*"`, given as the last pattern in a `case` construct, is equivalent to the default case in a C-language `switch` statement

The grammar shows that reserved words can be used as patterns, even if one is the first word on a line. Obviously, the reserved word `esac` cannot be used in this manner.

3.9.4.4 `if` Conditional Construct

The `if` command shall execute a *compound-list* and use its exit status to determine whether to execute another *compound-list*.

The format for the `if` construct is as follows.

```
if compound-list
then
        compound-list
[elif compound-list
then
        compound-list ]...
[else
        compound-list ]
fi
```

The `if` *compound-list* is executed; if its exit status is zero, the `then` *compound-list* is executed and the command shall complete. Otherwise, each `elif` *compound-list* is executed, in turn, and if its exit status is zero, the `then` *compound-list* is executed and the command shall complete. Otherwise, the `else` *compound-list* is executed.

Exit Status

The exit status of the `if` command shall be the exit status of the `then` or `else` *compound-list* that was executed, or zero, if none was executed.

3.9.4.4.1 `if` Conditional Construct Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.4.5 `while` Loop

The `while` loop continuously shall execute one *compound-list* as long as another *compound-list* has a zero exit status.

The format of the `while` loop is as follows

```
while compound-list-1
do
        compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a nonzero exit status, the `while` command shall complete. Otherwise, the *compound-list-2* shall be executed, and the process shall repeat.

Exit Status

The exit status of the `while` loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

3.9.4.5.1 `while` Loop Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.4.6 `until` Loop

The `until` loop continuously shall execute one *compound-list* as long as another *compound-list* has a nonzero exit status.

The format of the `until` loop is as follows

```
until compound-list-1
do
        compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a zero exit status, the `until` command shall complete. Otherwise, the *compound-list-2* shall be executed, and the process shall repeat.

Exit Status

The exit status of the `until` loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

3.9.4.6.1 `until` Loop Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.5 Function Definition Command

A function is a user-defined name that is used as a simple command to call a compound command with new positional parameters. A function is defined with a *function definition command*.

The format of a function definition command is as follows:

fname `()` *compound-command* **[** *io-redirect* . . . **]**

The function is named *fname*; it shall be a name (see 3.1.5). An implementation 1
may allow other characters in a function name as an extension. The implementa- 1
tion shall maintain separate namespaces for functions and variables.

The argument *compound-command* represents a compound command, as described in 3.9.4.

When the function is declared, none of the expansions in 3.6 shall be performed on the text in *compound-command* or *io-redirect*; all expansions shall be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* shall be performed during the execution of the function itself, not the function definition. See 3.8.1 for the consequences of failures of these operations on interactive and noninteractive shells.

When a function is executed, it shall have the syntax-error and variable-assignment properties described for special built-in utilities, in the enumerated list at the beginning of 3.14.

The *compound-command* shall be executed whenever the function name is specified as the name of a simple command (see 3.9.1.1). The operands to the command temporarily shall become the positional parameters during the execution of the *compound-command*; the special parameter `#` shall also be changed to reflect the number of operands. The special parameter `0` shall be unchanged. When the function completes, the values of the positional parameters and the special parameter `#` shall be restored to the values they had before the function was executed. If the special built-in `return` is executed in the *compound-command*, the function shall complete and execution shall resume with the next command after the function call.

Exit Status

The exit status of a function definition shall be zero if the function was declared successfully; otherwise, it shall be greater than zero. The exit status of a function invocation shall be the exit status of the last command executed by the function.

3.9.5.1 Function Definition Command Rationale *(This subclause is not a part of P1003.2)*

The description of functions in Draft 8 was based on the notion that functions should behave like miniature shell scripts; that is, except for sharing variables, most elements of an execution environment should behave as if it were a new execution environment, and changes to these should be local to the function. For example, traps and options should be reset on entry to the function, and any changes to them don't affect the traps or options of the caller. There were numerous objections to this basic idea, and the opponents asserted that functions were intended to be a convenient mechanism for grouping commonly executed commands that were to be executed in the current execution environment, similar to the execution of the `dot` special built-in.

Opponents also pointed out that the functions described in Draft 8 did not scope everything a new shell script would anyway, such as the current working directory, or umask, but instead picked a few select properties. The basic argument was that if one wanted scoping of the execution environment, the mechanism already exists: put the commands in a new shell script and call it. All traditional shells that implemented functions, other than the KornShell, have implemented functions that operate in the current execution environment. Because of this, Draft 9 removed any local scoping of traps or options. Local variables within a function were considered and included in Draft 9 (controlled by the special built-in `local`), but were removed because they do not fit the simple model developed for the scoping of functions and there was some opposition to adding yet another new special built-in from outside existing practice. Implementations should reserve the identifier `local` (as well as `typeset`, as used in the KornShell) in case this local variable mechanism is adopted in a future version of POSIX. 2.

A separate issue from the execution environment of a function is the availability of that function to child shells. A few objectors, including the author of the original Version 7 UNIX system shell, maintained that just as a variable can be shared with child shells by exporting it, so should a function—and so this capability has been added to the standard. In previous drafts, the `export` command therefore had a `-f` flag for exporting functions. Functions that were exported were to be put into the environment as *name* `()=`*value* pairs, and upon invocation, the shell would scan the environment for these, and automatically define these functions. This facility received a lot of balloting opposition and was removed from Draft 11. Some of the arguments against exportable functions were:

— There was little existing practice. The Ninth Edition shell provided them, but there was controversy over how well it worked.

— There are numerous security problems associated with functions appearing in a script's environment and overriding standard utilities or the application's own utilities.

— There was controversy over requiring `make` to import functions, where it has historically used an *exec* function for many of its command line executions.

— Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different than variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this insight might be lost on the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in POSIX.2 match those in System V. The KornShell has two methods of defining functions:

`function` *fname* `{` *compound-list* `}`

and

fname `() {` *compound-list* `}`

The latter uses the same definition as POSIX.2, but differs in semantics, as described previously. A future edition of the KornShell is planned to align the latter syntax with POSIX and keep the former as-is.

The name space for functions is limited to that of a *name* because of historical 1
practice. Complications in defining the syntactic rules for the function definition 1
command and in dealing with known extensions such as the KornShell's `@()` 1
prevented the name space from being widened to a *word*, as requested by some 1
balloters. Using functions to support synonyms such as the C-shell's `!!` and `%` is 1
thus disallowed to portable applications, but acceptable as an extension. For 1
interactive users, the aliasing facilities in the UPE should be adequate for this 1
purpose. It is recognized that the name space for utilities in the file system is 1
wider than that currently supported for functions, if the portable filename charac- 1
ter set guidelines are ignored, but it did not seem useful to mandate extensions in 1
systems for so little benefit to portable applications. 1

The `()` in the function definition command consists of two operators. Therefore, intermixing `<blank>`s with the *fname*, `(`, and `)` is allowed, but unnecessary.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls -l
#
[ X$i = Xyes ] && foo() {
        ls -l
}
```

3.10 Shell Grammar

The following grammar describes the Shell Command Language. Any discrepancies found between this grammar and the preceding description shall be resolved in favor of this clause.

3.10.1 Shell Grammar Lexical Conventions

The input language to the shell must be first recognized at the character level. The resulting tokens shall be classified by their immediate context according to the following rules (applied in order). These rules are used to determine what a ‘‘token’’ that is subject to parsing at the token level is. The rules for token recognition in 3.3 shall apply.

(1) A `<newline>` shall be returned as the token identifier `NEWLINE`.

(2) If the token is an operator, the token identifier for that operator shall result.

(3) If the string consists solely of digits and the delimiter character is one of < or >, the token identifier `IO_NUMBER` shall be returned.

(4) Otherwise, the token identifier `TOKEN` shall result.

Further distinction on `TOKEN` is context-dependent. It may be that the same `TOKEN` yields `WORD`, a `NAME`, an `ASSIGNMENT`, or one of the reserved words below, dependent upon the context. Some of the productions in the grammar below are annotated with a rule number from the following list. When a `TOKEN` is seen where one of those annotated productions could be used to reduce the symbol, the applicable rule shall be applied to convert the token identifier type of the `TOKEN` to a token identifier acceptable at that point in the grammar. The reduction shall then proceed based upon the token identifier type yielded by the rule applied. When more than one rule applies, the highest numbered rule shall apply (which in turn may refer to another rule). [Note that except in rule (7), the presence of an = in the token has no effect.]

The `WORD` tokens shall have the word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

3.10.2 Shell Grammar Rules

(1) [Command Name]
When the `TOKEN` is exactly a reserved word, the token identifier for that
reserved word shall result. Otherwise, the token `WORD` shall be returned. 1
Also, if the parser is in any state where only a reserved word could be the 1
next correct token, proceed as above. 1

NOTE: Because at this point quote marks are retained in the token, quoted strings cannot be recognized as reserved words. This rule also implies that reserved words will not be recognized except in certain positions in the input, such as after a `<newline>` or semicolon; the grammar presumes that if the reserved word is intended, it will be properly delimited by the user, and does not attempt to reflect that requirement directly. Also note that line joining is done before tokenization, as described in 3.2.1, so escaped newlines are already removed at this point.

NOTE: Rule (1) is not directly referenced in the grammar, but is referred to by other 1
rules, or applies globally. 1

(2) [Redirection to/from filename]
The expansions specified in 3.7 shall occur. As specified there, exactly
one field can result (or the result is unspecified), and there are additional 1
requirements on pathname expansion.

(3) [Redirection from here-document]
Quote removal [3.7.4]. shall be applied to the word to determine the del- 1
imiter that will be used to find the end of the here-document that begins 1
after the next `<newline>`. 1

(4) [Case statement termination]
When the `TOKEN` is exactly the reserved word `Esac`, the token identifier for `Esac` shall result. Otherwise, the token `WORD` shall be returned.

(5) [`NAME` in `for`]
When the `TOKEN` meets the requirements for a name [3.1.5], the token identifier `NAME` shall result. Otherwise, the token `WORD` shall be returned.

(6) [Third word of `for` and `case`]
When the `TOKEN` is exactly the reserved word `In`, the token identifier for `In` shall result. Otherwise, the token `WORD` shall be returned.

(7) [Assignment preceding command name] 1

(a) [When the first word]
If the `TOKEN` does not contain the character `=`, rule (1) shall be applied. Otherwise, apply (7)(b).

(b) [Not the first word]
If the `TOKEN` contains the equals-sign character:

— If it begins with `=`, the token `WORD` shall be returned.

— If all the characters preceding `=` form a valid name [3.1.5], the token `ASSIGNMENT_WORD` shall be returned. (Quoted characters cannot participate in forming a valid name.)

— Otherwise, it is unspecified whether it is `ASSIGNMENT_WORD` or `WORD` that is returned.

Assignment to the `NAME` shall occur as specified in 3.9.1.

(8) [`NAME` in function]
When the `TOKEN` is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, when the `TOKEN` meets the requirements for a name [3.1.5], the token identifier `NAME` shall result. Otherwise, rule (7) shall apply.

(9) [Body of function]
Word expansion and assignment shall never occur, even when required by the rules above, when this rule is being parsed. Each `TOKEN` that might either be expanded or have assignment applied to it shall instead be returned as a single `WORD` consisting only of characters that are exactly the token described in 3.3.

```
/*      -------------------------------------------------------
        The grammar symbols
        ------------------------------------------------------- */

%token  WORD
%token  ASSIGNMENT_WORD
%token  NAME
%token  NEWLINE
%token  IO_NUMBER

/*      The following are the operators mentioned above.       */

%token  AND_IF OR_IF DSEMI
/*      '&&'   '||'  ';;'                                       */

%token  DLESS DGREAT LESSAND GREATAND LESSGREAT DLESSDASH
/*      '<<'  '>>'   '<&'    '>&'     '<>'      '<<-'           */

%token  CLOBBER
/*      '>|'                                                    */

/*      The following are the reserved words                    */

%token   If   Then   Else   Elif   Fi   Do   Done
/*      'if' 'then' 'else' 'elif' 'fi' 'do' 'done'              */

%token   Case   Esac   While   Until   For
/*      'case' 'esac' 'while' 'until' 'for'                     */

/*      These are reserved words, not operator tokens, and are
        recognized when reserved words are recognized.          */

%token  Lbrace Rbrace Bang
/*       '{'    '}'    '!'                                      */

%token  In
/*      'in'                                                    */

/*      -------------------------------------------------------
        The Grammar
        ------------------------------------------------------- */

%start  complete_command

%%

complete_command    : list separator
                    | list                                          1
                    ;

list                : list separator_op and_or
                    |                   and_or
                    ;

and_or              :                         pipeline
```

```
                     | and_or AND_IF linebreak pipeline
                     | and_or OR_IF  linebreak pipeline
                     ;
pipeline             :      pipe_sequence
                     | Bang pipe_sequence
                     ;
pipe_sequence        :                             command
                     | pipe_sequence '|' linebreak command
                     ;
command              : simple_command
                     | compound_command
                     | compound_command redirect_list
                     | function_definition
                     ;
compound_command     : brace_group
                     | subshell
                     | for_clause
                     | case_clause
                     | if_clause
                     | while_clause
                     | until_clause
                     ;
subshell             : '(' compound_list ')'
                     ;
compound_list        :              term
                     | newline_list term
                     |              term separator
                     | newline_list term separator
                     ;
term                 : term separator and_or
                     |                and_or
                     ;
for_clause           : For name                             do_group
                     | For name In wordlist sequential_sep do_group
                     ;
name                 : NAME                     /* Apply rule (5) */        2
                     ;
in                   : In                       /* Apply rule (6) */
                     ;
wordlist             : wordlist WORD
                     |          WORD
                     ;
case_clause          : Case WORD In linebreak case_list Esac
                     | Case WORD In linebreak           Esac
                     ;
case_list            : case_list case_item
                     |           case_item
                     ;
```

```
case_item           :     pattern ')' linebreak     DSEMI linebreak
                    |     pattern ')' compound_list DSEMI linebreak
                    | '(' pattern ')' linebreak     DSEMI linebreak           2
                    | '(' pattern ')' compound_list DSEMI linebreak           2
                    ;
pattern             :            WORD          /* Apply rule (4) */
                    | pattern '|' WORD         /* Do not apply rule (4) */    1
                    ;
if_clause           : If compound_list Then compound_list else_part Fi
                    | If compound_list Then compound_list           Fi
                    ;
else_part           : Elif compound_list Then else_part
                    | Else compound_list
                    ;
while_clause        : While compound_list do_group
                    ;
until_clause        : Until compound_list do_group
                    ;
function_definition : fname '(' ')' linebreak function_body
                    ;
function_body       : compound_command                /* Apply rule (9) */
                    | compound_command redirect_list  /* Apply rule (9) */
                    ;
fname               : NAME                            /* Apply rule (8) */    2
                    ;
brace_group         : Lbrace compound_list Rbrace
                    ;
do_group            : Do compound_list Done
                    ;
simple_command      : cmd_prefix cmd_word cmd_suffix
                    | cmd_prefix cmd_word
                    | cmd_prefix
                    | cmd_name cmd_suffix
                    | cmd_name
                    ;
cmd_name            : WORD                   /* Apply rule (7)(a) */
                    ;
cmd_word            : WORD                   /* Apply rule (7)(b) */
                    ;
cmd_prefix          :            io_redirect
                    | cmd_prefix io_redirect
                    |            ASSIGNMENT_WORD
                    | cmd_prefix ASSIGNMENT_WORD
                    ;
cmd_suffix          :            io_redirect
                    | cmd_suffix io_redirect
                    |            WORD
```

```
                    | cmd_suffix WORD
                    ;
redirect_list       :              io_redirect
                    | redirect_list io_redirect
                    ;
io_redirect         :           io_file
                    | IO_NUMBER io_file
                    |           io_here
                    | IO_NUMBER io_here
                    ;
io_file             : '<'       filename
                    | LESSAND   filename
                    | '>'       filename
                    | GREATAND  filename
                    | DGREAT    filename
                    | LESSGREAT filename
                    | CLOBBER   filename
                    ;
filename            : WORD                      /* Apply rule (2) */
                    ;
io_here             : DLESS     here_end
                    | DLESSDASH here_end
                    ;
here_end            : WORD                      /* Apply rule (3) */
                    ;
newline_list        :              NEWLINE
                    | newline_list NEWLINE
                    ;
linebreak           : newline_list
                    | /* empty */
                    ;
separator_op        : '&'
                    | ';'
                    ;
separator           : separator_op linebreak
                    | newline_list
                    ;
sequential_sep      : ';' linebreak
                    | newline_list
                    ;
```

3.10.3 Shell Grammar Rationale. *(This subclause is not a part of P1003.2)*

There are several subtle aspects of this grammar where conventional usage implies rules about the grammar that in fact are not true.

For `compound_list`, only the forms that end in a `separator` allow a reserved
word to be recognized, so usually only a `separator` can be used where a com- 1
pound list precedes a reserved word (such as `Then`, `Else`, `Do`, and `Rbrace`. Explicitly requiring a separator would disallow such valid (if rare) statements as:

```
if (false) then (echo x) else (echo y) fi
```

See the NOTE under special grammar rule (1).

Concerning the third sentence of rule (1) (‘‘Also, if the parser . . . ’’): 1

— This sentence applies rather narrowly: when a compound list is terminated 1
by some clear delimiter (such as the closing `fi` of an inner `if_clause`) 1
then it would apply; where the compound list might continue (as in after a 1
;), rule (7a) [and consequently the first sentence of rule (1)] would apply. In 1
many instances the two conditions are identical, but this part of rule (1) 1
does not give license to treating a `WORD` as a reserved words unless it is in a 1
place where a reserved word must appear. 1

— The statement is equivalent to requiring that when the LR(1) lookahead set 2
contains exactly a reserved word, it must be recognized if it is present. 2
(Here ‘LR(1)’’ refers to the theoretical concepts, not to any real parser gen- 2
erator.) 2

For example, in the construct below, and when the parser is at the point 2
marked with ˆ, the only next legal token is `then` (this follows directly from 2
the grammar rules). 2

```
if if....fi then ....  fi
           ˆ
```

At that point, the `then` must be recognized as a reserved word. 2

(Depending on the actual parser generator actually used, ‘‘extra’’ reserved 2
words may be in some lookahead sets. It does not really matter if they are 2
recognized, or even if any possible reserved word is recognized in that state, 2
because if it is recognized and is not in the (theoretical) LR(1) lookahead 2
set, an error will ultimately be detected. In the example above, if some 2
other reserved word (e.g., `while`) is also recognized, an error will occur 2
later. 2

This is approximately equivalent to saying that reserved words are recog- 2
nized after other reserved words (because it is after a reserved word that 2
this condition will occur), but avoids the ‘‘except for. . .’’ list that would be 2
required for `case`, `for`, etc. (Reserved words are of course recognized any- 2
where a `simple_command` can appear, as well. Other rules take care of 2
the special cases of nonrecognition, such as rule (4) for case statements.) 2

Note that the body of here-documents are handled by Token Recognition (see 3.3) and do not appear in the grammar directly. (However, the here-document I/O redirection operator is handled as part of the grammar.)

The start symbol of the grammar (`complete_command`) represents either input from the command line or a shell script. It is repeatedly applied by the interpreter to its input, and represents a single ‘‘chunk’’ of that input as seen by the
interpreter. 1

The processing of here-documents is handled as part of token recognition (see 3.3) rather than as part of the grammar.

3.11 Signals and Error Handling

When a command is in an asynchronous list, the shell shall prevent SIGQUIT and SIGINT signals from the keyboard from interrupting the command. Otherwise, signals shall have the values inherited by the shell from its parent (see also 3.14.13).

When a signal for which a trap has been set is received while the shell is waiting 1
for the completion of a utility executing a foreground command, the trap associ- 1
ated with that signal shall not be executed until after the foreground command 1
has completed. When the shell is waiting, by means of the `wait` utility, for asyn- 1
chronous commands to complete, the reception of a signal for which a trap has 1
been set shall cause the `wait` utility to return immediately with an exit status 1
>128, immediately after which the trap associated with that signal shall be taken. 1

If multiple signals are pending for the shell for which there are associated trap actions (see 3.14.13), the order of execution of trap actions is unspecified.

3.12 Shell Execution Environment

A shell execution environment consists of the following:

— Open files inherited upon invocation of the shell, plus open files controlled by `exec`.

— Working Directory as set by `cd` (see 4.5).

— File Creation Mask set by `umask` (see 4.67).

— Current traps set by `trap` (see 3.14.13).

— Shell parameters that are set by variable assignment (see `set` in 3.14.11) or from the POSIX.1 {8} environment inherited by the shell when it begins (see `export` in 3.14.8).

— Shell functions (see 3.9.5.)

— Options turned on at invocation or by `set`.

— Process IDs of the last commands in asynchronous lists known to this shell 1
environment; see 3.9.3.1. 1