IEEE P1003.2 Draft 11.2 – September 1991

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 compound 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 . . .''):

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

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

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

```
if if....fi then ....  fi
           ^
```

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

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

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

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.

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

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 environment; see 3.9.3.1.