Bartlomiej JANIK

bartlomiej.janik@hotmail.fr

21SH PROJECT REPORT

21SH = ½ * 42SH

Reference documents	
N°	**Name**
1	The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition
2	Interprétation et Compilation, Jean Méhat, Université Paris VIII, 2013
3	Compilers: Principles, Techniques, and Tools (The Dragon Book), Alfred Aho, Ravi Sethi, Jeffrey Ullman, 1986
4	Unix Network Programming, W. Richard Stevens, 1990

TABLE OF CONTENTS

1. Introduction 4
1.1. Goals of 21sh 4
1.2. Grammar 4
2. Design 6
2.1. Shell implementation diagram 6
2.2. Shell initialization 6
2.3. User Input 7
2.4. Lexer 8
2.5. Parser 9
2.6. Execution engine 10
3. Implementation 11
4. What's next? 12

1. INTRODUCTION

1.1. GOALS OF 21SH

This project consists of consolidating the previous shell project, aka minishell: 21sh will have to integrate several mandatory features including:

- Complete line edition management
- Handling redirections (>, >>, <, <<, >&, <&)
- Handling pipes
- Command separator ‘;’
- Inhibitors (quotes, double-quotes and backslash).

We personally added to this some extra features:

- Logical operators AND_IF (&&) and OR_IF (||)
- History Search (with CTRL + R)
- Built-in history
- Built-in export to manage local and global environment variables.

1.2. GRAMMAR

The whole shell scripting part not being mandatory, I produced a simplified shell grammar for this project, based on the complete original one.

This grammar was validated with YACC. The output file generated by YACC (y.output) with this grammar as input was used to build the parser automaton.

```
0  $accept : program $end
1  program : complete_command NEWLINE
2           | NEWLINE
3  complete_command : list separator_op
4           | list
5  list : list separator_op and_or
6           | and_or
7  and_or : pipeline
8     | and_or AND_IF pipeline
9     | and_or OR_IF pipeline
10  pipeline : command
11           | pipeline '|' command
12  command : cmd_prefix cmd_word cmd_suffix
13          | cmd_prefix cmd_word
14          | cmd_prefix
15          | cmd_name cmd_suffix
16          | cmd_name
17  cmd_name : WORD
18  cmd_word : WORD
19  cmd_prefix : io_redirect
20             | cmd_prefix io_redirect
21  cmd_suffix : io_redirect
22             | cmd_suffix io_redirect
23             | WORD
24             | cmd_suffix WORD
25  io_redirect : io_file
26              | IO_NUMBER io_file
27              | io_here
28              | IO_NUMBER io_here
29  io_file : '<' filename
30          | LESSAND filename
31          | '>' filename
32          | GREATAND filename
33          | DGREAT filename
34  filename : WORD
35  io_here : DLESS here_end
36 here_end: WORD
37 separator_op     : '&'
38          | ';'
```

2. DESIGN

2.1. SHELL IMPLEMENTATION DIAGRAM

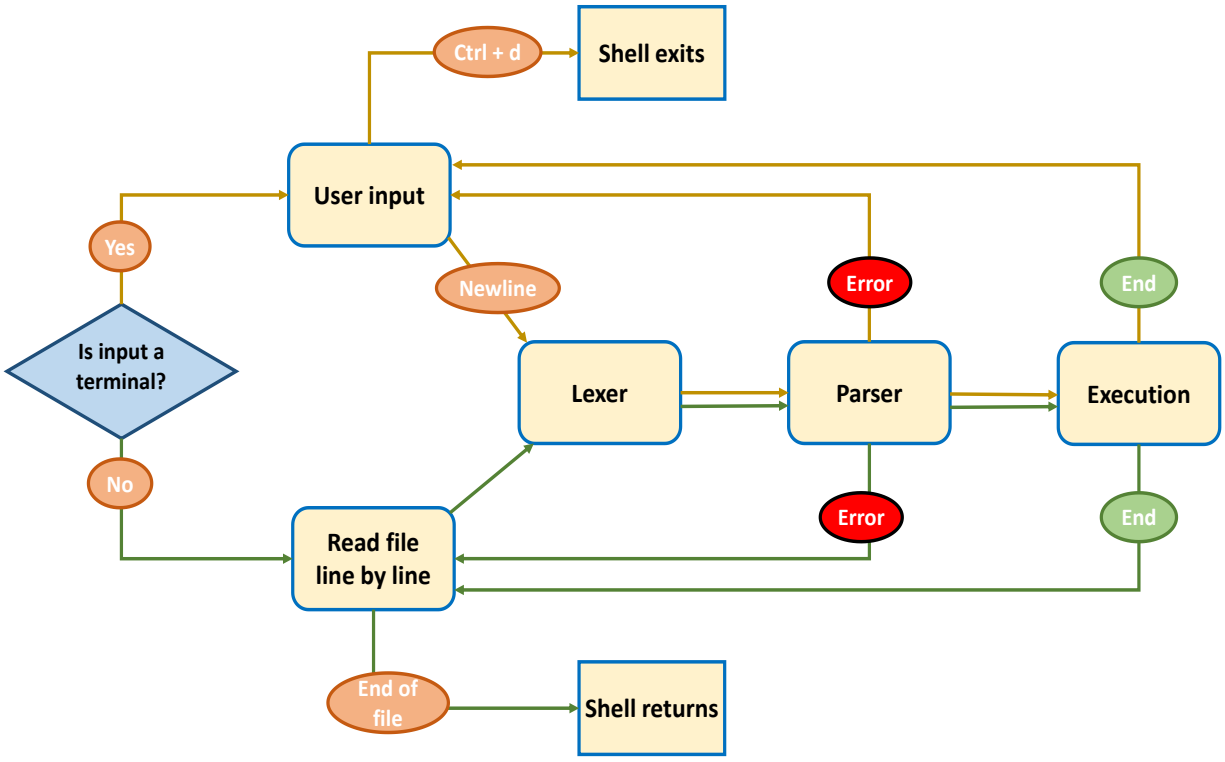


Figure 1: Shell implementation diagram

2.2. SHELL INITIALIZATION

First of all, we need to know if the standard input is referring a terminal or not. If it's the case, the shell is initialized in interactive mode, sets custom terminal settings for using termcaps library.

If the standard input does not refer to a terminal – meaning that the shell's input is a file – no specific terminal settings is needed.

2.3. User Input

The input buffer is handled as a string to which are appended – or deleted – characters depending on the keys typed by the user. Check the README file for the whole key bounds.

The input structure contains several variables that are updated is necessary.

- The input buffer containing user's input.
- The length of the input buffer.
- The size – number of bytes allocated - of the input buffer: if the length is about to become bigger than the size, then the shell reallocates enough memory to make sure there is no memory overflow.
- The cursor's position in the input buffer: each time the cursor moves this variable is updated.

When user presses RETURN key, a newline character is appended at the input's end, and input is passed to the lexer.

2.4. LEXER

The goal of the lexer is to split the input into tokens, accordingly to rules specified by the shell specification and assigning to each token a type (WORD, NEWLINE, operator token).

The lexer returns a list of tokens, which will be passed to the parser.

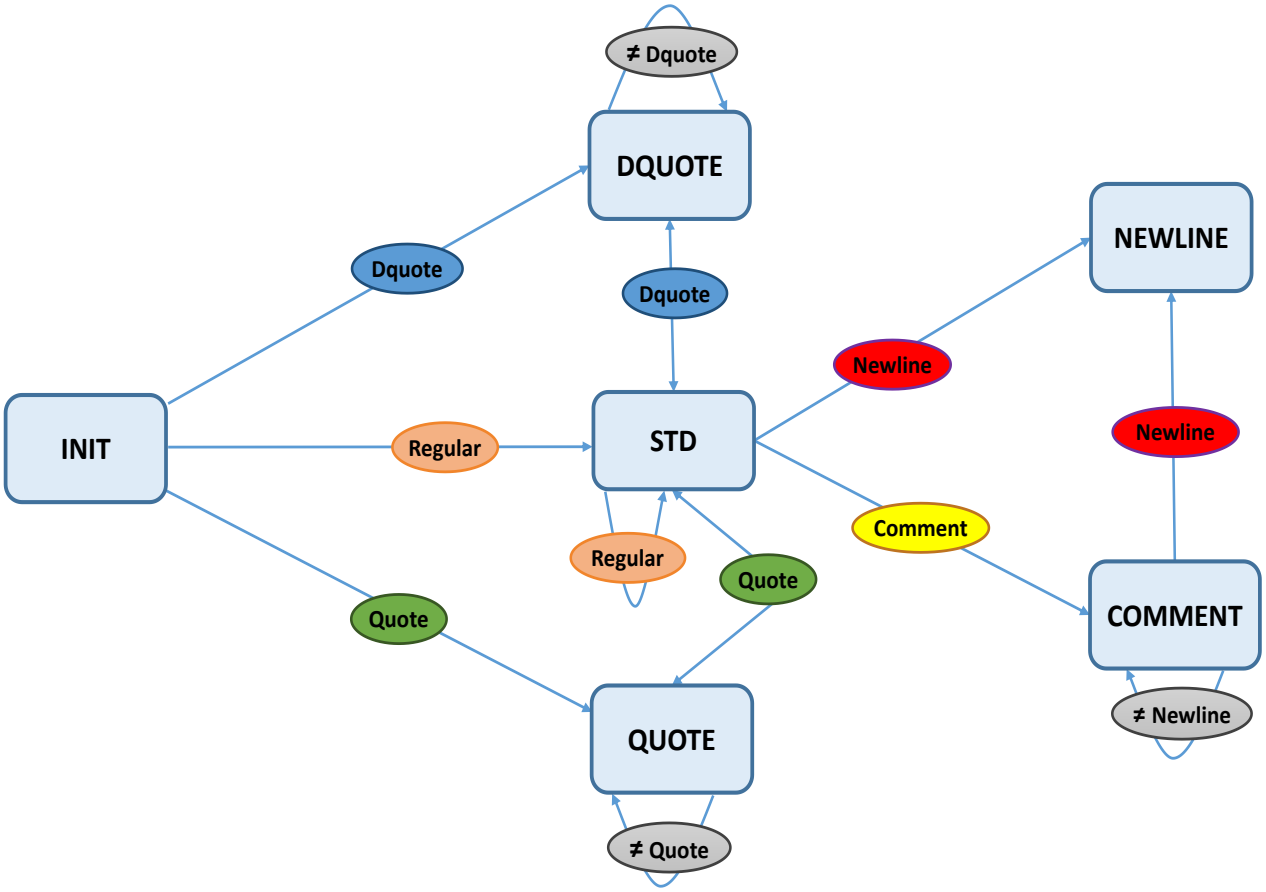
The lexer was coded as an automaton with two entries:

- The current state
- The current character
- For each couple {state, character} there is a transition function, whose pointer is stored in the automaton table, called and a resulting state.
- The lexer's automaton diagram is showed below.

The initial state is INIT and the final is NEWLINE. The events which can occur are the following:

- Dquote
- Quote
- Comment ("#")
- Newline
- Regular character (matches any character which does not belong to any of the previous categories)

Figure 2: Lexer state diagram



2.5. PARSER

The goal of the parser is first to check if the construction of the input is correct, and also to store all the data needed for the execution of the commands.

The parser implementation is based on a shift-reduce stack automaton. The parser's input data is the list of tokens previously generated by the lexer.

When receiving a token, the parser can do one of these actions:

- Push the token on the stack and switch the state (shift action)
- Pop one or more elements from the stack : one or more previously pushed tokens can be used, accordingly to one rule of the grammar, to form another symbol (reduce action)
- Detect a syntax error, and stop its processing and display an appropriate message
- Accept the input, meaning the input is grammatically correct.

The conditions of performing one of these actions depends on the current state and the current token type read by the parser.

The number of states and events would make the parser's automaton diagram unreadable, that's why we decided not to show it.

2.6. EXECUTION ENGINE

Once the parser accepted the input, the execution process can start.

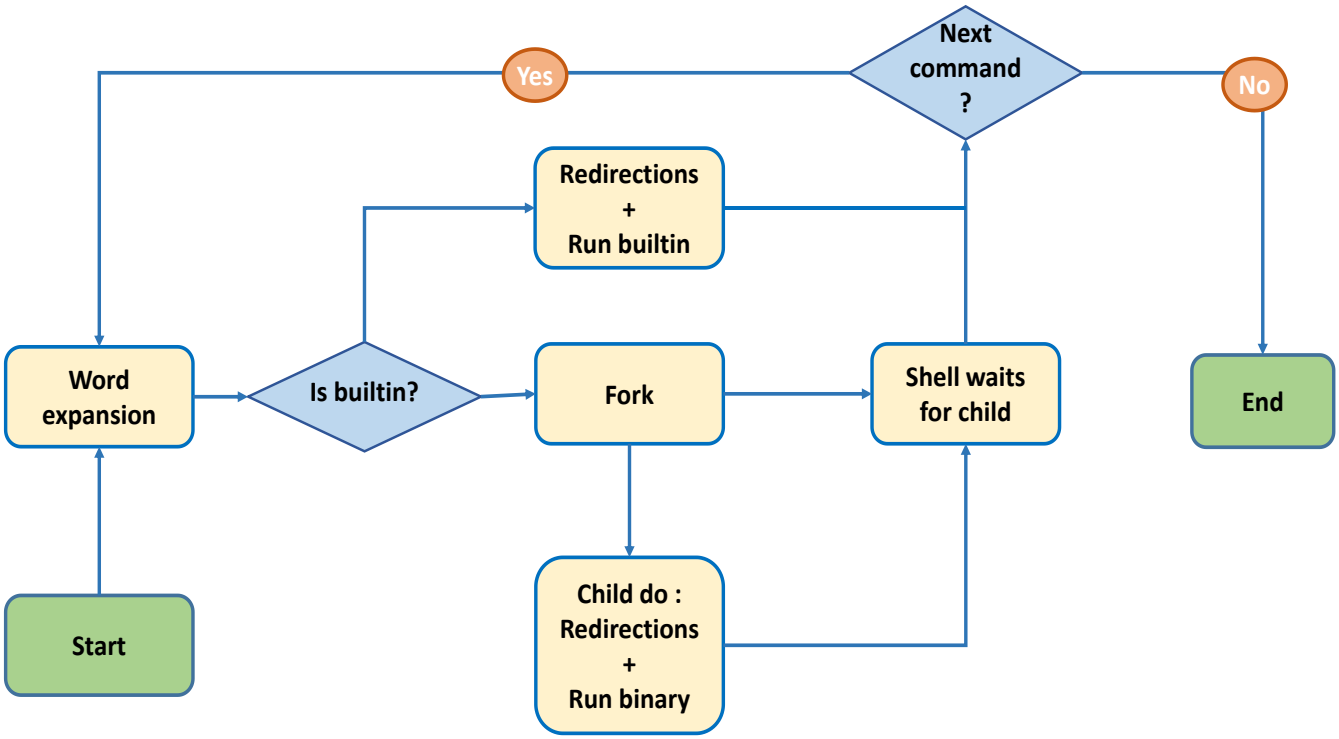
Several commands – separated by tokens SEMI or AND_IF or OR_IF – can be executed during one execution process.

These commands are stored in a linked list. Each element of the linked list contains a linked list of the command's arguments (this list will be converted to a string array in order to pass it as an argument to the system call *execve*), a linked list of all redirections that the command should perform.

The word expansion occurs just before the execution of each command. The word expansion consists of tilde expansion, environment variable expansion (ex: $VAR_NAME) and quote removal.

If the command is a builtin, then it is executed in the shell context, not in a child process. Otherwise, after creating the child process with the *fork* system call, the shell waits for the child's exit status.

Figure 3: Execution engine diagram



3. IMPLEMENTATION

The implementation relies on a main data structure which definition is showed below:

```
typedef struct  s_bsh

{

        t_input         *input;   Stores input buffer variables

        t_term          *term;    Stores initial and custom terminal settings

        t_lexer         *lexer;

        t_parser        *parser;

        t_exec          *exec;    Stores data needed for execution (redirections, argv…)

        t_env           *env;     Stores the environment

        t_env           *mod_env; Stores custom environment (when calling env builtin)

        t_history       *history; Stores all the historic commands

        t_token         *tokens[2]; Stores the tokens generated by the lexer.

        t_expander     *exp;

        pid_t           pid;      Stores the shell's process ID

        t_pipes         *pipes;   Store pipes  in the case of a pipeline sequence

        short int       interactive;  flag : 1 if shell is interactive, otherwise 0

        int             exit_status; Store last command's exit status

        char            *shell_name;

        int             saved_fds[3]; Store copies of STDIN, STDOUT and STDERR

        int             env_index;

        char            env_options[3];

}                       t_bsh;
```

The structure stores all the data needed in the program. For a better encapsulation and data separation it is implemented as a singleton. It is allocated when the program is started and we can access it anywhere later on by calling a function that returns a pointer to it.

```
t_bsh   *get_bsh(void)
{
        static t_bsh    *bsh = NULL;

        if (!bsh)
                bsh = init_bsh();
        return (bsh);
}
```

4. WHAT'S NEXT?

Next step to improve this shell is 42sh, 42's school final shell project. Although 21sh was made solo, 42sh is a team work.

Here are some features that could be implemented for 42sh:

- Completion
- Builtin Read
- Globbing (specify sets of filenames with wildcard characters such as '*', '?', […]).
- Hash table for binaries (remember the full pathnames of commands specified as name arguments, so they need not to be searched for on subsequent invocations).
- Bang '!' character for history expansion (ex : '!!' is last command typed, '!foo' is last command containing 'foo' etc …)
- Shell scripting (while , if then else, for)