Morbig: A Static Parser for POSIX Shell

Yann Régis-Gianas, Nicolas Jeannerod, Ralf Treinen

Morbig: A Static Parser for POSIX Shell

Yann Régis-Gianas
IRIF, Université Paris-Diderot, CNRS, INRIA PI.R2
Paris, France

Nicolas Jeannerod
IRIF, Université Paris-Diderot, CNRS
École normale supérieure
Paris, France

Ralf Treinen
IRIF, Université Paris-Diderot, CNRS
Paris, France

Abstract

The POSIX shell language defies conventional wisdom of compiler construction on several levels: The shell language was not designed for static parsing, but with an intertwining of syntactic analysis and execution by expansion in mind. Token recognition cannot be specified by regular expressions, lexical analysis depends on the parsing context and the evaluation context, and the shell grammar given in the specification is ambiguous. Besides, the unorthodox design choices of the shell language fit badly in the usual specification languages used to describe other programming languages. This makes the standard usage of Lex and Yacc as a pipeline inadequate for the implementation of a parser for POSIX shell. The existing implementations of shell parsers are complex and use low-level character-level parsing code which is difficult to relate to the POSIX specification. We find it hard to trust such parsers, especially when using them for writing automatic verification tools for shell scripts.

This paper offers an overview of the technical difficulties related to the syntactic analysis of the POSIX shell language. It also describes how we have resolved these difficulties using advanced parsing techniques (namely speculative parsing, parser state introspection, context-dependent lexical analysis and longest-prefix parsing) while keeping the implementation at a sufficiently high level of abstraction so that experts can check that the POSIX standard is respected. The resulting tool, called Morbig, is an open-source static parser for a well-defined and realistic subset of the POSIX shell language.

CCS Concepts • **Software and its engineering → Parsers**;

Keywords Parsing, POSIX shell, functional programming

ACM Reference Format:
Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. 2018. Morbig: A Static Parser for POSIX Shell. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276615

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276615

1 Introduction

Scripts are everywhere on UNIX machines, and many of them are written in POSIX shell. The POSIX shell is a central piece in the toolbox of a system administrator who may use it to write scripts that perform all kinds of repetitive administration tasks. Furthermore, scripts are used in a systematic way by GNU/Linux distributions for specific tasks, like for writing cron jobs which are regularly executed, init scripts (depending on the init system) that start or stop services, or scripts which are executed as part of the process of installing, removing or upgrading software packages. The Debian GNU/Linux distribution, for instance, contains 31.832[1] of these so-called maintainer scripts, 31,521 of which are written in POSIX shell.

These scripts are often executed with `root` privileges since they have to act on the global system installation, for instance when installing software packages. As a consequence, erroneous scripts can wreak havoc on a system, and there is indeed a history of disastrous shell scripts (one of the authors of this paper takes the blame for one of these). An ongoing research project[2] aims at using formal verification tools for analyzing shell scripts.

The first step when statically analyzing shell scripts is to analyze their syntactic structure, and to produce a syntax tree. This seems at first sight an easy task: after all, the POSIX standard contains a grammar, so one might think that a parser can be thrown together in a day or so, reusing what one has learned in an introductory course on compiler construction. The reality is far from that! It starts with the fact that the POSIX shell language was never designed for being statically analyzed. In fact, the shell analyses pieces of syntax of a script, on the fly, in a process that is intertwined with an evaluation mechanism called expansion. But this is only the start, the syntax of POSIX shell is full of pitfalls which we will explain in detail in the next section, and which make it surprisingly difficult to write a parser for POSIX shell.

For this reason, existing implementations of shell interpreters contain hand-crafted syntactic analyzers that are very hard to understand. Due to the way the shell semantics is defined, they do not construct a complete syntax tree, but produce pieces of syntax on the fly. We could probably have taken one of these implementations and tweaked it into constructing a complete syntax tree. The problem is, how can we trust such a parser? The parser is an essential part of our

[1] `unstable`, `amd64` architecture, as of 29/11/2016
[2] CoLiS, "Correctness of Linux Scripts", https://colis.irif.fr

tool chain, if the parser produces incorrect syntax trees then all formal analysis based on it will be worthless.

The standard techniques to implement syntactic analyzers are based on code generators. Using code generators is an excellent software engineering practice which allows us to write high-level and easily maintainable code. These tools take as input high-level formal descriptions of the lexical conventions and of the grammar and produce low-level efficient code using well-understood computational devices (typically finite-state transducers for lexical analysis, and pushdown automata for parsing). This standard approach is trustworthy because (i) the high-level descriptions of the lexical conventions and grammar are usually close to their counterparts in the specification; (ii) the code generators are based on well-known algorithms like LR-parsing which have been studied for almost fifty years[16]. The problem with this approach is that the standard Lex-Yacc pipeline is inadequate for POSIX shell, as we will argue in the next section. Despite the pitfalls of the shell language, we nonetheless managed to maintain an important part of generated code in our implementation, described in Section 3. To sum things up, we claim the following contributions:

(i) This paper provides an overview of the difficulties related to the syntactic analysis of the POSIX shell language as well as a list of technical requirements that are, in our opinion, needed to implement a static parser for this language.

(ii) This paper describes a modular architecture that arguably simplifies code review, especially because it follows the POSIX specification decomposition into token recognition and syntactic analysis, and because it embeds the official BNF grammar, which makes more explicit the mapping between the specification and the implementation.

(iii) This paper is finally a demonstration that an LR(1) parser equipped with a purely functional and incremental interface is a lightweight solution to realize the advanced parsing techniques required by POSIX shell parsing, namely speculative and reentrant parsing, longest-match parsing as well as parsing-dependent "negatively specified" lexing.

2 The perils of POSIX shell

The POSIX Shell Command Language is specified by the Open Group and IEEE in the volume "Shell & Utilities" of the POSIX standard. Our implementation is based on the latest published draft of this standard [14].

This standardization effort synthesizes the common concepts and mechanisms that can be found in the most common implementations of shell interpreters like `bash` or `dash`. Unfortunately, as said in the introduction, it is really hard to extract a high-level declarative specification out of these existing implementations because the shell language is inherently irregular, and because its unorthodox design choices fit badly in the usual specification languages used by other programming language standards.

Syntactic analysis is most often decomposed into two distinct phases: (i) *lexical analysis*, which synthesizes a stream of lexemes from a stream of input characters by recognizing lexemes as meaningful character subsequences and by ignoring insignificant character subsequences such as layout; (ii) *parsing* which synthesizes a parse tree from the stream of tokens according to some formal grammar.

In this section, we describe several aspects which make the shell language hard (and actually impossible in general) to parse using the standard decomposition described above, and more generally using the standard parsing tools and techniques. These difficulties not only raise a challenge in terms of programming but also in terms of reliability.

2.1 Non standard lexical conventions

2.1.1 Token recognition

In usual programming languages, most of the categories of tokens are specified by means of regular expressions. As explained earlier, lexer generators such as Lex conveniently turn such high-level specifications into efficient finite state transducers, which makes the resulting implementation both reliable and efficient.

The token recognition process for the shell language is described in Section 2.3 of the specification [13], unfortunately without using any regular expressions. While other languages use regular expressions with a longest-match strategy to recognize the next lexeme in the input, the specification of the shell language is formulated in a "negative way". Indeed, token recognition is based on a state machine which explains instead how tokens must be *delimited* in the input and how these delimited chunks must be classified into two categories: words and operators.

The state machine which recognizes the tokens is unfortunately not a regular finite state machine. It is almost as powerful as a pushdown automaton since it must be able to recognize nested quotations like the ones found in the following example.

Example 2.1 (Quotations)**.** Consider the following input:

```
1  BAR='foo'"ba"r
2  X=0 echo x$BAR" "$(echo $(date))
```

By the lexical conventions of most programming languages, the first line would be decomposed as five distinct tokens (namely `BAR`, `=`, `'foo'`, `"ba"` and `r`). On the contrary, the lexical conventions of the shell language considers the entire line `BAR='foo'"ba"r` as a single token, classified into the category of words. On the second line, the input is split into the tokens `X=0`, `echo` and `x$BAR" "$(echo $(date))`. Notice that the third token contains nested quotations of the form `$(..$(..) )` the recognition of which is out of the scope of regular finite state machines (without a stack).

2.1.2 Layout

The shell language also has some unconventional lexical conventions regarding the interpretation of newline characters. Usually, newline characters are simply ignored by the lexing phase since they only serve as delimiters between tokens. In shell, however, newline characters are meaningful, and there are even four different interpretations of a newline depending on the parsing context. Therefore, most of the newline characters (but not all, as we shall see in the next example) must be transmitted to the parser. Hence, one step of token recognition may produce several tokens: the delimited token and a potential delimiter that must also be transmitted to the parser. Again, this is not common practice since, usually, lexical scanners produce at most one token every time they are invoked.

Example 2.2 (Interpretations of newline characters)**.** The four interpretations of the newline characters occur in the following example:

```
1 $ for i in 0 1
2 > # Some interesting numbers
3 > do echo $i \
4 > + $i
5 > done
```

On line 1, the newline character has a syntactic meaning because it acts as a marker for the end of the sequence over which the **for**-loop is iterating. On line 2, the newline character at the end of the comment must not be ignored but is merged with the newline character of the previous line. On line 3, the newline character is preceded by a backslash. This sequence of characters is interpreted as a line-continuation, which must be handled at the lexing level. That is, in this case the newline is actually interpreted as layout. On lines 4 and 5, each of the final newlines terminates a command.

The recognition of comments of shell scripts are also non conventional. Even though the specification rule regarding comments seems quite standard:

> If the current character is a '#', it and all subsequent characters up to, but excluding, the next <newline> shall be discarded as a comment. The <newline> that ends the line is not considered part of the comment.

the fact that '#' is not a delimiter allows a word to contain the character '#', as in the following example.

Example 2.3.

```
1 ls foo#bar
```

In that example, `foo#bar` is recognized as a single word.

2.1.3 Delimiting subshell invocations

From the lexical point of view, a subshell invocation is simply a word. Delimiting these subshell invocations is hardly reducible to regular expression matching. Indeed, to determine the end of a subshell invocation, it is necessary to recursively call the shell command parser so that it consumes the rest of the input until a complete command is parsed.

Example 2.4 (Finding closing parenthesis requires context)**.**

```
1 echo $(echo ')')
```

On line 1, the first occurrence of the right parenthesis does not end the subshell invocation started by `$(` because it is written between single quotes.

2.1.4 Character escaping

String literals of most programming languages may contain escaping sequences to let the programmer use the entire character set including string delimiters. The backslash character typically introduces such escaping sequence as in `"\""` to insert a double quote or in `"\\"` to insert a backslash. The rule of escaping is pretty simple: if a character is preceded by a backslash, it must retain its literal meaning.

In a static parser for POSIX shell, this rule is significantly more complex because the nesting of double quotes and subshell invocations have an impact on the number of backslashes needed to escape a character, as shown by the following example.

Example 2.5 (Number of backslashes to escape)**.**

```
1 echo " ` echo \"  \\" \"  ` "
2 echo `echo \`echo \\\`echo \
3   \\\\\\\`echo foobar\\\\\\\`\\\`\`
```

On line 1, a subshell is nested inside a double-quoted string literal: in the subshell invocation, the first occurence of the character '"' is not escaped even though it is preceded by a backslash ; on the contrary, the second occurrence of '"' is escaped because it is preceded by two backslashes. The command starting on line 2 illustrates the dependency between the number of backslashes required to escape a character and the nesting depth of subshell invocations.

2.1.5 Here-documents

Depending on the parsing context, the lexer must switch to a special mode to deal with here-documents. Here-documents are chunks of text embedded in a shell script. They are commonly used to implement some form of template-based generation of files (since they may contain variables). To use that mode, the user provides textual end-markers and the lexer then interprets all the input up to an end-marker as a single token of the category of words. The input characters are copied verbatim into the representation of the token, with the possible exception of quotations which may still be recognized exactly as in the normal lexing mode.

Example 2.6 (Here-documents)**.**

```
1 cat > notifications << EOF
2 Hi $USER!
3 EOF
4 cat > toJohn << EOF1 ; cat > toJane << EOF2
5 Hi John!
6 EOF1
```

```
%token  WORD ASSIGNMENT_WORD NAME NEWLINE IO_NUMBER
// The following are the operators (see XBD Operator)
// containing more than one character.
%token  AND_IF    OR_IF    DSEMI // '&&' '||' ';;'
%token  DLESS  DGREAT  LESSAND // '<<' '>>' '<&'
%token 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'
```

Figure 1. The tokens of the shell language grammar.

```
Hi Jane!
EOF2
```

In this example, the text on lines 2 and 3 is interpreted as a single word which is passed as input to the `cat` command. The first `cat` command of line 5 is fed with the content of line 6 while the second `cat` command of line 5 is fed with the content of line 8. This example with two successive here-documents illustrates the non-locality of the lexing process of here-document: the word related to the end-marker `EOF1` is recognized several tokens after the introduction of `EOF1`. This non-locality forces some form of forward declaration of tokens, the contents of which is defined afterwards.

2.2 Parsing-dependent lexical analysis

While the recognition of tokens is independent from the parsing context, their classification into words, operators, newlines and end-of-file markers must be refined further to obtain the tokens actually used in the formal grammar specified by the standard. The declaration of these tokens is reproduced in Figure 1. While a chunk categorized as an operator is easily transformed into a more specific token like `AND_IF` or `OR_IF`, an input chunk categorized as a word can be promoted to a reserved word or to an assignment word only if the parser is expecting such a token at the current position of the input; otherwise the word is not promoted and stays a `WORD`. This means that the lexical analysis has to depend on the state of the parser. The following two sections describe this specific aspect of the shell syntax.

2.2.1 Parsing-sensitive assignment recognition

The promotion of a word to an assignment depends both on the position of this word in the input and on the string representing that word. The string must be of the form `w=u` where the substring `w` must be a valid name, a lexical category defined in Section 3.235 of the standard by the following sentence:

> [...] a word consisting solely of underscores, digits, and alphabetics from the portable character set. The first character of a name is not a digit.

Example 2.7 (Promotion of a word as an assignment).

```
1 CC=gcc make
2 make CC=cc
3 ln -s /bin/ls "X=1"
4 "./X"=1 echo
```

On line 1, the word `CC=gcc` is recognized as a word assignment of `gcc` to `CC` because `CC` is a valid name for a variable, and because `CC=gcc` is written just before the command name of the simple command `make`. On line 2, the word `CC=cc` is not promoted to a word assignment because it appears after the command name of a simple command. On line 4, since `"./X"` is not a valid name for a shell variable, the word `"./X=1"` is not promoted to a word assignment and is interpreted as the command name of a simple command.

2.2.2 Parsing-sensitive keyword recognition

A word is promoted to a reserved word if the parser state is expecting this reserved word at the current point of the input:

Example 2.8 (Promotion of a word to a reserved word).

```
1 for i in a b; do echo $i; done
2 ls for i in a b
```

On line 1, the words `for`, `in`, `do`, `done` are recognized as reserved words. On line 2, they are not recognized as such since they appear in position of command arguments for the command `ls`.

In addition to this promotion rule, some reserved words can never appear in the position of a command.

Example 2.9 (Forbidden position for specific reserved words).

```
1 else echo foo
```

The word `else` must be recognized as a reserved word and the parser must reject this input.

2.2.3 Richly structured semantic values

The semantic value of a word can be complex since it can be made of subshell invocations, variables and literals. As a consequence, even though the grammar considers a word as an atomic piece of lexical information, its semantic value is represented by a dedicated concrete syntax tree.

Example 2.10 (Forbidden position for specific reserved words).

```
1 x="$(echo foo)${x:-bar}"baz
```

This script is a single word read as an `ASSIGNMENT_WORD` by the grammar. The semantic value of this lexeme is a sequence of a double-quoted sequence followed by a literal. The double-quoted sequence is itself composed of a subshell invocation represented by the concrete syntax tree of its command, followed by a variable that uses the default value `bar` when expanded.

2.3 Evaluation-dependent lexical analysis

The lexical analysis also depends on the evaluation of the shell script. Indeed, the `alias` builtin command of the POSIX shell amounts to the dynamic definition of macros which are expanded just before lexical analysis. Therefore, even the lexical analysis of a shell script cannot be done without executing it, that is, lexical analysis of unrestricted shell scripts is undecidable. Fortunately, restricting the usage of the `alias` command to top level commands only (that is, outside of any control structure) and performing expansion of these aliases in a preprocessing pass of the parser allows us to implement a simple form of alias expansion without endangering decidability.

Example 2.11 (Lexical analysis is undecidable)**.**

```
1 if ./foo; then
2   alias x="ls"
3 else
4   alias x=""
5 fi
6 x for i in a b; do echo $i; done
```

To decide if `for` in line 6 is a reserved word, a lexer must be able to know the success of an arbitrary program `./foo`, which is impossible to do statically. Hence, the lexer must wait for the evaluation of the first command before parsing the second one.

```
1 alias x="ls"
2 x for i in a b; do echo $i; done
```

If the shell script only uses `alias` at the top level, the parser can maintain a table for aliases and apply on-the-fly a substitution of aliases by their definitions just before the lexical analysis. Notice that this substitution introduces a desynchronization between the positions of tokens in the lexing buffer and their actual positions in the source code: this complicates the generation of precise locations in error messages.

Another problematic feature of the shell language is `eval`. This builtin constructs a command by concatenating its arguments, separated by spaces, and then executes the constructed command in the shell. In other words, the *construction* of the command that will be executed depends on the execution of the script, and hence cannot be statically known by the parser.

2.4 Ambiguous grammar

The grammar of the shell language is given in Section 2.10 of the standard. Due to lack of space we only reproduce a fragment of it in Figure 2. At first sight, the specification seems to be written in the input format of the Yacc parser generator. However, Yacc cannot handle this specification as-is for two reasons: (i) the specification is annotated with nine special rules which are not directly expressible in terms of Yacc's parsing mechanisms; (ii) the grammar contains LR(1) conflicts.

2.4.1 Special rules

The nine special rules of the grammar are actually the place where the parsing-dependent lexical conventions are explained. By lack of space, we only focus on the Rule 4 to give the idea. This is an excerpt from the standard describing this rule:

> *[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.*

The grammar refers to that rule in the following case:

```
pattern:
 WORD /* Apply rule 4 */
| pattern '|' WORD /* Do not apply rule 4 */;
```

Roughly speaking, this annotation says that when the parser is recognizing a `pattern` and when the next token is the specific `WORD` **esac**, then the next token is actually not a `WORD` but the token `Esac`. In that situation, one can imagine that an LR parser must pop up its stack to a state where it is recognizing the non terminal `case_clause` defined as follows:

```
case_clause:
Case WORD linebreak in linebreak case_list Esac
| Case WORD linebreak in linebreak case_list_ns Esac
| Case WORD linebreak in linebreak Esac
```

to conclude the recognition of the current `case_list`.

2.4.2 LR(1) conflicts

Our LR(1) parser generator detects five shift/reduce conflicts in the Yacc grammar of the standard. All these conflicts are related to the analysis of newline characters in the body of case items in case analysis. Indeed, the grammar is not LR(1) with respect to the handling of these newline characters. Here is the fragment of the grammar that is responsible for these conflicts:

```
compound_list: linebreak term | linebreak term separator;
case_list_ns : case_list case_item_ns | case_item_ns;
case_list    : case_list case_item | case_item;
case_item_ns :     pattern ')' linebreak
             |     pattern ')' compound_list
             | '(' pattern ')' linebreak
             | '(' pattern ')' compound_list;
case_item    :     pattern ')' linebreak     DSEMI linebreak
             |     pattern ')' compound_list DSEMI linebreak
             | '(' pattern ')' linebreak     DSEMI linebreak
             | '(' pattern ')' compound_list DSEMI linebreak;
separator    : separator_op linebreak | newline_list;
newline_list : NEWLINE | newline_list NEWLINE;
linebreak    : newline_list | /* empty */;
```

When a `NEWLINE` is encountered after `term` in a context of the following form:

```
1 case ... in ...)
```

an LR parser cannot choose between reducing the `term` into a `compound_list` or shifting the `NEWLINE` to start the recognition of the final `separator` of the current `compound_list`.

Fortunately, as the newline character has no semantic meaning in the shell language, choosing between reduction or shift has no significant impact on the output parse tree.

```
program:
  linebreak complete_commands linebreak | linebreak;
complete_commands:
  complete_commands newline_list complete_command
| complete_command;
complete_command:
  list separator_op | list;
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:
  linebreak term | linebreak term separator;
term:
  term separator and_or | and_or;
while_clause:
  While compound_list do_group;
do_group:
  Do compound_list Done /* Apply rule 6 */;
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 7a */;
cmd_word:
  WORD /* Apply rule 7b */;
newline_list:
  NEWLINE | newline_list NEWLINE;
linebreak:
  newline_list | /* empty */;
separator_op:
  '&' | ';';
separator:
  separator_op linebreak | newline_list;
sequential_sep:
  ';' linebreak | newline_list;

// The rules for the following nonterminals are elided:
// for_clause, name, in, wordlist, case_clause,
// case_list_ns, case_list, case_item_ns, case_item,
// pattern if_clause, else_part, until_clause,
// function_definition, function_body, fname,
// brace_group, cmd_prefix, cmd_suffix, redirect_list,
// io_redirect, io_file, filename, io_here and here_end
    .
```

Figure 2. A fragment of the official grammar for the shell language.

3 Unorthodox parsing

Our parser library is designed for a variety of applications, including statistical analysis of the concrete syntax of scripts (see, for instance, Section 4.2). Therefore, contrary to parsers typically found in compilers or interpreters, our parser does not produce an abstract syntax tree from a syntactically correct source but a parse tree instead. A parse tree, or concrete syntax tree, is a tree whose nodes are grammar rule applications.Because we need concrete syntax trees (and also, as we shall see, because we want high assurance about the compliance of the parser with respect to the POSIX standard), reusing an existing parser implementation was not an option, as said in the introduction. Our research project required the reimplementation of a static parser from scratch.

Before entering the discussion about implementation choices, let us sum up a list of the main requirements that are implied by the technical difficulties explained in Section 2: (i) lexical analysis must be aware of the parsing context and of some contextual information like the nesting of double quotes and subshell invocations ; (ii) lexical analysis must be defined in terms of token delimitations, not in terms of token (regular) languages recognition; (iii) the syntactic analysis must be able to return the longest syntactically valid prefix of the input ; (iv) the parser must be reentrant ; (v) the parser must forbid certain specific applications of the grammar production rules ; (vi) the parser must be able to switch between the token recognition process and the here-document scanner. In addition to these technical requirements, there is an extra methodological one: the mapping between the POSIX specification and the source code must be as direct as possible.

The tight interaction between the lexer and the parser prevents us from writing our syntactic analyzer following the traditional design found in most textbooks [2], that is a pipeline of a lexer followed by a parser. Hence, we cannot use either the standard interfaces of code generated by Lex and Yacc, because these interfaces have been designed to fit this traditional design. There exists alternative parsing technologies, e.g. scannerless generalized LR parsers or top-down general parsing combinators, that could have offered elegant answers to many of the requirements enumerated previously, but as we will explain in Section 7, we believe that none of them fulfill the entire list of these requirements.

In this situation, one could give up using code generators and fall back to the implementation of a hand-written character-level parser. This is done in Dash for instance: the parser of Dash 0.5.7 is made of 1569 hand-crafted lines

of C code. This parser is hard to understand because it is implemented by low-level mechanisms that are difficult to relate to the high-level specification of the POSIX standard: for example, lexing functions are implemented by means of **goto**s and complex character-level manipulations; the parsing state is encoded using activation and deactivation of bit fields in one global variable; some speculative parsing is done by allowing the parser to read the input tokens several times, etc.

Other implementations, like the parser of Bash, are based on a Yacc grammar extended with some code to work around the specificities of shell parsing. We follow the same approach except on two important points. First, we are stricter than Bash with respect to the POSIX standard: while Bash is using an entirely different grammar from the standard, we literally cut-and-paste the grammar rules of the standard into our implementation to prevent any change in the recognized language. Second, in Bash, the amount of hand-written code that is accompanying the Yacc grammar is far from being negligible. Indeed, we counted approximately 5000 extra lines of C to handle the shell syntactic peculiarities. In comparison, our implementation only needed approximately 1000[3] lines of OCaml to deal with them.

Of course, these numbers should be taken with some precaution since OCaml has a higher abstraction level than C, and since Bash implements a significant extension of the shell language. Nonetheless, we believe that our design choices greatly help in reducing the amount of *ad hoc* code accompanying the Yacc grammar of the POSIX standard. The next sections try to give a glimpse of the key aspects of our parser implementation.

3.1 A modular architecture

Our main design choice is not to give up on modularity. As shown in Figure 3, the architecture of our syntactic analyzer is similar to the common architecture found in textbooks as we clearly separate the lexing phase and the parsing phase in two distinct modules with clear interfaces. Let us now describe the original aspects of this architecture.

As suggested by the illustration, we decompose lexing into two distinct subphases. The first phase called "prelexing" is implementing the "token recognition" process of the POSIX standard. As said earlier, this parsing-independent step classifies the input characters into three categories of "pretokens": operators, words and potentially significant layout characters (newline characters and end-of-input markers). This module is implemented using OCamllex, a lexer generator distributed with the OCaml language. In Section 3.2, we explain which features of this generator we use to get a high-level implementation of lexical conventions close to the informal description of the specification.

[3]The total number of lines of code is 2141, including type definitions, utilities and infrastructure.

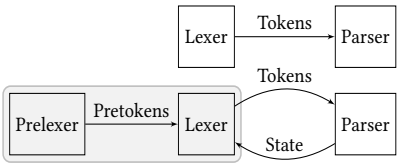

Figure 3. Architectures of syntactic analyzers: at the top of the figure, the standard pipeline commonly found in compilers and interpreters; at the bottom of the figure, the architecture of our parser in which there is a bidirectional communication between the lexer and the parser.

The second phase of lexing is parsing-dependent. As a consequence, a bidirectional communication between the lexer and the parser is needed. On one side, the parser is waiting for a stream of tokens to reconstruct a parse tree. On the other side, the lexer needs some parsing context to promote words to keywords or assignment words, to switch to the lexing mode for here-documents, and to discriminate between the four interpretations of the newline character (see Example 2.2). We manage to implement all these *ad hoc* behaviors using speculative parsing, which is easily implemented thanks to the incremental and purely functional interface produced by the parser generator Menhir [21]. This technique is described in Section 3.3.

3.2 Mutually recursive parametric lexers

The lexer generators of the Lex family are standard tools that compile a pattern matching made of regular expressions into an efficient finite state machine. When a specific regular expression is matched, the generated code triggers the execution of some piece of user-written code. In theory, there is no limitation on the computational expressiveness of lexers generated by Lex since any side-effect on the lexing engine may be performed in the arbitrary code attached to each regular expression. In practice though, it can be difficult to develop complex lexical analyzers with Lex especially when several sublexers must be composed to recognize a single token which is the concatenation of several words of distinct nature (like the word `$BAR" "$(echo $(date))` we encountered earlier) or when they have to deal with nested constructions (like the parenthesized quotations of the shell language, for instance).

OCamllex is the lexer generator of the OCaml programming language. OCamllex extends the specification language of Lex with many features, two of which are exploited in our implementation. First, in OCamllex, a lexer can be defined by a set of mutually recursive entry points. This way, even if a word can be recognized as a concatenation of words from distinct sublanguages, we are not forced to define these sublanguages in the same pattern matching: on the contrary, each category can have a different entry point in the lexer which leads to modular and readable code. Thanks to this

organization of the lexical rules, we were able to separate the lexer into a set of entry points where each entry point refers to a specific part of the POSIX standard. This structure of the source code eases documentation and code reviewing, hence it increases its reliability. Second, each entry point of the lexer can be parameterized by one or several arguments. These arguments are typically used to have the lexer track contextual information along the recognition process. Combined with recursion, these arguments provide to lexers the same expressiveness as deterministic pushdown automata. This extra expressive power of the language allows our lexer to parse nested structures (e.g. parenthesized quotations) even if they are not regular languages. In addition, the parameters of the lexer entry points make it possible for several lexical rules to be factorized out in a single entry point. Last but not least, the prelexer context is flexible enough to maintain the word-level concrete syntax trees mentioned in Section 2.2.3.

3.3 Incremental and purely functional parsing

Yacc-generated parsers usually provide an all-or-nothing interface: when they are run, they either succeed and produce a semantic value, or they fail if a syntax error is detected. Once invoked, these parsers take control and do not give it back unless they have finished their computation. During its execution, a parser calls its lexer to get the next token but the parser does not transmit any information during that call since the lexer is usually independent from parsing.

As we have seen, in the case of the shell language, when the lexer needs to know if a word must be promoted to a keyword or not, it must inspect the parser context to determine if this keyword is an acceptable token at the current position of the input. Therefore, the conventional calling protocol of lexers from parsers is not adapted to this situation.

Fortunately, the Menhir [21] parser generator has been recently extended by François Pottier to produce an incremental interface instead of the conventional all-or-nothing interface. In that new setting, the caller of a parser must manually provide the input information needed by this parser for its next step of execution and the parser gives back the control to its caller after the execution of this single step. Hence, the caller can implement a specific communication protocol between the lexer and the parser. In particular, the state of the parser can be transmitted to the lexer. This protocol between the incremental parser generated by Menhir and the parsing engine is specified by a single type definition:

```
type 'a checkpoint = private
  | InputNeeded of 'a env
  | Shifting of 'a env * 'a env * bool
  | AboutToReduce of 'a env * production
  | HandlingError of 'a env
  | Accepted of 'a
  | Rejected
```

A value of type `'a checkpoint` represents the entire *immutable* state of the parser generated by Menhir. The type parameter `'a` is the type of semantic values produced by a successful parsing. The type `'a env` is the internal state of the parser which roughly speaking contains the stack and the current state of the generated LR pushdown automaton. As specified by this sum type, there are six situations where the incremental parser generated by Menhir interrupts itself to give the control back to the parsing engine: (i)`InputNeeded` means that the parser is waiting for the next token. By giving back the control to the parsing engine and by exposing a parsing state of type `'a env`, the lexer has the opportunity to inspect this parsing state and decide which token to transmit. This is the property we exploit to implement the parsing-dependent lexical analysis. (ii) `Shifting` is returned by the generated parser just before a *shift* action. We do not exploit this particular checkpoint. (iii) `AboutToReduce` is returned just before a *reduce* action. We exploit this checkpoint to implement the treatment of reserved words. (See Section 3.3.1.) (iv) `HandlingError` is returned when a syntax error has just been detected. We do not exploit this checkpoint. (v) `Accepted` is returned when a complete command has been recognized. In that case, if we are not at the end of the input file, we reiterate the parsing process on the remaining input. (vi) `Rejected` is returned when a syntax error has not been recovered by any handler. This parsing process stops on an error message.

Now that the lexer has access to the state of the parser, how can it exploit this state? Must it go into the internals of LR parsing to decipher the meaning of the stack of the pushdown automaton?

Actually, a far simpler answer can be implemented most of the time: the lexer can simply perform some speculative parsing to observationally deduce information about the parsing state. In other words, to determine if a token is compatible with the current parsing state, the lexer just executes the parser with the considered token to check whether it produces a syntax error, or not. If a syntax error is raised, the lexer backtracks to the parsing state that was just before the speculative parsing execution.

If the parsing engine of Menhir were imperative, then the backtracking process required to implement speculative parsing would necessitate some machinery to undo parsing side-effects. Since the parsing engine of Menhir is purely functional we do not need such a machinery: the state of the parser is an explicit immutable value passed to the parsing engine which returns in exchange a fresh new parsing state without modifying the input state. The API to interact with the generated parser is restricted to only two functions:

```
val offer:
  'a checkpoint -> token * position * position
  -> 'a checkpoint
val resume:
  'a checkpoint -> 'a checkpoint
```

The function `offer` is used when the `checkpoint` is exactly of the form `InputNeeded`. In that specific case, the argument is a triple of type `token * position * position` passed to the generated parser.

The function `resume` is used for the other cases to give the control back to the generated parser without transmitting any new input token.

From the programming point of view, backtracking is as cheap as declaring a variable to hold the state to recover it if a speculative parsing goes wrong. From the computational point of view, thanks to sharing, the overhead in terms of space is negligible and the overhead in terms of time is reasonable since we never transmit more than one input token to the parser when we perform such speculative parsing.

Another essential advantage of immutable parsing states is the fact that the parsers generated by Menhir are *reentrant* by construction. As a consequence, multiple instances of our parser can be running at the same time. This property is needed because the prelexer can trigger new instances of the parser to deal with subshell invocations.

Notice that the parsing of subshell invocations are not terminated by a standard end-of-file marker: indeed, they are usually stopped by the closing delimiter of the subshell invocation. For instance, parsing `echo $(date "+%Y%m%d")` requires a subparser to be executed after `$(` and to stop before `)`.

As it is very hard to delimit correctly subshell invocation without parsing their content, this subparser is provided the entire input suffix and it is responsible for finding the end of this subshell invocation by itself.

The input suffix is never syntactically correct. Thus, when a subparser encounters the closing delimiter (the closing parenthesis in our example), it will produce a syntax error.

To tackle this issue, our parser can be run in a special mode named "longest valid prefix". In that mode, the parser returns the longest prefix of the input that is a valid complete command. This feature is similar to backtracking and is as easy as implement thanks to immutable parsing states.

3.3.1 Recognizing reserved words

In this section, we describe our technique to handle the promotion of words to reserved words in a parsing-context sensitive way as well as the handling of promoted words which generate syntax errors. As explained earlier, this technique intensively uses the fact that the parser generated by Menhir is incremental and purely functional.

Let us first show the code of the function which decides whether to promote a word into a reserved word:

```
1  let recognize_reserved_word_if_relevant =
2  fun checkpoint pstart pstop w ->
3  FirstSuccessMonad.(
4      try
5        let kwd = keyword_of_string w in
6        let kwd' = (kwd, pstart, pstop) in
7        if accepted_token checkpoint kwd' then
8          return kwd
9        else
10         raise Not_found
11     with Not_found ->
12       if is_name w then
13         return (NAME (CST.Name w))
14       else
15         return (WORD (CST.Word w))
16 )
```

Line 3 declares that this function is in the `FirstSuccessMonad`, the details of which are not important here. On line 5, a lookup in a table detects if the word `w` is an actual keyword. If not, the exception `Not_found` is raised. Otherwise, the corresponding keyword token `kwd` is passed to the function `accepted_token` to determine if the promotion of `w` to `kwd` does not introduce a syntax error. If the token is not accepted, `Not_found` is raised. The exception handler on line 11 classifies `w` as a name if it falls into a specific lexical category.

The definition of `accepted_token` is:

```
let accepted_token checkpoint token =
  match checkpoint with
  | InputNeeded _ ->
    close (offer checkpoint token)
  | _ ->
    false
```

If the parser is in a state where an input is needed we offer it the token. The resulting new checkpoint is passed to the following recursive function `close` to determine if a syntax error is detected by the parser:

```
let rec close checkpoint = match checkpoint with
| AboutToReduce _ -> close (resume checkpoint)
| Rejected | HandlingError _ -> false
| Accepted _ | InputNeeded _ | Shifting _ -> true
```

Notice that this function always terminates since the recursive call to `close` is done just before a reduction which always consumes some entries at the top of the pushdown automaton stack.

This speculative parsing solves the problem of reserved words only partially. Indeed, if a keyword is used where a `cmd_word` or a `cmd_name` is expected, that is as the command of a `simple_command`, it must be recognized as a reserved word even though it generates a syntax error.

Therefore, the function `recognize_reserved_word_if_relevant` is counterproductive in that case because it will prevent the considered word from being promoted to a reserved word and would fail to detect the expected syntax error. Thanks to the `AboutToReduce` case, we are able to detect *a posteriori* that a word, which has not been promoted to a reserved word, has been used to produce a `cmd_word` or a `cmd_name`:

```
| AboutToReduce (env, production) -> begin try
  if lhs production = X (N N_cmd_word)
  || lhs production = X (N N_cmd_name) then
  match top env with
  | Some (Element (state, v, _, _)) ->
    let analyse_top
    : type a. a symbol * a -> _ = function
    | T T_NAME, Name w when is_reserved_word w
    | T T_WORD, Word w when is_reserved_word w ->
```

```
10        raise ParseError
11      | _ -> assert false
12      in analyse_top (incoming_symbol state, v)
13    | _ -> assert false
14    else raise Not_found
15    with Not_found -> parse (resume checkpoint) end
```

Let us explain this code. First, it is a pattern-matching branch for the case `AboutToReduce`. Conceptually, the argument named `env` represents the stack of the LR pushdown automaton and the argument named `production` is a descriptor for the reduction that is about to happen. On Lines 2 and 3, we first check that this production is indeed a rule whose left-hand-side (the produced nonterminal) is either a `cmd_name` or a `cmd_word`. In that case, we extract the topmost element of the automaton stack: it must be a token `NAME` or `WORD`. We just have to check that the semantic values of these tokens are not reserved words to determine if a syntax error must be raised or if the parsing can go on.

3.4 From the code to the POSIX specification

What makes us believe that our approach to implement the POSIX standard will lead to a parser that can be trusted? Actually, as the specification is informal, it is impossible to prove our code formally correct. We actually do not even claim the absence of bugs in our implementation: this code is far too immature to believe that.

In our opinion, our approach to develop Morbig is likely to lead to a trustworthy implementation because (i) its code is written in such a way that it facilitates code review; (ii) it includes the formal shell grammar of the POSIX as-is; (iii) it has been tested with a rigorous method; (iv) it seems to behave like POSIX-compliant shells.

Code review Comments represent 40% of the Morbig source code. We tried to quote the POSIX specification related to each code fragment so that a code reviewer can evaluate the adequacy between the implementation and its interpretation of the specification. We also document every implementation choice we make and we explain the programming technique used to ease the understanding of the unorthodox part of the program, typically the speculative parsing.

Cut-and-paste of the official shell grammar We commit ourselves to not modifying the official BNF of the grammar despite its incompleteness or the exotic nine side rules described earlier. In our opinion, it is a strength of our approach because this BNF is the most declarative and formal part of the specification, knowing that our generated parser recognizes the same language as this BNF is in our opinion a reason to trust our implementation.

Testsuite Morbig comes with a testsuite which follows the same structure as the specification: for every section of the POSIX standard, we have a directory containing the tests related to that section. At this time, the testsuite is relatively small since it is only made of 149 tests. A code reviewer may still be interested by this testsuite to quickly know if some cornercase of the specification has been tested and, if not, to contribute to the testsuite by the insertion of a test for this cornercase.

Comparison to existing shell implementations To disambiguate several paragraphs of the standard, we have checked that the behavior of Morbig coincides with the behavior of shell implementation which are believed to be POSIX-compliant, typically Dash and Bash (in POSIX mode).

4 Applications

4.1 Shell parsing toolkit

There are two interfaces to the Morbig parser: a Command Line Interface (CLI) and an Application Programming Interface (API).

The CLI of Morbig is an executable program called `morbig`. It takes as input a list of filenames and, for each syntactically correct input file, it produces a JSON file containing a textual representation of the concrete syntax tree of the shell script.

To use the API of Morbig, a programmer writes an OCaml program linked to a library called `libmorbig`. The parsing API of Morbig is contains just one function:

```
1 (** [parse filename] performs the syntactic analysis of
       [filename] and returns a concrete syntax tree if
      [filename] content is syntactically correct.
2     Raises {SyntaxError (pos, msg)} otherwise. *)
3 val parse : string -> CST.complete_command list
```

The API is richer when it comes to analyzing and transforming concrete syntax trees. Indeed, in addition to the type definitions for the concrete syntax trees, the module `CST` defines several classes of *visitors*. The visitor design pattern [10] is an object-oriented programming technique to define a computation over one or several mutually recursive object hierarchies. The next section explains the advantages of defining an analysis with such visitors. In the API, six classes of visitors are provided: `iter` to traverse a CST, `map` to transform a CST into another CST, `reduce` to compute a value by a bottom-up recursive computation on a CST, as well as `iter2`, `map2` and `reduce2` which traverse two input CSTs of similar shapes at the same time. These visitors come for free as we use a preprocessor [20] which automatically generates visitors classes out of type definitions.

4.2 An analyzer for Debian maintainer scripts

The original motivation for the Morbig parser comes from a research project on the development of formal methods for the verification of the so-called *maintainer scripts* present in the Debian GNU/Linux distribution. As a first step of this project, we need a statistical analysis of the corpus in order to know what elements of the shell language and what UNIX commands with which options are mostly used in our corpus. It is easy to implement such an analysis operating on the concrete syntax trees produced by Morbig. Individual

analyzers are written using the visitor design pattern [10] in order to cope with the 108 distinct cases in the type of concrete syntax trees.

5 Current limitations and future work

An important issue is how to validate our parser. Counting the number of scripts that are recognized as being syntactically correct is only a first step since it does not tell us whether the syntax tree constructed by the parser is the correct one. We can imagine several ways how the parser can be validated.

One approach is to write a *pretty-printer* which sequentializes the concrete syntax tree constructed by the parser. The problem is that our parser has dropped part of the layout present in the shell script, in particular information about spaces, and comments. Still, a pretty printer can be useful to a human when verifying the correct action of the parser on a particular case of doubt: this is the technique we used to build our testsuite.

It might also be possible to compare the result obtained by our pretty-printer with the original script after passing both through a simple filter that removes comments and normalizes spaces. Furthermore, a pretty-printing functionality can be used for an automatic smoke test on the complete corpus: the action which consists of parsing a shell script and then pretty-printing it must be idempotent, that is performing it twice on a shell script must yield the same result as performing it once.

Another possible approach is to combine our parser with an interpreter that executes the concrete syntax tree. This way, we can compare the result of executing a script obtained by our interpreter with the result obtained by one of the existing POSIX shell interpreters.

Finally, the scripts of our corpus may not cover all the diversity of the shell scripts that can be found in the nature since they are dedicated to a very specific task, namely package maintainance. We are currently working on a new corpus of 7,5 millions of shell scripts extracted from the archive of The Software Heritage project[7].

6 Availability and Benchmarks

Morbig is Free Software, published under the GPL3 license. It is available at https://github.com/colis-anr/morbig and as an OPAM package.

On a i7-4600U CPU @ 2.10GHz with 4 cores, an SSD hard drive and 8GB of RAM, it takes 41s[4] to parse successfully the 31521 scripts of the corpus (which represents 99% of the 31832 files of the corpus) and to serialize the correspond concrete syntax trees on the disk. The average time to parse a script from the corpus of Debian maintainer scripts is therefore 1.3ms (with a standard deviation which is less than 1% of this duration). The maximum parsing time is 100ms, reached for the `prerm` script of package `w3c-sgml-lib_1.3-1_all` which is 1121 lines long.

[4]Measured with the UNIX `/usr/bin/time` command.

7 Related work

7.1 About the POSIX shell language

Analysis of package maintainer scripts To our knowledge, the only existing attempt to analyze a complete corpus of package maintainer scripts was done in the context of the Mancoosi project [6]. An architecture of a software package installer is proposed that simulates a package installation on a model of the current system in order to detect possible failures. The authors have identified 52 templates which cover completely 64.3% of all the 25.440 maintainer scripts of the Debian Lenny release. These templates are then used as building blocks of a DSL that abstracts maintainer scripts. In this work, a first set of script templates had been extracted from the relevant Debian toolset (debhelper), and then extended by clustering scripts using the same statements [8]. The tool used in this works is geared towards comparing shell scripts with existing snippets of shell scripts, and is based on purely textual comparisons.

Analysis of shell scripts There have been few attempts to formalize the shell. Recently, Greenberg [11] has presented elements of formal semantics of POSIX shell. The work behind Abash [17] contains a formalization of the part of the semantics concerned with variable expansion and word splitting. The Abash tool itself performs abstract interpretation to analyze possible arguments passed by Bash scripts to UNIX commands, and thus to identify security vulnerabilities in Bash scripts. Several tools can spot certain kinds of errors in shell scripts. The checkbashisms [5] script detects usage of Bash-specific syntax in shell scripts, it is based on matching Perl regular expressions against a normalized shell script text. This tool is currently used in Debian as part of the `lintian` package analyzing suite. The tool shellcheck [12] detects error-prone usage of the shell language. This tool is written in Haskell with the parser combinator library Parsec. Therefore, there is no Yacc grammar in the source code to help us determine how far from the POSIX standard the language recognized by shellcheck is. Besides, the tool does not produce intermediate concrete syntax trees which forces the analyses to be done on-the-fly during parsing itself. This approach lacks modularity since the integration of any new analysis requires the modification of the parser source code. Nevertheless, as it is hand-crafted, the parser of shellcheck can keep a fine control on the parsing context: this allows for the generation of very precise and helpful error messages. We plan to use the recent new ability [19] of Menhir to obtain error messages of similar quality.

7.2 About parsing technologies

General parsing frameworks Menhir[21] is based on a conservative extension of LR(1)[16], inspired by Pager's

algorithm[18]: it produces pushdown automata almost as compact as LALR(1) automata without the risk of introducing LALR(1) conflicts. As a consequence, the resulting parsers are both efficient (word recognition has a linear complexity) and reasonable in terms of space usage.

However, the set of LR(1) languages is a strict subset of the set of context-free languages. For context-free languages which are not LR(1), there exist well-known algorithms like Earley's [4, 9], GLR[24], GLL[22] or general parser combinators [15]. These algorithms can base their decision on an arbitrary number of lookups, can cope with ambiguous grammars by generating parse forests instead of parse trees, and generally have a cubic complexity. There also exist parsing algorithms and specifications that go beyond context-free grammars, e.g. reflective grammars [23] or data-dependent grammars [1].

Since the grammar of POSIX shell is ambiguous, one may wonder why we stick to an LR(1) parser instead of choosing a more general parsing framework like the ones cited above. First, as explained in Section 2.4, the POSIX specification embeds a Yacc grammar specification which is annotated by rules that change the semantics of this specification, but only locally by restricting the application of some of the grammar rules. Hence, if we forget the shift/reduce conflicts mentioned in Section 2.4, this leads us to think that the author of the POSIX specification actually have a subset of an LR(1) grammar in mind. Being able to use an LR(1) parser generator to parse the POSIX shell language is in our opinion an indication that this belief is true. Second, even though we need to implement some form of speculative parsing to efficiently decide if a word can be promoted to a reserved word, the level of non-determinism required to implement this mechanism is quite light. Indeed, it suffices to exploit the purely functional state of our parser to implement a backtracking point just before looking at one or two new tokens to decide if the context is valid for the promotion, or not. This machinery is immediately available with the interruptible and purely functional LR(1) parsers produced by Menhir. In our opinion, the inherent complexity of generalized parsing frameworks is not justified in that context.

Scannerless parsing Many legacy languages (e.g. PL/1, COBOL, FORTRAN, R, ...) enjoy a syntax which is incompatible with the traditional separation between lexical analysis and syntactic analysis. Indeed, when lexical conventions (typically the recognition of reserved words) interact in a nontrivial way with the parsing context, the distinction between lexing and parsing fades away. For this reason, it can perfectly make sense to implement the lexical conventions in terms of context-free grammar rules and to mix them with the language grammar. With some adjustments of the GLR parsing algorithm to include a longest-match strategy and with the introduction of specification mechanisms to declare layout conventions efficiently, the ASF+SDF project[25] has been able to offer a declarative language to specify modular scannerless grammar[26] specifications for many legacy languages with parsing-dependent lexical conventions.

Unfortunately, as said in Section 2.1.1, the lexical conventions of POSIX shell are not only parsing-dependent but also specified in a "negative way": POSIX defines token recognition by characterizing how tokens are delimited, not how they are recognized. Besides, as shown in Section 2.1.2, the layout conventions of POSIX shell, especially the handling of newline characters, are unconventional, hence they hardly match the use cases of existing scannerless tools. Finally, lexical conventions depend not only on the parsing context but also on the nesting context as explained in Section 2.1.4. For all these reasons, we are unable to determine how these unconventional lexical rules could be expressed following the scannerless approach. More generally, it is unclear to us if the expressivity of ASF+SDF specifications is sufficient to handle the POSIX shell language without any extra code written in a general purpose programming language.

Schrödinger's tokens Schrödinger's tokens[3] is a technique to handle parsing-dependent lexical conventions by means of a superposition of several states on a single lexeme produced by the lexical analysis. This superposition allows to delay to parsing time the actual interpretation of an input string while preserving the separation between the scanner and the parser. This technique only requires minimal modification to parsing engines. Morbig's promotion of words to reserved words follows a similar path: the prelexer produces pretokens which are similar to Schrödinger's tokens since they enjoy several potential interpretations at parsing time. The actual decision about which is the right interpretation of these pretokens as valid grammar tokens is deferred to the lexer and obtained by speculative parsing. No modification of Menhir's parsing engine was required thanks to the incremental interface of the parsers produced by Menhir: the promotion code can be written on top of this interface.

8 Conclusion

Statically parsing shell scripts is notoriously difficult, due to the fact that the shell language was not designed with static analysis in mind. Nevertheless, we found ourselves in need of a tool that allows us to easily perform a number of different statistical analyses on a large number of scripts. We have written a parser that maintains a high level of modularity, despite the fact that the syntactic analysis of shell scripts requires an interaction between lexing and parsing that defies traditional approach.

Acknowledgment

We are grateful to the reviewers of the different versions of this paper. Their comments helped us to improve the paper as well as the implementation. We also thank Patricio Pelliccione and Davide Di Ruscio for discussion of their work

done in the context of the Mancoosi project. This work has been supported by the French national research project ANR CoLiS (contract ANR-15-CE25-0001).

References

[1] A. Afroozeh and A. Izmaylova. Iguana: a practical data-dependent parsing framework. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 267–268. ACM, 2016.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] J. Aycock and R. N. Horspool. Schrödinger's token. *Softw., Pract. Exper.*, 31:803–814, 2001.

[4] J. Aycock and R. N. Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[5] R. Braakman, J. Rodin, J. Gilbey, and M. Hobley. checkbashisms. https://sourceforge.net/projects/checkbaskisms/, Nov. 2015.

[6] R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming*, 76(12):1144–1160, 2011.

[7] R. Di Cosmo and S. Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, 2017.

[8] D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Towards maintainer script modernization in FOSS distributions. In *IWOCE 2009: International Workshop on Open Component Ecosystem*, pages 11–20. ACM, 2009.

[9] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[10] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[11] M. Greenberg. Understanding the POSIX shell as a programming language. In *Off the Beaten Track 2017*, Paris, France, Jan. 2017.

[12] V. Holen. shellcheck. https://github.com/koalaman/shellcheck, 2015.

[13] IEEE and The Open Group. The open group base specifications issue 7. http://www.unix.org/version3/online.html, 2016.

[14] IEEE and The Open Group. The open group base specifications issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[15] A. Izmaylova, A. Afroozeh, and T. v. d. Storm. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–12. ACM, 2016.

[16] D. E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[17] K. Mazurak and S. Zdancewic. ABASH: finding bugs in bash scripts. In *PLAS07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 105–114, San Diego, CA, USA, June 2007.

[18] D. Pager. A practical general method for constructing LR (k) parsers. *Acta Informatica*, 7(3):249–268, 1977.

[19] F. Pottier. Reachability and error diagnosis in LR(1) parsers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 88–98, 2016.

[20] F. Pottier. Visitors. http://gallium.inria.fr/~fpottier/visitors/manual.pdf, 2017.

[21] F. Pottier and Y. Régis-Gianas. The Menhir parser generator. *See: http://gallium. inria. fr/fpottier/menhir*.

[22] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

[23] P. Stansifer and M. Wand. Parsing reflective grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, page 10. ACM, 2011.

[24] M. Tomita and S.-K. Ng. The generalized LR parsing algorithm. In *Generalized LR parsing*, pages 1–16. Springer, 1991.

[25] M. G. van den Brand, A. van Deursen, J. Heering, H. De Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, et al. The asf+sdf meta-environment: A component-based language development environment. In *International Conference on Compiler Construction*, pages 365–370. Springer, 2001.

[26] E. Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.