Journal of Computer Languages 57 (2020) 100944

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.editorialmanager.com/cola/default.aspx

View Points

MORBIG: A Static parser for POSIX shell

Yann Régis-Gianas*,[a,b], Nicolas Jeannerod[a], Ralf Treinen[a]

[a] IRIF, Université de Paris, CNRS, Paris, France
[b] Inria, π.r², Paris, France

ARTICLE INFO

Keywords:
Parsing
POSIX
Shell
Scripts

MSC:
00-01
99-00

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 and lexical analysis depends on the parsing context and the evaluation context. 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 that 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. Its implementation crucially relies on the purity and incrementality of LR(1) parsers generated by MENHIR, a parser generator for OCAML.

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 that are executed as part of the process of installing, removing or upgrading software packages. The Debian GNU/Linux distribution, for instance, contains 31,582[1] of these so-called maintainer scripts, 31,330 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. For instance, trivial mistakes in the installation scripts of `bumblebee` and `steam` [1] led in both cases to removal of unrelated files. One of the authors of this paper is responsible for having introduced a similar bug in the Debian package `cmigrep` [2]. An ongoing research project[2] aims at using formal verification tools for analyzing shell scripts and has so far found more than 150 bugs [3].

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 using 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. By definition, a static parser cannot evaluate scripts to parse them, so it must correctly simulate evaluation at the parsing level when that is feasible, or reject scripts whose syntactic structure cannot be decided statically. But this is only the start, the syntax of POSIX shell is

* Corresponding author at: IRIF, Université de Paris, CNRS, Paris, France.
E-mail address: yrg@irif.fr (Y. Régis-Gianas).

[1] `unstable`, `amd64` architecture, as of 18/03/2019
[2] CoLiS, "Correctness of Linux Scripts", https://colis.irif.fr

https://doi.org/10.1016/j.cola.2020.100944
Received 3 April 2019; Received in revised form 1 November 2019; Accepted 28 January 2020
Available online 18 February 2020

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. 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 maintain such a parser in the long run? If the parser's source code is too low-level and too complex, any modification in the standard will be difficult to take into account correctly and is likely to introduce a bug. The parser is an essential part of our tool chain, if the parser produces incorrect syntax trees then all formal analysis based on it will be worthless.

A standard technique to implement syntactic analyzers consists in writing a high-level specification, which is automatically converted into code. Using code generators is a software engineering practice which allows the programmer 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 that reduces the probability of programming errors; and (ii) the code generators are based on well-known algorithms like LR-parsing that have been studied for over fifty years [4], which reduces the probability that the generated code is faulty. 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:

1. 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 we had to fulfill to implement a static parser for this language.
2. 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 the mapping between the specification and the implementation more explicit.
3. 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.

The Morbig parser described in this paper is a *static* parser for a subset of the POSIX shell language, that is it constructs a concrete syntax tree of a complete script without evaluating constructs of the language, with the notable exception of the `alias` builtin as explained in Section 2.3. The only limitations of Morbig are that it cannot handle shell constructs that are inherently dynamic in nature: the `eval` builtin, unrestricted use of the `alias` builtin (only its use outside of control structures is permitted), and premature termination of a script by an `exit` with trailing garbage in the file. These restrictions are justified by the static nature of our parser.

Plan The rest of this article is organized as follows. We start by discussing the difficulties that we have encountered with the syntax of POSIX shell in Section 2. Section 3 explains how we could maintain a modular, though nonstandard, design of our syntactic analyzer despite the pitfalls of the shell language. In Section 4 we present some applications that we have built on top of our library. We conclude with some current limitations of our tool and plans for future work in Section 5, give some benchmarks in Section 6, and compare our approach to related work in Section 7.

Description of the additional material This paper is an extended version of a conference paper published at SLE'2018 [5]. The paper presentation is globally improved and it covers the following additional material:

- We updated the paper to the latest version of the POSIX standard [6].
- Several new pitfalls of POSIX shell are described in Section 2 (the exotic semantics of alias expansions, the proper delimitation of reserved words, the inner structure of words, the complete description of the specific grammar rules that annotate the POSIX shell grammar, etc.).
- Several new implementation aspects of Morbig are detailed in Section 3, including the parsing of word components, the parsing of bracket regular expressions, and the internal state of the prelexer.
- The paper now includes the description of several applications of Morbig in Section 4: a *linter* for shell scripts, a statistical analyzer of a corpus of shell scripts, a C library to allow for interoperability with other languages than OCaml, and an abstract syntax tree for shell scripts.
- Finally, the paper reports in Section 5 on an experiment that compares the behavior of Morbig and Dash on a testsuite of almost 7.5 million shell scripts obtained from the *Software Heritage* archive [7].

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 [6].

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 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 tokens from a stream of input characters by recognizing tokens as meaningful character subsequences and by ignoring insignificant character subsequences such as layout; and (ii) *parsing* which synthesizes a parse tree from the stream of tokens according to some formal grammar.

In this section, we describe several aspects that 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 most programming languages, 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 [6], unfortunately without using any regular expressions. Most languages use regular expressions with a longest-match strategy to recognize the next token in the input

"positively" that is, through the characters that compose the token. On the contrary, the specification of the shell language is formulated in a "negative way" by specifying the characters that are not in the tokens. In other words, token recognition is based on a state machine that explains how tokens must be *delimited* in the input. Token recognition also explains how these delimited chunks must be classified into two categories: words and operators.

The state machine that recognizes the tokens is unfortunately not a finite automaton. It is almost as powerful as a pushdown automaton since it must be able to recognize nested quotations. The complexity of nesting we have to cope with here goes beyond what is known from other languages like nested comments in Java, as one can see in the following example.

Example 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 the token `x$BAR" "$(echo $(date))`. Notice that the last token contains nested quotations of the form `$(.$(.))` 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. In many languages, 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. Since a token is defined negatively, that is by what may follow it instead of the longest match of a regular expression, one can produce a token only after one has at least started to read the following token. For this reason, the tokenizer must be able to transmit to the parser a list of tokens, instead of a single token at a time as it is common practice.

Example 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 is 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 3.

```
1 ls foo#bar
```

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

2.1.3. Character escaping

String literals of most programming languages may contain escaping sequences to let the programmer use the entire character set including string delimiters themselves. The backslash character typically introduces such escaping sequence as in `"\""` to insert a double quote or in `"\\"` to insert a backslash. In these cases, 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 4 (*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 occurrence 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.

2.1.4. 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 5 (*Finding closing parenthesis is difficult***).**

```
1 echo ‘echo \‘echo $(echo foo\\\\\\)\‘‘
```

On this command, determining if the right parenthesis is ending the subshell requires deciding if the parenthesis is escaped or not. However, as explained in the previous section, this analysis is non trivial and more or less requires to perform a complete lexical analysis of the input.

2.1.5. Here-documents

Here-documents are chunks of text embedded in a shell script, just like string literals. Here-documents are commonly used to implement some form of template-based generation of files (since they may contain variables). However, their lexing is different from the one for strings. For this reason, depending on the parsing context, the lexer must switch to a special mode to deal with here-documents. 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  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'
```

Fig. 1. The tokens of the shell language grammar.

token, with the possible exception of quotations which may still be recognized exactly as in the normal lexing mode. As a result, the lexer has a dedicated structure which allows it to track the end-markers and to interpret characters in a specific way, e.g., newline characters are not interpreted at all whereas they have more complex interpretations in the standard lexing mode. (See Section 2.1.2.)

Example 6 (*Here-documents*).

```
1 cat > notifications << EOF
2 Hi $USER!
3 EOF
4 cat > toJohn << EOF1 ; cat > toJane << EOF2
5 Hi "John"!
6 EOF1
7 Hi Jane!
8 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 4 is fed with the content of line 5 while the second `cat` command of line 4 is fed with the content of line 7. This example with two successive here-documents illustrates the non-locality of the lexing process of here-documents: 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. Here-documents are merely recognized as double-quoted words, except that double-quotes keep their literal meaning.

To complete this description, notice that if the delimiter is quoted, the here-document is processed as a literal surrounded by single quotes.

Example 7 (*Here-documents*).

```
1 cat << 'EOF'
2 Hi $USER!
3 EOF
```

This example evaluates into `Hi $USER!`.

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 Fig. 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 under *ad hoc* conditions; 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 3.2.35 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 8 (*Promotion of a word to an assignment*).

```
1 CC=gcc make
2 make CC=cc
3 "./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 3, 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

The Rule 1 of the Shell Grammar Rules described in Section 2.10.2 of the standard demands that a word is promoted to a reserved word if the parser state is expecting this reserved word at the current point of the input. If a word that is a potential reserved word is located where a reserved word is not expected, it is not promoted and interpreted as any other word.

Example 9 (*Promotion of a word to a reserved word*).

```
1 for do in for do in echo done; do echo $do; done
```

In that example, the first occurrence of `do` as well as the words between the first occurrence of `in` and the first semicolon are *not* promoted to reserved words while the other occurrences of `for`, `do`, `in` and `done` are.

In addition to this promotion rule, some reserved words can never appear in the position of a command. This is therefore an exception to the previous rule.

Example 10 (*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.

To complete the picture, the grammar given in the POSIX standard does not define completely the shell language. In addition to the rules, the parser must also take into account constraints as expressed in the "note" in the description of Rule 1 given in Section 2.10.2 of the standard:

> This rule also implies that reserved words are not 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 is properly delimited by the user, and does not attempt to reflect that requirement directly.

Example 11 (*The grammar is too flexible*). The following program is valid if we follow solely the grammar:

```
1 if foo then echo bar fi
```

but it must be rejected because `then` and `fi` are not "properly delimited" by the user.

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 12 (*A word can have many components*).

```
1 f=~linus/"$(echo foo)${x:-bar}"'baz'[a-b]*
```

This script is a single word read as an `ASSIGNMENT_WORD` by the grammar. The right-hand-side of this assignment is a sequence starting with a so-called "tilde-prefix", followed by 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. The double-quoted word is completed with a literal `baz`, a bracket range expression and pattern-matching operator matching all words.

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 that 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 on the fly while parsing allows us to implement a simple form of alias expansion without compromising decidability. This simple strategy is sufficient to parse the 31,330 POSIX shell scripts in the corpus of Debian maintainer scripts (see Section 4.3), which contained anyway only 2 occurrences of the `alias` command.

Example 13 (*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.

Example 14 (*Tractable usage of alias*).

```
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.

Example 15 (*Non standard semantics of alias expansion*). In the following example, notice that the right-hand-side of the alias definition ends with a space:

```
1 alias x="echo "
2 x x y x
```

```
1  program:
2    linebreak complete_commands linebreak | linebreak;
3  complete_commands:
4    complete_commands newline_list complete_command
5  | complete_command;
6  complete_command:
7    list separator_op | list;
8  list:
9    list separator_op and_or | and_or;
10 and_or:
11   pipeline
12 | and_or AND_IF linebreak pipeline
13 | and_or OR_IF  linebreak pipeline;
14 pipeline:
15   pipe_sequence | Bang pipe_sequence;
16 pipe_sequence:
17   command | pipe_sequence '|' linebreak command;
18 command:
19   simple_command | compound_command
20 | compound_command redirect_list | function_definition;
21 compound_command:
22   brace_group | subshell | for_clause | case_clause
23 | if_clause | while_clause | until_clause;
24 subshell:
25  '(' compound_list ')';
26 compound_list:
27   linebreak term | linebreak term separator;
28 term:
29   term separator and_or | and_or;
30 while_clause:
31   While compound_list do_group;
32 do_group:
33   Do compound_list Done /* Apply rule 6 */;
34 simple_command:
35   cmd_prefix cmd_word cmd_suffix
36 | cmd_prefix cmd_word
37 | cmd_prefix
38 | cmd_name cmd_suffix
39 | cmd_name;
40 cmd_name:
41   WORD /* Apply rule 7a */;
42 cmd_word:
43   WORD /* Apply rule 7b */;
44 newline_list:
45   NEWLINE | newline_list NEWLINE;
46 linebreak:
47   newline_list | /* empty */;
48 separator_op:
49   '&' | ';';
50 separator:
51   separator_op linebreak | newline_list;
52 sequential_sep:
53   ';' linebreak | newline_list;
54
55 // The rules for the following nonterminals are elided:
56 // for_clause, name, in, wordlist, case_clause,
57 // case_list_ns, case_list, case_item_ns, case_item,
58 // pattern if_clause, else_part, until_clause,
59 // function_definition, function_body, fname,
60 // brace_group, cmd_prefix, cmd_suffix, redirect_list,
61 // io_redirect, io_file, filename, io_here and here_end.
```

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

This space affects the semantics of alias expansion by enabling an *expansion chain*: in that case, the alias expansion applies not only to the command name but also to every word that follows it as long as this word is itself subject to alias expansion. As a consequence, after alias expansion, this script is rewritten as:

```
1 echo echo y x
```

This mechanism forces the parser to maintain a state to determine if the alias expander has triggered such a chain of alias expansions.

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.

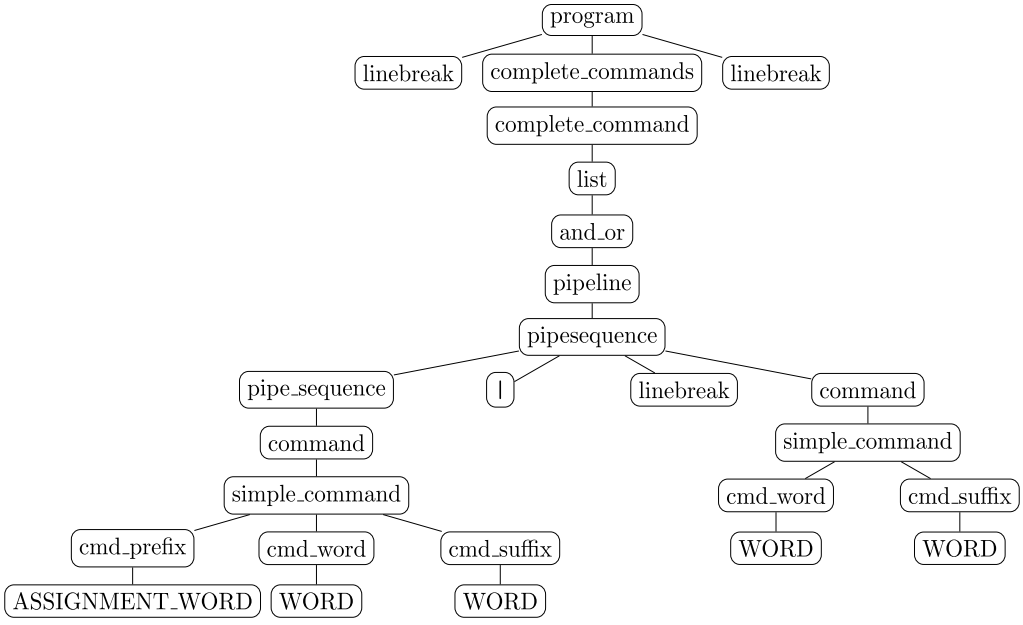

Fig. 3. Parse tree for `CC=gcc make all | grep 'error'`.

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 Fig. 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 because the specification is annotated with nine special rules that are not directly expressible in terms of YACC's parsing mechanisms. In a previous version of the standard [8], the grammar also contained LR(1) conflicts. These conflicts have fortunately been fixed in the latest POSIX specification.

2.4.1. Special rules

The nine special rules of the grammar are actually the place where the parsing-dependent lexical conventions are explained. We briefly review all the rules and explain their implications.

Rule 1 We already discussed Rule 1 about promotion to reserved words in Section 2.2.2.

Rule 2 This rule restricts the words that can be used as the target filename for a redirection. This restriction is related to the expansion of this word, hence it is a semantic criterion that should not be tackled at parsing stage as far as we understand it.

Rule 3 This rule is expressed as follows:

> [Redirection from here-document]
> Quote removal shall be applied to the word to determine the delimiter that is used to find the end of the here-document that begins after the next ⟨newline⟩.

This rule has an impact on the lexing of here-documents as it defines how the end marker of an here-document must be built from the word producing the non terminal `end_here`. If the quote removal of this word is not the identity (i.e. if the word is quoted), the lexing of the here-document is similar to a word enclosed between single quotes. Otherwise, the lexing is similar to a word enclosed between double quotes.

Rule 4 Here is an excerpt from the standard describing Rule 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.

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`.

Rule 5 This rule annotates the following grammar rules:

```
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 */;
```

Lexically speaking, the tokens of the form `NAME` are a subset of the words. Therefore, the lexical analysis cannot produce `NAME`s as it already produces `WORD`s. Rule 5 can be seen as a parsing-dependent lexical rule embedded in the grammar whose role is to reinterpret a token `WORD` as a `NAME` in appropriate parsing contexts (here, after a `for`):

> *[NAME in for]*
> *When the TOKEN meets the requirements for a name (see XBD Name), the token identifier NAME shall result. Otherwise, the token WORD shall*

be returned.

Rule 6 This rule is expressed as follows in the standard:

[Third word of for and case]
[case only] When the TOKEN is exactly the reserved word in, the token identifier for in shall result. Otherwise, the token WORD shall be returned.
[for only] When the TOKEN is exactly the reserved word in or do, the token identifier for in or do shall result, respectively. Otherwise, the token WORD shall be returned.
(For a. and b.: As indicated in the grammar, a linebreak precedes the tokens in and do. If ⟨newline⟩ characters are present at the indicated location, it is the token after them that is treated in this fashion.

This rule defines cases where the keywords `in` and `do` are properly located in the input, even though that may not be properly delimited by a semicolon or a linebreak.

Rule 7 This rule is similar to the Rule 5 except that it focuses on the valid promotion of a word to an assignment word.

Rule 8 This rule restricts the valid names for function identifiers to exclude reserved words.

Rule 9 This rule states that the function bodies are not subject to expansion and assignment. Like Rule 2, we consider this rule to be related to semantics, not parsing.

2.4.2. LR(1) Conflicts

Our LR(1) parser generator has detected five shift/reduce conflicts in the YACC grammar of a previous version of the standard [8]. Even though these conflicts are now fixed in the latest specification of POSIX shell, we keep the description of these conflicts for the sake of history and since it can still help shell implementers.

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:

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

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

```
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
```

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.

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 4.3). 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:

1. lexical analysis must be aware of the parsing context and of some contextual information like the nesting of double quotes and subshell invocations;
2. lexical analysis must be defined in terms of token delimitations, not in terms of token (regular) languages recognition;
3. the syntactic analysis must be able to return the longest syntactically valid prefix of the input;
4. the parser must be reentrant;
5. the parser must forbid certain specific applications of the grammar production rules; and
6. 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 [9], 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 exist alternative parsing technologies, e.g. scannerless generalized LR parsers or topdown 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 fulfills 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.

[3] The total number of lines of code is 2622, including type definitions, utilities and infrastructure.

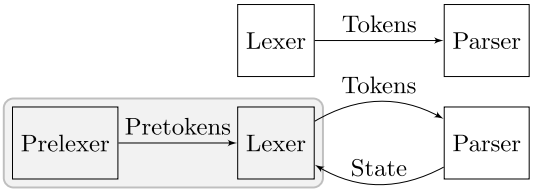

Fig. 4. Architectures of syntactic analyzers: at the top of the figure, the standard pipeline commonly found in compilers and interpreters; at the bottom of the figure, the architecture of our parser in which there is a bidirectional communication between the lexer and the parser.

The 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 Fig. 4, 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.

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 to 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), etc. 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 [10]. This technique is described in Section 3.3.

3.2. Mutually recursive parametric lexers

The lexer generators of the LEX family are standard tools which compile an ordered list 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 that 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 more 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 (we come back on this aspect in Section 3.4.1).

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 [10] parser generator has been recently extended 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:

```
1 val offer:
2   'a checkpoint -> token * position * position
3   -> 'a checkpoint
4 val resume:
5   'a checkpoint -> 'a checkpoint
```

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:

- `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. (See `accepted_token`in Section 3.3.1.)
- `AboutToReduce` is returned just before a *reduce*action. We exploit this checkpoint to implement a specific aspect of the treatment of reserved words. (See Section 3.3.1.)
- `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.
- `Rejected` is returned when a syntax error has not been recovered by any handler. This parsing process stops on an error message.
- `Shifting` is returned by the generated parser just before a *shift* action. `HandlingError` is returned when a syntax error has just been detected. We do not exploit these particular checkpoints.

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 often be implemented: 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 active 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:

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

This function takes a boolean `well_delimited` that is `true` if the context is sufficient to delimit a keyword, a checkpoint corresponding to the current state of the parser, some positions `pstart` and `pstop` as well as a string `w`. The function `valid_token` is a predicate that accepts a keyword `kwd` if it is compatible with the current parsing state and if it is well delimited. Then, line 7 declares that this function is in the `FirstSuccessMonad`, the details of which are not important here [4]: it simply allows us to first try to promote `w` as a keyword if it is

```
1  let recognize_reserved_word_if_relevant =
2    fun well_delimited checkpoint (_, pstart, pstop) w ->
3    let valid_token kwd =
4      accepted_token checkpoint (kwd, pstart, pstop) <> Wrong
5      && must_be_well_delimited well_delimited kwd
6    in
7    FirstSuccessMonad.(
8      (keyword_of_string w >>= fun kwd ->
9       return_if (valid_token kwd) kwd
10     ) +> (return_if (Name.is_name w) (NAME (CST.Name w)))
11   )
```

The function `offer` is used when the `checkpoint` is 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:

[4] Technically, this monad is the Maybe monad with a left-biaised choice operator.

```
type word = Word of string * word_cst

and word_cst = word_component list

and word_component =
  | WordSubshell of subshell_kind * program located
  | WordName of string
  | WordAssignmentWord of assignment_word
  | WordDoubleQuoted of word
  | WordSingleQuoted of word
  | WordLiteral of string
  | WordVariable of variable
  | WordGlobAll
  | WordGlobAny
  | WordReBracketExpression of bracket_expression
  | WordEmpty

and variable =
  | VariableAtom of string * variable_attribute

and variable_attribute =
  | NoAttribute
  | ParameterLength of word
  | UseDefaultValues of string * word
  | AssignDefaultValues of string * word
  | IndicateErrorifNullorUnset of string * word
  | UseAlternativeValue of string * word
  | RemoveSmallestSuffixPattern of word
  | RemoveLargestSuffixPattern of word
  | RemoveSmallestPrefixPattern of word
  | RemoveLargestPrefixPattern of word

and subshell_kind =
  | SubShellKindBackQuote
  | SubShellKindParentheses

and name = Name of string

and assignment_word = name * word
```

Fig. 5. The concrete syntax trees for words.

indeed a valid keyword, or else turn it into a name if it is a valid name. Otherwise, this computation fails.

The definition of `accepted_token` is:

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

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:

```
bracket_expression : '[' matching_list ']'
                   | '[' nonmatching_list ']';
matching_list  : bracket_list;
nonmatching_list : '^' bracket_list;
bracket_list   : follow_list
                   | follow_list '-';
follow_list    :            expression_term
                   | follow_list expression_term;
expression_term : single_expression
                   | range_expression;
single_expression : end_range
                   | character_class
                   | equivalence_class;
range_expression : start_range end_range
                   | start_range '-';
start_range    : end_range '-';
end_range      : COLL_ELEM_SINGLE
                   | collating_symbol;
collating_symbol : Open_dot COLL_ELEM_SINGLE Dot_close
                   | Open_dot COLL_ELEM_MULTI Dot_close
                   | Open_dot META_CHAR Dot_close;
equivalence_class : Open_equal COLL_ELEM_SINGLE Equal_close
                   | Open_equal COLL_ELEM_MULTI Equal_close;
character_class : Open_colon class_name Colon_close;
```

Fig. 6. Grammar for bracket regular expressions.

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) ->
  let nt = nonterminal_of_production production in
  let rec reject_cmd_words_which_are_reserved_words () =
    if is_cmd () && top_is_keyword () then parse_error ()
  and is_cmd () =
    nt = AnyN N_cmd_word || nt = AnyN N_cmd_name
  and top_is_keyword () =
    on_top_symbol env { perform }
  and perform : type a. a symbol * a -> _ = function
    | N N_word, Word (w, _) -> is_reserved_word w
    | _ -> false
  in
  ...
```

```
** Conflict (shift/reduce) in state 35.
** Token involved: MINUS
** This state is reached from bracket_expression after reading:

LBRACKET end_range

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations
   begin to differ.)

bracket_expression
LBRACKET matching_list RBRACKET EOF
         bracket_list
         (?)

** In state 35, looking ahead at MINUS, reducing production
** single_expression -> end_range
** is permitted because of the following sub-derivation:

follow_list MINUS // lookahead token appears
expression_term // lookahead token is inherited
single_expression // lookahead token is inherited
end_range .

** In state 35, looking ahead at MINUS, shifting is permitted
** because of the following sub-derivation:

follow_list
expression_term
range_expression
start_range end_range
end_range . MINUS
```

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 Line 4 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` by calling `is_cmd`. In that case, `top_is_keyword` extracts 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.

3.4. Parsing of words

As explained in Section 2.2.3, even though words are tokens from the perspective of the POSIX shell grammar, words also have an internal syntactic structure of their own. Therefore, words cannot simply be represented by strings. Morbig enriches the signature of the concrete syntax trees of the shell language grammar with a sublanguage for words as specified by the datatype definition of Fig. 5.

The construction of this specific concrete syntax trees for words significantly increases the complexity of the parser for two reasons: (i) these syntax trees must be built along the lexical analysis (this is not standard since lexical analysis usually produces a sequence of tokens, not syntax trees) ; and (ii) the sub-language of regular bracket expressions requires an LR parser whose grammar specification is given in a dedicated section of the POSIX standard (namely Section 9.3.5).

3.4.1. Production of word concrete syntax trees by the prelexer

To attach concrete syntax trees to words, the prelexer maintains a stack of concrete syntax trees of type `word_component` as defined in Fig. 5. A dedicated module named `PrelexerState` provides the operations to incrementally build syntax trees, possibly one character at a time.

For instance, if a (non quoted) double quote is scanned, the prelexer state enters a mode where all subsequent characters are stored in a buffer until the closing double quotes is reached. At this point, the contents `w` of the buffer is used to construct a word concrete syntax tree of the form `WordDoubleQuoted w`. Notice that the buffer may recursively contain other words, as in the word `"$X`date`"` where a `WordVariable` and a `WordSubshell` are contained by the `WordDoubleQuoted`.

During our tests, we found many shell scripts with large literals (e.g. binaries embedded in scripts as here-documents). To deal with them, we optimized the data structure used for bufferization for the case of long sequences of character insertions.

3.4.2. LR Parser dedicated to bracket regular expressions

Bracket Regular Expressions like `[a-z]` or `[!A-]` are specified by a grammar given in the YACC format, reproduced in Fig. 6.

There are no specific annotations of the rule as in the grammar for the shell language, which simplifies its interpretation. However, our tool detected one shift/reduce conflict in that grammar:

Table 1
Comparison of MORBIG and DASH on the whole corpus from Software Heritage. The percentages are in function of the total number of scripts.

MORBIG \ DASH	All	Accepted	Rejected
All	7,436,215 (100%)	5,981,054 (80%)	1,455,161 (20%)
Accepted	5,609,366 (75%)	5,607,331 (75%)	2035 (< 1%)
Rejected	1,826,849 (25%)	373,723 (5%)	1,453,126 (20%)

```
(** {3 Parsing shell scripts} *)

(** [parse_file filename] performs the syntactic analysis of
   [filename] and returns a concrete syntax tree if [filename] content
   is syntactically correct.

   Raises exceptions from {!Errors}. *)
val parse_file: string -> CST.program

(** [parse_string filename content] is similar to [parse_file] except
   the script source code is provided as a string. *)
val parse_string: string -> string -> CST.program

(** {3 Serialization of CST} *)

(** [load_binary_cst cin] retrieves a serialized CST from
   input_channel [cin]. *)
val load_binary_cst: in_channel -> CST.program

(** [save_binary_cst cout cst] stores a serialized [cst] in [cout]. *)
val save_binary_cst: out_channel -> CST.program -> unit

(** [load_json_cst cin] retrieves a CST in JSON format from
   input_channel [cin]. *)
val load_json_cst: in_channel -> CST.program

(** [save_json_cst cout cst] stores a [cst] using JSON format in
   [cout]. *)
val save_json_cst: out_channel -> CST.program -> unit

(** [save_dot_cst cout cst] stores a [cst] using DOT format in
   [cout]. *)
val save_dot_cst: out_channel -> CST.program -> unit

(** {3 CST helpers} *)

(** [on_located f] applies [f] on a located value, preserving its
   location. *)
val on_located : ('a -> 'b) -> 'a CST.located -> 'b

(** [start_of_position p] returns the beginning of a position [p]. *)
val start_of_position : CST.position -> Lexing.position

(** [end_of_position p] returns the end of a position [p]. *)
val end_of_position : CST.position -> Lexing.position

(** [filename_of_position p] returns the filename of a position
   [p]. *)
val filename_of_position : CST.position -> string

(** [string_of_lexing_position p] returns a human-readable
   representation of the lexing position [p], using a format
   recognized by Emacs, and other decent editors. *)
val string_of_lexing_position : Lexing.position -> string

(** {3 POSIX related helpers} *)

(** [remove_quotes s] yields a copy of string [s], with all quotes
   removed as described in the POSIX specification.*)
val remove_quotes : string -> string
```

Fig. 7. OCaml API.

The conflict comes from the fact that the LR(1) parser does not know if a `MINUS` terminates an `end_range` or not. The sensible choice is the second part of this alternative which amounts to prefer shift over reduce.

Just like with the shell language, the lexical analysis of bracket regular expressions is parsing dependent: typically, the syntactic interpretation of the character '-' varies. In `[!A-]`, the first occurrence of dash is a token used by the grammar to separate the start and the end of a range, while the second occurrence must be recognized as a token named `COLL_ELEM_SINGLE`, which includes all characters that can be matched by bracket regular expression. Morbig again makes use of an incremental parser generated by Menhir to deal with this parsing-dependent tokenization as with the global grammar of POSIX shell.

3.5. 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.

To improve our chance to converge to a trustworthy implementation, MORBIG development follows several guidelines: (i) its code is

```
type name = string
and descr = int

and attribute =
  | NoAttribute
  | ParameterLength of word
  | UseDefaultValues of word
  | AssignDefaultValues of word
  | IndicateErrorifNullorUnset of word
  | UseAlternativeValue of word
  | RemoveSmallestSuffixPattern of word
  | RemoveLargestSuffixPattern of word
  | RemoveSmallestPrefixPattern of word
  | RemoveLargestPrefixPattern of word

and word_component =
  | WLiteral of string
  | WDoubleQuoted of word
  | WVariable of name * attribute
  | WSubshell of program
  | WGlobAll
  | WGlobAny
  | WBracketExpression of Morbig.CST.bracket_expression

and word = word_component list
and word' = word located

and pattern = word list
and pattern' = pattern located

and assignment = name * word
and assignment' = assignment located

and program = command' list

and command =
  | Simple of assignment' list * word' list
  | Async of command
  | Seq of command' * command'
  | And of command' * command'
  | Or of command' * command'
  | Not of command'
  | Pipe of command' * command'
  | Subshell of command'
  | For of name * word list option * command'
  | Case of word * case_item' list
  | If of command' * command' * command' option
  | While of command' * command'
  | Until of command' * command'
  | Function of name * command'
  | Redirection of command' * descr * kind * word
  | HereDocument of command' * descr * word'

and command' = command Location.located

and case_item = pattern' * command' option
and case_item' = case_item located

and kind =
  | Output
  | OutputDuplicate
  | OutputAppend
  | OutputClobber
  | Input
  | InputDuplicate
  | InputOutput
```

Fig. 9. Morsmall's AST.

written in such a way that it facilitates code review; (ii) it includes the formal shell grammar of the POSIX as-is; (iii) it is tested on a hand-written testsuite that contains many cornercases and on a large testsuite of real-world scripts; and (iv) it seems to behave like POSIX-compliant shells.

Code review

Comments represent almost 20% 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 parts 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. BNF is the most declarative and formal part of the specification, knowing that our generated parser recognizes the same language as this BNF argues in favor of trusting 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 contains just 177 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 addition 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).

More importantly, we ran both MORBIG and DASH on all the files detected as shell scripts by `libmagic`[5] in the Software Heritage [7] archive. This archive contains all the scripts in GitHub, and more, for a total of 7,436,215 files. Table 1 shows general numbers about what both parsers accept or reject in this archive. On most scripts (95%), MORBIG and DASH do agree. It is interesting to consider the cases where they disagree, because this is where one can find bugs in one parser or the other.

Out of the scripts accepted by Dash and rejected by Morbig the majority (350,259, *ie.* 94% and 4.7% of the total) contains BASH-specific constructs in words. DASH, in parse-only mode, separates words but does not look into them, hence it will only refuse them when executing the script. MORBIG, on the other hand, does parse words and rejects such scripts. This is neither a bug in DASH nor in MORBIG as the POSIX standard does not specify whether such invalid words must be rejected during parsing or during execution. The remaining 23,464 (0.3% of the corpus) that are accepted by DASH and rejected by MORBIG are due to remaining bugs in MORBIG or in DASH.

There are only 0.03% of scripts which are accepted by MORBIG and refused by DASH. These are either due to bugs in MORBIG, or in DASH, or to the fact that the standard is ambiguous.

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).

Command line interface The CLI of MORBIG is an executable program called `morbig`. It takes as input a list of filenames and can as an option produce, for each syntactically correct input file, a file containing a representation of its concrete syntax tree. The available output formats are a binary format that is very efficient in space and time consumption, a complete JSON text format including information about localization in the input file, and a *simplified* JSON format without localization information.

OCaml API

To use the OCaml API of MORBIG, a programmer writes an OCAML program linked to the `morbig` library. The part of the API concerning parsing, signalization and extraction of information from concrete syntax trees, is kept simple (see Fig. 7).

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 [11] is an object-oriented programming

[5] `libmagic` is a standard library to detect file formats with heuristics.

```c
#include <caml/mlvalues.h>

/* Initialize morbig. This function must be called before any other
 * calls to other functions of this API. */
void initialize_morbig (char** argv);

/* The type for concrete syntax trees. */
typedef value cst_t;

/* Parse ‘filename‘ and return the corresponding syntax tree if
 * ‘filename‘ contains a syntactically valid shell script. If
 * an error is detected, ‘NULL‘ is returned and ‘get_error ()‘
 * returns a human readable error message.  */
cst_t parse_file (char* filename);

/* If a call to ‘parse_file (filename)‘ produced an error,
 * ‘get_error_message‘ returns a human readable error message.  */
char* get_error_message ();

/* Let OCaml GC reclaim the concrete syntax tree.  */
void  dispose_cst (cst_t cst);

/* A concrete syntax tree is a tree with three kind of nodes:
 *
 * - LOCATION nodes annotate a tree with two positions that characterize
 *   the portion of the source file that produced this tree.
 *
 * - NODE nodes correspond to the application of a POSIX grammar rule.
 *   This application is specified by a rule name and a possibly empty
 *   list of producers (i.e. concrete syntax subtrees).
 *
 * - DATA nodes correspond to literals.
 */
typedef enum kind { LOCATION, NODE, DATA } kind_t;

/* Return the kind of the node of the cst root. */
kind_t get_kind (cst_t cst);

/* Return the located concrete syntax tree.
 * Precondition: ‘get_kind(cst) == LOCATION‘. */
cst_t get_located_value (value cst);

/* Return the rule name.
 * Precondition: ‘get_kind(cst) == NODE‘. */
char* get_rule_name (value cst);

/* Return the number of concrete subtrees.
 * Precondition: ‘get_kind(cst) == NODE‘. */
int get_number_of_children (value cst);

/* Return the number of concrete subtrees.
 * Precondition: ‘get_kind(cst) == NODE
 *             && k < get_number_of_children(cst)‘. */
cst_t get_children (value cst, int k);

/* Return the literal string representation.
 * Precondition: ‘get_kind(cst) == DATA‘ */
char* get_data (value cst);
```

Fig. 8. An excerpt of the C API of Morbig library.

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 [12,13] which automatically generates visitors classes out of type definitions.

C API

Like most general purpose languages, the OCaml language provides an interoperability interface with the C language[6] Morbig exploits this interoperability layer to offer a programmable interface for C programs. Therefore, the `morbig` library can be seen as a basic C library, and can also be used from any programming language that is able to interact with C.

In OCaml, concrete syntax trees are represented by values of algebraic datatypes, with each non-terminal of the shell grammar corresponding to a different type and each grammar rule being representing by a data constructor. As a consequence, static typing guarantees that concrete trees are made of valid applications of grammar rules.

Even though there are unfortunately no such rich types in C, these guarantees also hold about the C values of type `cst_t`[7] Indeed, the type

[6] As documented at http://caml.inria.fr/pub/docs/manual-ocaml/intfc.html.

[7] We assume that the programmer does not cast this type in order to break abstraction.

```
let process_script filename cst =
let detect_parameter =
  object (self)
    inherit [_] Morbig.CST.iter as super
    method! visit_word_component for_variables = function
      | WordVariable (VariableAtom(s,_)) ->
         if not (List.mem s for_variables)
         then register_identifier s filename;
      | x -> super#visit_word_component for_variables x

    method! visit_for_clause for_variables = function
      | ForClause_For_Name_DoGroup(name',do_group')
      | ForClause_For_Name_SequentialSep_DoGroup _
      | ForClause_For_Name_LineBreak_In_SequentialSep_DoGroup  _ ->
         self#visit_do_group' ((unName' name')::for_variables) do_group'
      | ForClause_For_Name_LineBreak_In_WordList_SequentialSep_DoGroup
         (name',_,word_list',_,do_group') ->
         self#visit_do_group'
           (if expandable word_list'.value
            then for_variables else (unName' name')::for_variables)
           do_group'
  end
in detect_parameter#visit_program [] cst
```

Fig. 10. Counting variables that are not set by a for-loop.

```
Analyzer.(register_analyzer (
struct
  let name = "quoting/find"
  let author = "Donald Duck <dd@unix.org>"
  let short_description = "'find' patterns are quoted."
  let documentation =
"
 In `find -name *.c`, the glob *.c is expanded
 before the execution of find while it should be passed
 as a pattern to the 'name' argument as in
 `find -name '*.c'`.
"

let analyzer =
  let pattern_commands = ["-name"; "-path"] in
  let message = "Patterns of find must be quoted." in
  for_all_command (fun _ c ->
    for_all_arguments c (fun pos arg ->
      alarm_at pos message (
        one_of find_pattern_commands (word_precedes arg)
        && check_argument arg is_not_quoted_word)
    )
  )
end)
```

Fig. 11. A `lintshell` plugin to detect invalid quoting of `find` patterns.

`cst_t` is abstract and the API only provides functions that extract information from concrete syntax trees.

It is the OCaml runtime that takes care of the memory management of these values through the OCaml garbage collector. A function call of the form `dispose_cst(t)` informs the OCaml runtime that the C program does not need the concrete syntax tree `t` anymore. As usual in C, the programmer is responsible for assuring that this is indeed the case.

4.2. An abstract syntax tree for POSIX shell

For many applications (e.g. an interpreter), a concrete syntax tree contains too much information. For instance, it is often not interesting to know whether two commands are separated by a semicolon or a newline and, in that case, how many newlines there are. The objective of an *abstract syntax tree* (AST) is to abstract away from such information.

In several cases, the POSIX grammar allows some part of a script to be omitted and implicitly replaced by a default value. For instance, in redirections, specifying the precise `IO_NUMBER` to be redirected to is not mandatory. If it is not specified, then a default value is used, like 1 in case of an output redirection. In that case, we only need to know which `IO_NUMBER` is used, independently of whether this value was explicitly specified or whether the default value was filled in.

It is also interesting to notice that some constructions in the CST are semantically equivalent. For instance, adding braces around a portion of a shell script does not change its semantics in any way. Thus, this construction will not be represented in the AST.

We developed a tool called MORSMALL that defines an AST for POSIX shell as well as a conversion from MORBIG's CST. The complete definition of its AST can be found in Fig. 9. Some parts like the type `word_component` are very similar to what can be found in MORBIG. Some other parts are much more abstract. The type `command` alone regroups 21 types from the CST. In total, there are only 16 types in the AST while there are 117 types in the CST.

4.3. 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 needed a statistical analysis of our corpus of POSIX shell scripts in order to know what elements of the shell language and what UNIX commands with which options are mostly used in our corpus.

Already the apparently simple problem of counting the occurrences of control structures in a corpus of shell scripts goes beyond what can be done by counting the matches of some regular expression, due to the fact that words are keywords, and hence may mark the start of a control structure, only when they appear in a context where a keyword is expected (see the discussion in Section 2.2.2). Our analyzer works on the concrete syntax trees produced by MORBIG. Individual analyzers are written using the visitors provided by the shell parsing toolkit (see Section 4.1). For instance, the essential parts of an analyzer which counts the number of parameters in a script is shown in Fig. 10. This again is more difficult than just counting the number of occurrences of the character $ in the corpus since we have to exclude occurrences of the symbol in comments and here-documents in case they are protected from variable expansion. To demonstrate the use of an environment of the visitors we have refined our analysis further to exclude parameters that are set in an enclosing *for* loop for which the *wordlist*, that is the list of values over which the loop iterates, is not subject to shell expansion and hence is statically known.

4.4. A user-extensible LINT for POSIX shell

The historical `lint` utility, which was created by Bell Labs in 1979, is a static analyzer for C programs which gives programmers feedback

about potential problems in their code. A similar tool exists today for shell scripts: SHELLCHECK is a HASKELL program that parses shell scripts and generates warnings on the fly when it encounters bad shell programming patterns. The number of patterns recognized by SHELLCHECK is impressive, but its architecture makes it difficult to extend with new patterns. Indeed, no syntax tree is actually produced by SHELLCHECK, so the detection of a new pattern must be introduced in the parser itself, which is of course quite complex.

We started the development of a new `lint`-like tool for shell scripts which offers a plugin system to allow any OCAML programmer to introduce a new shell script analyzer in the tool. This tool is based on the `morbig` and `morsmall` libraries, this way a user has access to the API described in Section 4.1. An analysis can therefore be written concisely using visitors just like the analyzers of shell scripts described in Section 4.3. But, given the specificity of `lint` analyzers, we were also able to design a *combinator* library that allows for high-level and declarative definitions of many of these analyzers.

To get a glimpse of `lintshell` plugin programming, consider the code of Fig. 11. This code is the whole definition of a plugin which detects if a script invokes the `find` command without quoting its arguments which are `find` patterns. From Line 3 to Line 6, the user provides some metadata about the plugin (its name, author, and some user documentation).

The most interesting part is the definition of the analyzer which starts on Line 17. An analyzer could be seen as a function from concrete syntax trees of scripts to alarms. We decided, however, to provide an Embedded Domain Specific Language (EDSL) to help with the implementation of such functions. A program in our EDSL defines *criteria* that must trigger alarms. The combinators to build analyzers using this DSL allow us to quantify over specific syntactic constructions of the shell language (like the `for_all_command` of Line 17), and they also allow to provide common predicates over these syntactic constructions (like the `is_not_quoted_word` of Line 21). We found that the EDSL enhances the readability of plugins code and therefore simplifies the review, the maintenance and the evolution of these scripts.

In addition, plugins written using this EDSL compose well: `lintshell` can factorize their applications into a single traversal of the syntax trees. On the contrary, each plugin written with a visitor or as bare OCaml function of type `program -> alarm list` costs one traversal of the syntax trees. Of course, an EDSL has a limited expressivity and it is sometimes necessary to fall back to these more expensive ways of implementing analysis for more complex analysis.

5. Current limitations and future work

An important issue is how to validate our parser. The testsuite presented in Section 3.5 is only a partial answer to this problem. Indeed, counting the number of scripts that are recognized as being syntactically correct or incorrect is only a first step to validate our parser since it does not tell us whether the syntax tree constructed by the parser is the correct one. For instance, the interpretation of backslashes in word literals cannot be validated unless we actually compare the trees produced by MORBIG with the trees produced by other POSIX shell implementations. We can nonetheless imagine several ways how the parser can be validated in the future.

One approach is to write a *pretty-printer* that 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 that 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.

We also recall that static parsing is incompatible with aliases and the `eval` mechanism, as explained in Section 2.3. For this reason, MORBIG will always have only a limited support for these constructions.

6. Availability and benchmarks

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

On a i7-4600U CPU @ 2.10GHz with 4 cores, an SSD hard drive and 8GB of RAM, it takes 7.38s[8] to parse the 31,330 POSIX scripts among the 31,582 maintainer scripts in the Debian GNU/Linux distribution and to serialize the corresponding concrete syntax trees on the disk. Our parser fails on only one script which uses indeed a BASH-specific extension of the syntax.

The average time to parse a script from the corpus of Debian maintainer scripts is therefore 0.2ms (with a standard deviation which is less than 1% of this duration). The maximum parsing time is 70ms, reached for the `prerm` script of package `w3c-sgml-lib_1.3-1_all` which is 1121 lines long.

We compared MORBIG to DASH on the whole archive from Software Heritage, containing 7,436,215 scripts in total. We used a machine with an Intel Xeon Processor E5-4640 v2 @ 2.20GHz with 40 cores and 756GB of RAM, where all the scripts were loaded in a tmpfs in RAM. It takes 400s to DASH and 3400s to MORBIG to parse all these scripts. This means respectively 19,000 and 2200 scripts per second. Although DASH is faster, the difference is less than an order of magnitude.

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 [14]. 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 [15]. 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 [16–18] has presented a formal semantics of POSIX shell. This work is however not focused on the syntax and relies on external parsers[18]. suggests that MORBIG could be one of these in the future. The work behind Abash [19] 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. It is however

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

limited to this particular point of Bash scripts. Several other tools can spot certain kinds of errors in shell scripts. The CHECKBASHISMS [20] script detects usage of Bash-specific syntax in shell scripts, it is based on matching Perl regular expressions against a normalized shell script text. It does not include a parser for any variant of shell. This tool is currently used in Debian as part of the `lintian` package analyzing suite. The tool SHELLCHECK [21] 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 MENHIR's new mechanism to specify error cases to produce error messages of similar quality.

7.2. About parsing technologies

General parsing frameworks

MENHIR [10] is based on a conservative extension of LR(1)[4], inspired by Pager's algorithm [22]: 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 [23,24], GLR [25], GLL [26] or general parser combinators [27]. 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 [28] or data-dependent grammars [29].

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 may be 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 nondeterminism 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.

Scannerless parsing

Many legacy languages (e.g. PL/1, COBOL, FORTRAN, R, etc.) enjoy a syntax that 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 [30] has been able to offer a declarative language to specify modular scannerless grammar [31] 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 2.1.3. 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 [32] 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 the traditional approach.

Acknowledgments

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 thank François Pottier for the incremental engine of Menhir without which this work would not have been impossible. 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] Moved ~ /.local/share/steam. ran steam. it deleted everything on system owned by user, GitHub bug report (Jan. 2014). https://github.com/valvesoftware/steam-for-linux/issues/3671.

[2] A.M. Ucko, Cmigrep: broken emacsen-install script, Debian Bug Report 431131 (2007). https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=431131.

[3] The Debian Project, Bugs tagged colis. https://bugs.debian.org/cgi-bin/pkgreport.

cgi?tag = colis-shparser;users = treinen@debian.org.
[4] D.E. Knuth, On the translation of languages from left to right, Inf. Control 8 (6) (1965) 607–639.
[5] Y. Régis-Gianas, N. Jeannerod, R. Treinen, Morbig: a static parser for POSIX shell, in: D. Pearce, T. Mayerhofer, F. Steimann (Eds.), Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, ACM, Boston, MA, USA, 2018, pp. 29–41, , https://doi.org/10.1145/3276604.3276615. November 05–06, 2018
[6] 2018,. IEEE, The Open Group, The open group base specifications issue 7, https://pubs.opengroup.org/onlinepubs/9699919799.2016edition/.
[7] J. Abramatic, R.D. Cosmo, S. Zacchiroli, Building the universal archive of source code, Commun. ACM 61 (10) (2018) 29–31, https://doi.org/10.1145/3183558.
[8] 2016, IEEE, The Open Group, The open group base specifications issue 7, http://www.unix.org/version3/online.html.
[9] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, (2nd ed.), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[10] F. Pottier, Y. Régis-Gianas, The menhir parser generator, http://gallium.inria.fr/fpottier/menhir.
[11] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India (1995).
[12] F. Pottier, Visitors, 2017a, http://gallium.inria.fr/~fpottier/visitors/manual.pdf.
[13] F. Pottier, Visitors unchained, Proceedings of the ACM on Programming Languages 1 (ICFP), (2017), pp. 28:1–28:28, https://doi.org/10.1145/3110272.
[14] R.D. Cosmo, D.D. Ruscio, P. Pelliccione, A. Pierantonio, S. Zacchiroli, Supporting software evolution in component-based FOSS systems, Sci. Comput. Program. 76 (12) (2011) 1144–1160, https://doi.org/10.1016/j.scico.2010.11.001.
[15] D.D. Ruscio, P. Pelliccione, A. Pierantonio, S. Zacchiroli, Towards maintainer script modernization in FOSS distributions, Proceedings of the International Workshop on Open Component Ecosystem, IWOCE, ACM, 2009, pp. 11–20, https://doi.org/10.1145/1595800.1595803.
[16] M. Greenberg, Understanding the POSIX shell as a programming language, (2017) Paris, France. Off the Beaten Track 2017
[17] M. Greenberg, Word expansion supports POSIX shell interactivity, in: S. Marr, J.B. Sartor (Eds.), Proceedings of the Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, ACM, Nice, France, 2018, pp. 153–160, , https://doi.org/10.1145/3191697.3214336. April 09–12, 2018
[18] M. Greenberg, A.J. Blatt, Executable formal semantics for the POSIX shell, PACMPL 4(POPL): 43:1-43:30 (2020), doi:10.1145/3371111.
[19] K. Mazurak, S. Zdancewic, 2007, San Diego, CA, USA. ABASH: finding bugs in bash scripts, Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, 105–114.
[20] R. Braakman, J. Rodin, J. Gilbey, M. Hobley, Checkbashisms, 2015, https://sourceforge.net/projects/checkbaskisms/.
[21] V. Holen, Shellcheck, 2015, https://github.com/koalaman/shellcheck.
[22] D. Pager, A practical general method for constructing LR (k) parsers, Acta Inf. 7 (3) (1977) 249–268.
[23] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (2) (1970) 94–102.
[24] J. Aycock, R.N. Horspool, Practical earley parsing, Comput. J. 45 (6) (2002) 620–630.
[25] M. Tomita, S.K. Ng, The generalized LR parsing algorithm, Generalized LR Parsing, Springer, 1991, pp. 1–16.
[26] E. Scott, A. Johnstone, GLL Parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189.
[27] A. Izmaylova, A. Afroozeh, T.v.d. Storm, Practical, general parser combinators, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ACM, 2016, pp. 1–12.
[28] P. Stansifer, M. Wand, Parsing reflective grammars, Proceedings of the Eleventh Workshop on Language Descriptions,Tools and Applications, ACM, 2011, p. 10.
[29] A. Afroozeh, A. Izmaylova, Iguana: a practical data-dependent parsing framework, Proceedings of the 25th International Conference on Compiler Construction, ACM, 2016, pp. 267–268.
[30] 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, Proceedings of the International Conference on Compiler Construction, Springer, 2001, pp. 365–370.
[31] E. Visser, et al., Scannerless generalized-LR parsing, 1997, Universiteit van Amsterdam. Programming Research Group.
[32] J. Aycock, R.N. Horspool, Schrödinger's token, Softw. Pract. Exp. 31 (2001) 803–814.