Parsing POSIX [S]hell

Yann Régis-Gianas

in collaboration with Nicolas Jeannerod and Ralf Treinen

PPS Days, November 9th, 2018

CoLiS : Verification of Debian maintainer scripts

- Debian maintainer scripts are shell scripts executed as root.
- They are critical and complex pieces of code that should be verified.
- The CoLiS project develops automated tools to analyze these scripts.
- These analysis tools are part of our trusted base.

This talk

How to write a POSIX Shell parser you can trust?

This talk

How to write a POSIX Shell parser you can trust?

All hope abandon ye who enter here.
– Dante's Divine Comedy

This talk

How to write a POSIX Shell parser you can trust?

All hope abandon ye who enter here.
– Dante's Divine Comedy

Our contributions (SLE'18):

- Give an overview of the difficulties of POSIX Shell parsing.
- Follow a modular architecture to tackle these difficulties.
- Use a purely functional LR(1) parser for advanced parsing techniques.
- `morbig`, a static parser for POSIX Shell.

Compiler Construction 101

Figure: Parsing "as in the textbook".

From informal specifications to high-level formal ones

- Rewrite the lexical conventions into a Lex specification.
- Rewrite the BNF grammar into a Yacc specification.
- Being declarative, these specifications are trustworthy.
- Code generators, like compilers, are trustworthy too.

[S]hell specification deciphering

The POSIX Shell specification

- POSIX Shell is specified by the Open Group and IEEE.
- There is a Yacc grammar in the specification! Hurray!

[S]hell specification deciphering

The POSIX Shell specification

- POSIX Shell is specified by the Open Group and IEEE.
- There is a Yacc grammar in the specification! Hurray!

  ...but it is “annotated” by side-conditions out of reach of LR(1) parsers.

[S]hell specification deciphering

The POSIX Shell specification

- POSIX Shell is specified by the Open Group and IEEE.
- There is a Yacc grammar in the specification! Hurray!

  ...but it is “annotated” by side-conditions out of reach of LR(1) parsers.

[S]hell specification deciphering

The POSIX Shell specification

- POSIX Shell is specified by the Open Group and IEEE.
- There is a Yacc grammar in the specification! Hurray!

  ...but it is "annotated" by side-conditions out of reach of LR(1) parsers.

```
pattern          :             WORD         /* Apply rule 4 */
                 | pattern '|' WORD         /* Do not apply rule 4 */
                 ;
function_body    : compound_command                /* Apply rule 9 */
                 | compound_command redirect_list  /* Apply rule 9 */
                 ;
fname            : NAME                            /* Apply rule 8 */
                 ;
brace_group      : Lbrace compound_list Rbrace
                 ;
do_group         : Do compound_list Done           /* Apply rule 6 */
                 ;
```

> 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.

[S]hell specification deciphering

The POSIX Shell specification

- POSIX Shell is specified by the Open Group and IEEE.
- There is a Yacc grammar in the specification! Hurray!

  ...but it is "annotated" by side-conditions out of reach of LR(1) parsers.

Horror!

After careful analysis, we understood that the [S]hell language "enjoys":

- a **parsing-dependent**, **"shell nesting"-dependent** lexical analysis ;
- an **ambiguous** and even **undecidable** problem (if `alias` is used) ;
- a **lot of irregularities** and **unconventional design choices**.

The forthcoming examples illustrate (some of) these problems.

Token recognition

Unconventional lexical conventions

- In usual specifications, regular expressions with a longest-match strategy describe how to recognize the next lexeme in the input.
- The Shell specification uses a state machine which explains instead how tokens must be **delimited** in the input.
- The Shell specification tells us how the delimited chunks of input must be classified into two categories of "pretokens": **words** and **operators**.
- The meaning of newline characters **depends on the parsing context**.
- The meaning of escaping sequences **depends on the nesting of subshells and double-quotes**.

Example of token recognition

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

- Line 1 contains only one word.
- Line 2 contains four words and one operator.

Example of token recognition

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

- Line 1 contains only one word.
- Line 2 contains four words and one operator.

This token recognition logic impacts the style of Lex specifications.

What does this newline mean?

Newline has four different meanings

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

- On Lines 1 and 4, `\n` is a token.
- On Line 2, `\n` is ignored as part of a comment.
- On Line 3, `\n` is a line-continuation.
- On Line 5, `\n` is a end-of-phrase marker.

What does this newline mean?

Newline has four different meanings

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

- On Lines 1 and 4, `\n` is a token.
- On Line 2, `\n` is ignored as part of a comment.
- On Line 3, `\n` is a line-continuation.
- On Line 5, `\n` is a end-of-phrase marker.

Some newline characters – but not all – occur in grammar rules.

Do you want to escape?

Quiz

In **`dash`**, which is the command that outputs `\\`?

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

Do you want to escape?

Quiz

In **dash**, which is the command that outputs `\\`?

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

Six backslashes are needed to achieve proper escaping! and what about:

```
1  echo `echo "\\\\\\"`
```

?

Do you want to escape?

Quiz

In **dash**, which is the command that outputs `\\`?

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

Six backslashes are needed to achieve proper escaping! and what about:

```
1 echo `echo "\\\\\\"`
```

?

```
dash: 1: Syntax error: Unterminated quoted string
```

Do you want to escape?

Quiz

In **dash**, which is the command that outputs `\\`?

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

Six backslashes are needed to achieve proper escaping! and what about:

```
1  echo `echo "\\\\\\"`
```

?

```
dash: 1: Syntax error: Unterminated quoted string
```

Escaping depends on the nesting of subshells and double quotes.

Which exact token is that?

Promotion of words

- The grammar specification is not defined in terms of words and operators, which are actually pretokens, but with respect to a more refined set of tokens.
- Hence, words must sometimes be promoted into:
  - Assignment words, e.g. `X=foo`.
  - Reserved words, e.g. `if`, `for`, etc.
- This promotion **depends on the parsing context**.

Promotion of a word to a reserved word

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

- The first `for` is a reserved word, the second one is a word.
- The first and second `do` are words, the third one is a reserved word.
- The first `in` is a reserved word, the second one is a word.

Promotion of a word to a reserved word

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

- The first `for` is a reserved word, the second one is a word.
- The first and second `do` are words, the third one is a reserved word.
- The first `in` is a reserved word, the second one is a word.

A word is promoted to a reserved word if the parser expects it here.

Forbidden positions for specific reserved words

```
1 else echo foo
```

- `else` is not allowed here, even as a regular word!
- Thus, `/bin/else` is not a good naming choice for your next tool...

Forbidden positions for specific reserved words

```
1 else echo foo
```

- `else` is not allowed here, even as a regular word!
- Thus, `/bin/else` is not a good naming choice for your next tool...

These irregularities constrain the parser with adhoc side-conditions.

`alias` aka “decidability breaker”

Icing on the cake

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

- This script has a syntax error, or not! `./foo` decides!

`alias` aka “decidability breaker”

Icing on the cake

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

- This script has a syntax error, or not! `./foo` decides!

This makes static parsing of script files undecidable!
(Yes, parsing depends on evaluation!)

Does this talk even exist?

How to write a POSIX Shell parser ~~you can trust~~?

Just a glimpse of Dash parser

```
1       case TFOR:
2               if (readtoken() != TWORD || quoteflag || ! goodname(wordtext))
3                       synerror("Bad for loop variable");
4               n1 = (union node *)stalloc(sizeof (struct nfor));
5               n1->type = NFOR;
6               n1->nfor.linno = savelinno;
7               n1->nfor.var = wordtext;
8               checkkwd = CHKNL | CHKKWD | CHKALIAS;
9               if (readtoken() == TIN) {
10                      app = &ap;
11                      while (readtoken() == TWORD) {
12                              n2 = (union node *)stalloc(sizeof (struct narg));
13                              n2->type = NARG;
14                              n2->narg.text = wordtext;
15                              n2->narg.backquote = backquotelist;
16                              *app = n2;
17                              app = &n2->narg.next;
18                      }
19                      *app = NULL;
20                      n1->nfor.args = ap;
21                      if (lasttoken != TNL && lasttoken != TSEMI)
22                              synexpect(-1);
23              } else {
24                     [...]
25              }
26              checkkwd = CHKNL | CHKKWD | CHKALIAS;
27              if (readtoken() != TDO)
28                      synexpect(TDO);
29              n1->nfor.body = list(0);
30              t = TDONE;
31              break;
```

My feelings

Not the kind of code I would like to maintain (and to trust)

Open your (advanced) textbooks again!

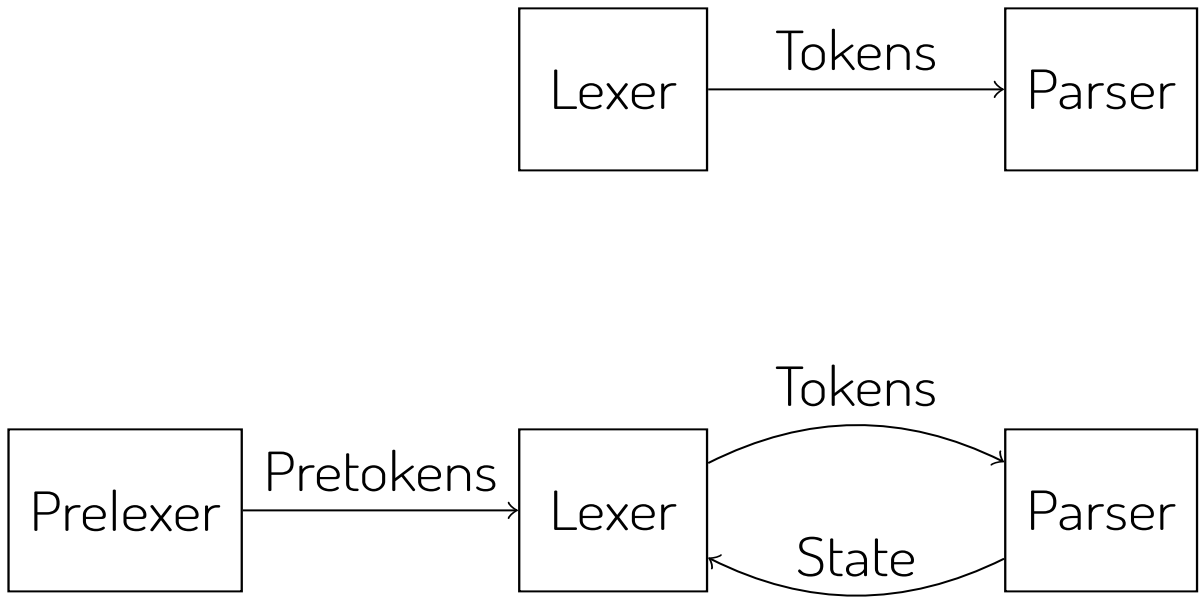

Figure: Another modular architecture for parsing.

Morbig, a **modular** parser for POSIX Shell scripts written in OCaml

Key parsing techniques

- **Parameterized lexers** to deal with contextual dependencies.
- **Speculative parsing** to promote words to reserved words.
- **Longest-prefix and reentrant parsing** to handle nested subshells.
- **Parser state introspection** to handle irregularities modularly.

Morbig, a **modular** parser for POSIX Shell scripts written in OCaml

Key parsing techniques

- **Parameterized lexers** to deal with contextual dependencies.
- **Speculative parsing** to promote words to reserved words.
- **Longest-prefix and reentrant parsing** to handle nested subshells.
- **Parser state introspection** to handle irregularities modularly.

Key implementation aspects

- Our prelexer is generated by a "standard" ocamllex specification.
- Yacc grammar is a **cut-and-paste from the standard**.
- Adhoc parsing rules are externally and modularly defined.
- We crucially rely on the **purely functional** and **incremental** parsers produced by Menhir, an LR(1) parser generator for OCaml.

Menhir functional and incremental parsing interface

- Usually, parser generators produce a function of type:

```
parse : lexer -> ast
```

- Menhir has an alternative signature, roughly speaking of type:

```
parse : unit -> 'a checkpoint
```

where

```
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
```

Menhir functional and incremental parsing interface

- The **incremental** interaction with the parser is done through:

```
offer: 'a checkpoint -> token -> 'a checkpoint
```

  to provide the parser with only one token at a time ; and

```
resume: 'a checkpoint -> 'a checkpoint
```

  to let the parser realizes a single step of analysis.
- The entire parsing state is encapsulated in the `checkpoint`.
- Backtracking is transparent: it is a mere restart from a `checkpoint`.
- Menhir provides **parsing state introspection** functions.

Constrained parsing

```
| AboutToReduce (env, p) ->
let rec reject_cmd_word_if_reserved_word () =
  if is_cmd () && is_keyword () then parse_error ()

and is_cmd () =
  nonterminal_of_production p = AnyN N_cmd_word

and is_keyword () =
  on_top_symbol env keyword_found

and keyword_found : type a. a symbol * a -> _
= function
  | N N_word, Word (w, _) -> is_reserved_word w
  | _ -> false
in
...
```

Conclusion

Morbig

- A standalone program **`morbig`** and a library.
- Turn a shell script into a syntax tree, represented in JSON.
- Successful parsing of 31521 Debian scripts ($\simeq$9s on my laptop)

Conclusion

Morbig

- A standalone program `morbig` and a library.
- Turn a shell script into a syntax tree, represented in JSON.
- Successful parsing of 31521 Debian scripts ($\simeq$9s on my laptop)

Do we trust Morbig (yet)?

- **NO**!
- Our goal is to reach a state where:
  - there is a as-clearest-as-possible mapping between spec. and code ;
  - our understanding of POSIX Shell is made explicit by a readable code.

Thank you for your attention

(and sorry for the nightmares!)

`morbig` is free software:

`https://github.com/colis-anr/morbig`

"If you are going through [s]hell, keep going." – Winston S. Churchill

Other tricks

Here-documents

- Switching between two lexers is easy in incremental mode.
- We "back-patch" semantic values of `WORD`s once here-documents are entirely parsed. (Yes, using references.)

Newlines

- Our lexer may produce one or more tokens at each (pre)lexing step.
- A buffer synchronizes prelexer and parser.
- Some newlines are manually ignored depending on parsing context.

Alias

- No magic bullet about `alias` since we refuse to embed an interpreter.
- We only accept toplevel aliases.

What I did not talk about, the secret monsters

Escaping

- Shell escaping sequences are "interesting".
- A well-chosen nesting of `$(...)` and `` `...` `` requires an exponential number of backslashes.

Parsing a script

- `EOF` in the grammar does not mean end-of-file.
- It means end-of-phrase.
- The specification forgets to say something about empty scripts.

More monsters

> The syntax of the shell command language has an ambiguity for expansions beginning with "$((", which can introduce an arithmetic expansion or a command substitution that starts with a subshell. Arithmetic expansion has precedence; that is, the shell shall first determine whether it can parse the expansion as an arithmetic expansion and shall only parse the expansion as a command substitution if it determines that it cannot parse the expansion as an arithmetic expansion.

Arithmetic expressions

This is not yet implemented.

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

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

Comments

Recognition of comments

- `#` is **not** a delimiter.
- Therefore, there is no comment in the following phrase:

```
ls foo#bar
```

- but there is one here:

```
ls foo #bar
```

Here documents

Here-documents recognition is non-local

```
1 cat > notifications << EOF
2 Hi $USER,
3 Enjoy your day!
4 EOF
5 cat > toJohn << EOF1 ; cat > toJane << EOF2
6 Hi John!
7 EOF1
8 Hi Jane!
9 EOF2
```

- The word related to **`EOF1`** is recognized several tokens after the location of **`EOF1`**.

Promotion of a word to an assignment word

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

Speculative parsing

```
let recognize_reserved_word_if_relevant =
fun checkpoint pstart pstop w ->
FirstSuccessMonad.(
  let as_keyword =
    keyword_of_string w >>= fun kwd ->
    let kwd' = (kwd, pstart, pstop) in
    return_if (
      accepted_token checkpoint kwd' <> Wrong
    ) kwd
  in
  let as_name =
    return_if (Name.is_name w) (NAME (CST.Name w))
  in
  as_keyword +> as_name
)
```