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.
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

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.

Figure: Parsing “as in the textbook”.
The POSIX Shell specification

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

```plaintext
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
```  

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.
Shell 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

▶ Line 1 contains only one word.
▶ Line 2 contains four words and one operator.
Example of token recognition

1. \texttt{BAR='foo''ba'\textbackslash r}
2. \texttt{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

```
$ for i in 0 1
> # Some interesting numbers
> do echo $i \n
> + $i
> 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

```bash
$ for i in 0 1
> # Some interesting numbers
> do echo $i \n>
> + $i
> 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 `\`?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>echo &quot;\&quot;</td>
</tr>
<tr>
<td>2</td>
<td>echo &quot;\&quot;</td>
</tr>
<tr>
<td>3</td>
<td>echo &quot;\\&quot;</td>
</tr>
</tbody>
</table>
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.**
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

```bash
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

```
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

```bash
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

```bash
if ./foo; then
    alias mystery="for"
else
    alias mystery=""
fi
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

```bash
if ./foo; then
    alias mystery="for"
else
    alias mystery=""
fi
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

case TFOR:
    if (readtoken() != TWORD || quoteflag || ! goodname(wordtext))
        synerror("Bad for loop variable");
    n1 = (union node *) stalloc(sizeof (struct nfor));
    n1->type = NFOR;
    n1->nfor.linno = savelinno;
    n1->nfor.var = wordtext;
    checkkw = CHKNL | CHKKWD | CHKALIAS;
    if (readtoken() == TIN) {
        app = &ap;
        while (readtoken() == TWORD) {
            n2 = (union node *) stalloc(sizeof (struct narg));
            n2->type = NARG;
            n2->narg.text = wordtext;
            n2->narg.backquote = backquotelist;
            *app = n2;
            app = &n2->narg.next;
        }
        *app = NULL;
        n1->nfor.args = ap;
        if (lasttoken != TNL && lasttoken != TSEMI)
            synexpect(-1);
    } else {
        ...
    }
    checkkw = CHKNL | CHKKWD | CHKALIAS;
    if (readtoken() != TDO)
        synexpect(TDO);
    synexpect(TDO);
    n1->nfor.body = list(0);
    t = TDONE;
    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.

**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.
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:

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

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

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

where

```ocaml
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:

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

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

  1. `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

```ocaml
| 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_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 (≈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.
Conclusion

Morbig

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

Do we trust Morbig (yet)?

- \textbf{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 `WORDs` 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 $\ldots$ and `\ldots` 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 =
  \[
  \text{match } \text{checkpoint } \text{with} \\
  | \text{InputNeeded } _ \rightarrow \\
  \quad \text{close (offer checkpoint token)} \\
  | _ \rightarrow \\
  \quad \text{false}
  \]

let rec close checkpoint = \text{match } \text{checkpoint } \text{with}
  \[
  | \text{AboutToReduce } _ \rightarrow \text{close (resume checkpoint)} \\
  | \text{Rejected } | \text{HandlingError } _ \rightarrow \text{false} \\
  | \text{Accepted } _ | \text{InputNeeded } _ | \text{Shifting } _ \rightarrow \text{true}
  \]
Recognition of comments

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

```
1  ls foo#bar
```

- but there is one here:

```
1  ls foo  #bar
```
Here documents

Here-documents recognition is non-local

```bash
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

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