Parsing [S]hell

Yann Régis-Gianas and Ralf Treinen

in collaboration with Nicolas Jeannerod

Minidebconf Hamburg, May 18, 2018

CoLiS : Verification of Debian maintainer scripts

- `preinst`, `postinst`, `prerm`, `postrm`
- Executed as root during package installation/removal/upgrade
- Must work correctly in different contexts (installed packages)
- May modify files in directories created by other packages: emacs, texlive, . . .
- We need automated tools that can analyze these scripts.

Why Testing May Not Be Enough

```
Date: Sun, 18 Mar 2018 14:43:45 -0400
Subject: Bug#893424: Cannot uninstall package

...
Removing sendmail-base (8.15.2-10) ...
rm: cannot remove '/etc/mail/m4': Is a directory
```

- version 8.15.2-10 of sendmail accepted in sid on 2018-01-19
- popcon number of sendmail-base: 2953
- why wasn't this bug observed before?

The origin of bug#893424

- The `postrm` contains

  ```
  find /etc/mail -maxdepth 1 -size 0 | xargs -r rm
  ```

- The maintainer has probably assumed that directories cannot have size 0.
- However, the unfortunate user had `/etc` on a btrfs filesystem, where directories may have size 0.
- Obvious fix: add `-type f` to the invocation of `find`.

So let's analyze scripts!

- Sid, 2016-11-29, amd64, all three areas: 31.832 maintainer scripts:
  - 296 bash scripts,
  - 14 perl scripts,
  - 1 ELF executable,
  - 31.521 POSIX shell scripts.
- So. let us focus on POSIX shell scripts.
- The first step of our toolchain: a parser for POSIX shell scripts.

This talk

How to write a POSIX Shell parser you can trust?

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

Compiler Construction 101

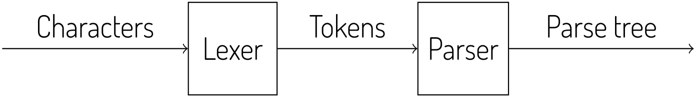


Figure: Parsing "as in the textbook".

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!
- ...but it is "annotated" by side-conditions out of reach of LR(1) parsers.
- Besides, the specification is low-level, unconventional and informal...

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**.

The forthcoming examples illustrate (very few 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.

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.

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

Do you want to escape?

Quiz

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

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

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

```
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

```
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

```
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

```
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~~?

Forget your textbooks! This is real world!

Existing implementations

- Existing implementations are not following the textbook architecture.
- The parser of Dash is made of ~ 1600 lines of hand-crafted C.
- The parser of Bash is based on a Yacc grammar (entirely different from the standard) extended with an extra ~ 5000 lines of C.

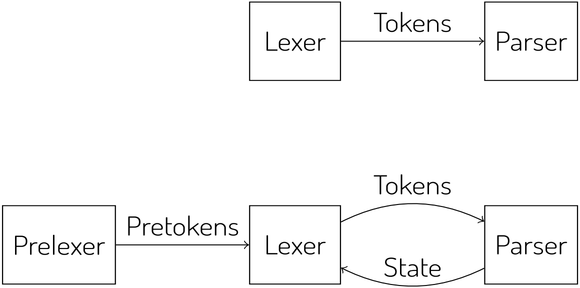
Just a glimpse of Dash parser

```
    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;
                checkkwd = 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 {
                       [...]
                }
                checkkwd = CHKNL | CHKKWD | CHKALIAS;
                if (readtoken() != 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!



Lexer

Tokens

Parser

Prelexer

Pretokens

Lexer

Tokens

State

Parser

Figure: Another modular architecture for parsing.

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

Key implementation aspects

- Yacc grammar is a cut-and-paste from the standard.
  (minus 5 shift/reduce conflicts)
- Our prelexer is generated by a "standard" ocamllex specification.
- We crucially rely on the **purely functional** and **incremental** parsers produced by Menhir, an LR(1) parser generator for OCaml.

Key parsing techniques (thanks to Menhir)

- **Speculative parsing** to promote words to reserved words.
- **Longest-prefix parsing** to handle nesting subshell parsing.
- **Parameterized lexers** to deal with contextual-depencencies.
- **Parser state introspection** to handle irregularities modularly.

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:

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

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

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

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

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)?

- Of course **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!

Wait for the release in June, then be brave enough to try it:

`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

```
cat > notifications << EOF
Hi $USER,
Enjoy your day!
EOF
cat > toJohn << EOF1 ; cat > toJane << EOF2
Hi John!
EOF1
Hi Jane!
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
```

Speculative parsing

```
let recognize_reserved_word_if_relevant =
fun checkpoint pstart pstop w ->
  try
    let kwd = keyword_of_string w in
    let kwd' = (kwd, pstart, pstop) in
    if accepted_token checkpoint kwd' then
      return kwd
    else
      raise Not_found
  with Not_found ->
    if is_name w then
      return (NAME (CST.Name w))
    else
      return (WORD (CST.Word w))
```

Constrained parsing

```
| AboutToReduce (env, production) -> begin try
  if lhs production = X (N N_cmd_word)
  || lhs production = X (N N_cmd_name) then
    match top env with
    | Some (Element (state, v, _, _)) ->
      let analyse_top = function
      | T T_NAME, Name w when is_reserved_word w
      | T T_WORD, Word w when is_reserved_word w ->
        raise ParseError
      | _ -> assert false
      in
      analyse_top (incoming_symbol state, v)
    | _ -> assert false
  else
    raise Not_found
  with Not_found -> parse (resume checkpoint)
end
```