Parsing [s]hell

Yann Régis-Gianas

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

Séminaire Gallium, September 18, 2017

Package scripts are critical pieces of software! **Right!**

Package scripts are critical pieces of software! **Right!**

Let us verify they cannot break our systems! **Yes!**

Package scripts are critical pieces of software! **Right!**

Let us verify they cannot break our systems! **Yes!**

By the way, they are written in POSIX shell!

CoLiS : Verification of Debian packages installation scripts

Package scripts are critical pieces of software! **Right!**
Let us verify they cannot break our systems! **Yes!**
By the way, they are written in POSIX shell! **. . . Glups**

This talk

How to write a shell parser you can trust?

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.

The [s]hell specification

The POSIX Shell Command Language

- It is specified by the Open Group and IEEE.
- The volume "Shell & Utilities" is the one we focus on.
- It is accessible online at:

  `http://pubs.opengroup.org/onlinepubs/9699919799/`

After deciphering

The POSIX Shell language defies conventional parsing wisdom

- The specification is low-level, unconventional and informal. . .
- It is also contradictory and ambiguous.
- After some analysis, we understood that the Shell language "enjoys":
    - a parsing-dependent lexical analysis ;
    - an undecidable parsing (when `alias` is used) ;
    - a lot of irregularities.
- The forthcoming examples illustrate some of these problems.

Token recognition

Unconventional lexical conventions

- In usual specifications, regular expressions with a longest-match strategy descrube 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: **words and operators**.

Example of token recognition

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

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.

No big deal! I am not afraid of recognizing nested languages with `ocamllex` and regular expressions can also be used to specify delimiters.

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
```

What does this newline mean?

Newline has four different meanings

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

What does this newline mean?

Newline has four different meanings

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

Some newline characters - *but not all* - occur in grammar rules.

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.

Which token is that?

Promotion of words

- The grammar specification is not defined in terms of words and operators 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 an assignment word

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

Promotion of a word to a reserved word

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

Forbidden positions for specific reserved words

```
else echo foo
```

alias aka "decidability breaker"

Ice on the cake

```
if ./foo; then
  alias x="ls"
else
  alias x=""
fi
x for i in a b; do echo $i; done
```

Are you afraid of LR(1) conflicts?

Menhir has spoken

- The Yacc grammar of the standard has **five** shift/reduce conflicts.
- All of them are related to the token `newline`.
- Does this newline is a separator (shift) or a terminator (reduce)?

Forget your textbooks! This is real world!

Existing implementations

- Existing implementations are not following the textbook architecture.
- The parser of DASH is made of 1569 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

```
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.

Open your (advanced) textbooks again!

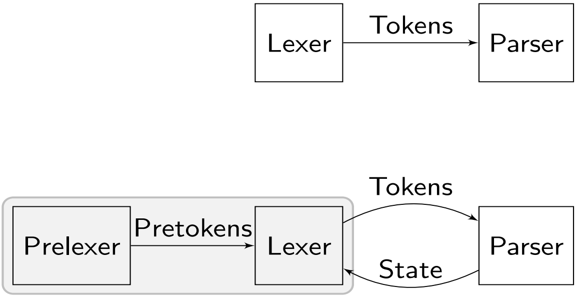

Figure: Another modular architecture for parsing.

Morbig, a parser for shell scripts

Key implementation aspects

- Our Yacc grammar is a cut-and-paste from the standard.
- Our prelexer is generated by a "standard" OCAMLLEX specification.
- Our engine implements the two arrows of the previous diagram.
- We crucially rely on the incremental and purely functional parsers produced by Menhir.

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 interaction with the generated parser is done through:

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

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))
```

```
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
```

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 : type a. a symbol * a -> _ = function
       | T T_NAME, Name w when is_reserved_word w
       | T T_WORD, Word w when is_reserved_word w ->
         raise ParseError
       | _ -> assert false
       in
       analyse_top (incoming_symbol state, v)
     | _ -> assert false
  else
    raise Not_found
  with Not_found -> parse (resume checkpoint)
end
```

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.

Conclusion

Morbig

- A standalone program `morbig` and a library.
- Successful parsing of 31521 Debian scripts ($\simeq$40s on my i7)
- A user-extensible lint for POSIX Shell

Do we trust Morbig (yet)?

- As is, we will probably never trust it.
- Our goal is to reach a state where:
  - there is a as-clearest-as-possible mapping between spec. and code ;
  - our view of POSIX is made explicit by the code and its testsuite.

Thanks for your attention and sorry for the nightmares!

A release of morbig will happen in few weeks.

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.