FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Command Line Description Generator
Student:	Nikita Evstigneev
Supervisor:	Ing. Jiří Kašpar
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

Study the Command Line Description (CLD) language and its extension for Linux and Windows OS.
Analyze the structure and format of Linux man pages.
Design and implement a tool generating command descriptions in the CLD language from Linux man pages.
Test the tool on man pages of the base SUSE Linux distribution.
Evaluate the resulting command descriptions in the dclsh shell prototype provided by the supervisor.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 17, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Command Line Description Generator

Nikita Evstigneev

Department of Information Security
Supervisor: Ing. Jiří Kašpar

January 6, 2021

Acknowledgements

I would like to thank my supervisor Ing. Jiří Kašpar for his help. Also, I would like to thank my family, friends, and colleagues for their support during this long academic journey.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 6, 2021 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Evstigneev, Nikita. *Command Line Description Generator.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato bakalářská práce analyzuje problém generovaní popisu příkazů v jazyce CLD z manuálních stránek, které pak mohou být použité v dclsh shellu. Cílem této práce je automatizovat proces psaní popisu příkazů tak, aby dosáhnout pohodlnější komunikaci mezi uživatelem a dclsh shellem.

Klíčová slova CLD, dclsh, man2cld, manuální stranka, příkazový řádek, shell

Abstract

This bachelor thesis analyzes the problem of generating command description in CLD language from its manual page, which then can be used in dclsh shell. The goal of this work is to automate the process of writing command descriptions in order for dclsh shell to achieve more convenient communication between user and shell.

Keywords CLD, dclsh, man2cld, man page, command line, shell

Contents

List of Figures

List of Tables

Introduction

When working with a command line, it's quite common to look up the syntax or available options of the command we are interested in. Unix shell provides a convenient way to do it with its manual pages. Though, there are use cases when navigating through a 15-pages manual is not efficient. What if shell could help us complete the option name or warn us about invalid argument type. What if shell knew that we need to pass a name of a user as an argument and listed it for us? Finally, what it would require to make shell more intelligent?

Apart from the shell itself, it would definitely need to know how a command is used. We can't make the shell read manual pages, they are written for humans. But what we can do is to prepare command description in a specific format, that shell would understand. One of such formats, that was used for the same purpose in VMS shell is described in this work.

The more commands shell is aware of the better experience will be achieved. Though, preparing a description even for one command takes a significant amount of time, but imagine writing descriptions for a hundred of them and keeping it up to date when new features are released.

This work focuses on the problem of automatic generation of command description in CLD language from its manual page and analysis of CLD language itself.

The first chapter covers specifics of shells, shell commands, and the introduction of CLD language. It is followed by the specification of the required extension of CLD language to describe Unix commands.

The second chapter describes Linux manual pages, it's structure, and format.

The third chapter contains a deeper analysis of the manual page, inner sections, and information that can be parsed from it. Example definition is written for `mv` command. Based on the analysis, the solution design is introduced.

The fourth chapter focuses on realization and testing. It describes used technologies, generator usage, and algorithms for each of the individual sub-

problems. The testing section of the chapter describes the process of automated testing (unit tests) and manual testing in order to evaluate the solution. In the last section further required CLD extension is introduced to fulfill requirements that occurred after deeper problem analysis.

Chapter 1

Shell commands

[4] defines shell as „the command interpreter used to pass commands to an operating system; so called because it is the part of the operating system that interfaces with the outside world.“

In order user to use shell properly, it is required to know which commands does shell support, the format of commands, and also individual command's syntax, options, and arguments.

1.1 Shell Command Format

Format of shell commands may vary, depending on concrete platform implementation. Though, terminology and order are common. Shell command starts with the name of the command, following by options and arguments separated by spaces. Options, options' arguments, and command's arguments may be optional or required.

1.1.1 Unix

Unix format of shell commands uses dash (`-`) to distinguish a beginning of short option (single alphanumeric character) and double dash (`--`) for a long option. Short options can be combined under one dash following by options without spaces (`-la`). The argument of an option is separated by an equal sign (`=`) or space depending on the fact whether it's required or not. If an argument is required, it can be separated by a space. If an argument is optional, an equal sign as a separator is required to resolve the ambiguity.

An example of Unix shell command using two short options combined under a single dash and one command argument is shown below.

```
$ ls -la --sort=size /var/log
```

1.1.2 Windows

Windows shell has its root in DOS operating system. It is case-insensitive, uses forward slash (`/`) as a start of an option and backward slash (`\`) to split path components. Unlike Unix, spaces are not required to separate command and option.

An example of Windows shell command is shown below.

```
DIR /aa C:\Windows\System32\winevt\Logs
```

1.1.3 VMS

VMS shell uses DCL (DIGITAL Command Language), which defines (similarly to Windows) forward slash (`/`) to be a beginning of an option and is also case-insensitive.

An example of VMS shell command is shown below.

```
set audit /alarm /enable=(authorization, breakin=all)
```

1.1.4 Wildcard characters

Most of the command lines support the ability to work with multiple files with help of wildcard characters. Those can represent any character(s). The process of expanding a wildcard pattern into the list of pathnames is called *globbing*. Unix, Windows, and VMS support the following wildcards:

Question mark (?) represents a single character

Asterisk (*) represents any number of characters including an empty string

Apart from those with a common meaning, there are also differences between operating systems in wildcard characters: Unix support brackets (`[]`) to represent one character of the set and curly braces (`{}`) to specify a sequence. VMS supports percentage sign (`%`) which also represents a single character and exists due to backward compatibility, hyphen (`-`) and ellipsis (`...`).

The next difference is in a place where globbing happens. On Unix systems application will receive an expanded list from shell, while in Windows and VMS patterns are passed to the application. An application then needs to expand patterns into a list itself with help of OS services.

1.1.5 Command line completion

Command line completion is a feature that helps user with typing long names of files and directories (basic completion). When user presses TAB key, it automatically fills a partially typed name based on the current context.

In Unix systems, when multiple files are matching given input, it will fill the common part of the name. Next TAB keypress will display a list of matching names.

On Windows, even if more files are matching the given input, the first one will be filled. Next TAB keypress will replace the filled input with the next matched name. This is called rotating completion.

Unix shells offer advanced configurable completion. Bash, for example, has its `complete` and `compgen` commands to manipulate the programmable completion facilities. Whether Z shell offers fully programmable completion, allowing completion of options and parameters based on its' type derived from current context and knowledge of all possible parameters and options of the current command.

Similarly to traditional Unix shells, Windows PowerShell provides customizable completion capabilities. CMD.EXE offers only basic completion of file and directory names.

VMS traditional DCL shell replaces completion with other feature. It allows commands, options, and keywords to shorten up to the shortest unambiguous strings.

1.2 Command Definition Language

Command Definition Language (CDL) was designed by Digital Equipment Company to be used with their Digital Command Language on RSX11M+, TOPS20, and later on OpenVMS. It defines how a command, its arguments, and options should be described in form of text specification, usually stored in a file called Command Definition File with .CLD extension.

1.2.1 Structure

According to [2], command definition file contains *statements*. Inside statements, *clauses* can be used to specify additional information, separated by either newline, space, or comma. Both statements and clauses can be written in multiple lines.

Following statements are supported by CDL:

1. DEFINE SYNTAX
2. DEFINE TYPE
3. DEFINE VERB
4. IDENT
5. MODULE

1.2.2 Describing command

Command in the terminology of CDL is a *verb* and is defined by DEFINE VERB statement. There is no limit for the number of verbs defined per one file.

The format of this statement is shown below:

```
DEFINE VERB verb-name [verb-clause[, ... ]]
```

where *verb-clause* is a clause specifying additional information about verb and can be one of the following:

DISALLOW Controls the use of an entity or a combination of entities.

NODISALLOWS Permits all entities and entity combinations.

IMAGE Specifies an image to be invoked by the verb.

PARAMETER Defines a command parameter.

NOPARAMETERS Disallows parameters.

QUALIFIER Defines a command qualifier.

NOQUALIFIERS Disallows qualifiers.

ROUTINE Specifies a routine to be invoked by the verb.

SYNONYM Specifies a verb synonym.

1.2.3 Describing command's option

To describe command's option, we can use *qualifier-clause*, which has following syntax: `QUALIFIER` *qualifier-name*. If an option also has an argument, *value-clause* can be used.

Similarly, to describe command's parameter, we use *PARAMETER* clause: `PARAMETER` *parameter-name*, *value-clause*.

1.2.4 Describing values

To describe a value of parameter, qualifier, or keyword, we can use *value-clause*. In parenthesis, we also can specify the value type and a flag whether it is required. For example, `VALUE(TYPE=$NUMBER,REQUIRED)` says, that the value should be a number and it is required.

The type of value can be either a built-in type or user-defined type. List of supported built-in types are described in table 1.1.

In CDL it is also possible to write a list of values separated by commas in place of one parameter or a list in parenthesis if it's a value of a qualifier. To achieve this, we need to add `LIST` statement, for example:
`VALUE(REQUIRED, TYPE=$FILE, LIST)`.

Type	Short description
`$ACL`	The value must be an access control list.
`$DATETIME`	The value must be an absolute time or a combination time. DCL converts truncated time values, combination time values, and keywords for time values (such as TODAY) to absolute time format. DCL fills blank date fields from the current system date and fills omitted time fields with zeros.
`$DELTATIME`	The value must be a delta time. DCL fills missing fields with zeros.
`$EXPRESSION`	The value must be a DCL-style expression. DCL evaluates the expression and provides the results.
`$FILE`	The value must be a valid file specification.
`$INFILE`	The value must be an existing file.
`$DIRECTORY`	The value must be a directory.
`$NUMBER`	The value must be an integer represented by either decimal, octal, or hexadecimal numbers.
`$PARENTHESIZED_VALUE`	The value must be enclosed in parentheses. Note that DCL does not remove the parentheses.
`$QUOTED_STRING`	The value must be a string enclosed in quotation marks. Note that DCL does not remove the quotation marks.
`$REST_OF_LINE`	DCL treats the rest of the line literally as the specified value, ignoring spaces or punctuation marks. DCL does not remove quotation marks when processing the string.

Table 1.1: Built-in CLD types [2]

Operator	Precedence	Meaning
`ANY2`	1	True if any two or more of the entities listed are present
`NEG`	1	True if the negated form of the entity is present
`NOT`	1	True if the entity is not present or if an entity is present by default
`AND`	2	True if both entities are present
`OR`	3	True if either entity is present

Table 1.2: Logical operators [2]

1.2.5 Defining custom types

When built-in types are not enough, we can use DEFINE TYPE statement to introduce a new type, which represents a keywords list and may later be referenced in *value-clause*. It has following syntax: `DEFINE TYPE` *type-name* [type-clause[, . . .]] where *type-name* is a name of the keyboards list and *type-clause* lists all acceptable keywords.

Each possible keyword is described in a *type-clause*. It starts with reserved word `KEYWORD`, which is followed by the keyword itself. For example, `KEYWORD GREEN`. To specify the default value we may use the reserved word DEFAULT: `KEYWORD RED, DEFAULT`.

1.2.6 Disallowing entities

CDL allows us to selectively disallow entities in a special verb clause DISALLOW. It has the following format:

DISALLOW expression

An expression is logically evaluated when the command string is parsed. Each entity in expression is checked on its presence, true - entity in expression is present, and false - entity in expression is absent. A logical operator and parenthesis can be used to combine multiple entities inside an expression. Supported logical operators are shown in table 1.2.

1.2.6.1 Specifying expression entities

To specify an entity in an expression, we need to reference it uniquely. We can do it by using:

1. A parameter, qualifier, keyword name or label
2. A keyword path
3. A definition path

Referencing the entity via its name or label is possible only if it is defined in the current definition and its name or label is unique.

To assign a label to an entity, `LABEL=`*label-name* can be used. If an entity has no assigned label, a name should be used.

When we need to reference a keyword defined in other definition than current, a keyword path should be used. Keyword path consists of (at most eight) entity names or labels that are separated by a period. The first name in a path is the name (or label) of the first entity that references the keyword's value type definition.

So, let's take a closer look at an example. Let's assume, that we have defined two parameters PARAMETER P1, VALUE(TYPE=COLORS) and PARAMETER P2, VALUE(TYPE=COLORS). Its value type is a custom value, defined as keyword list COLORS consisting of keywords RED and BLUE, the definition of which is followed the parameter definitions. Consider now we want to disallow usage of parameter P1 value to be RED, when qualifier Q is present. To do this, and not accidentally remove the support of P2 value to be RED with qualifier Q, we should specify DISALLOW Q AND P1.RED.

Name or label and keyword path are possible to use when DISALLOW command is used in the same definition as where the entities are defined. But in case we want to refer to an entity that is defined in another DEFINE statement, we will need to use a definition path. For example, a definition path is needed when a syntax definition provides new disallow clauses for parameters or qualifiers that are defined in a primary definition. A definition path has the following format:

<definition-name>entity-spec

The definition name is the name of the DEFINE statement where the entity is defined or the keyword path begins. The entity specification can be an entity name, a label, or a keyword path. The angle brackets are required.

1.2.7 Defining syntax

A quite common case is when a command has multiple valid syntaxes. For example, command `mv`, which is moving files, supports 3 of them: one for renaming and two for moving files (destination directory specified as option's argument or as command's argument).

CDL provides `DEFINE SYNTAX` statement to define an alternative syntax of a previously defined verb, which will be used based on the presence of described keywords, parameters, or qualifiers in the command string. Redefining syntax is also possible for previously defined syntax.

Defining syntax consists of two parts: referencing syntax from the affected command verb (primary DEFINE statement) and syntax definition itself (secondary DEFINE statement).

To refer to the syntax, verb clause `SYNTAX=syntax-name` should be specified. For example, `QUALIFIER LINE, SYNTAX=LINE`, which says, that when qualifier LINE is seen, syntax LINE should be used.

To define syntax itself, statement `DEFINE SYNTAX` should be used, which has following format:

```
DEFINE SYNTAX syntax-name [verb-clause[, ... ]]
```

The syntax-name verb clause is the name of the syntax. As verb clause, the same verb clauses can be used as are allowed in DEFINE VERB statement, except SYNONYM.

1.2.8 Extension of CDL to describe syntax of Unix commands

Based on preliminary analysis CDL was extended with following statements:

- `IMAGETYPE {VMS, UNIX, WINDOWS}` - to be able to distinguish between syntax of the command and way of passing command line arguments
- `UNIXOPT string` - to specify option's short name(s)
- `UNIXLIST` - to specify list of parameters separated by space
- `HELP string` - to specify help string

Chapter 2

Linux manual pages

History of manual pages in Linux according to [1] has begun in 1964, when Jerome H. Saltzer wrote a utility *RUNOFF* for MIT's IBM 7094 CTSS operating system, the original purpose of which was to format Saltzer's doctoral thesis proposal.

In 1969, McIlroy made an important release of Multics BCPL port to GECOS GE-645 computer at AT&T Bell Labs, Murray Hill, that influenced the history of *RUNOFF*. He did not refer to the CTSS RUNOFF source code in writing runoff, nor any other speculated derivatives of Saltzer's utility. *RUNOFF* started to be called shortly *roff*.

Later on, as most of the programming team of Multics continued to work with UNIX, *runoff* was incorporated as UNIX *runoff(1)* in Version 1 AT&T UNIX, 1971.

In future versions of UNIX there were plenty other implementations of this utility, such as *nroff(1)* (Joseph F. Ossanna, 1972), *troff(1)* (Joseph F. Ossanna, 1973), *ditroff(1)* (1979, Joseph F. Ossanna, Brian Kernighan).

The most popular in modern UNIX installations is the GNU *troff* - *groff*. In this work, we will primarly focus *groff*, which is compatible with *troff* implementation, but has many extensions.

2.1 Format

English text and special control words shape an input for the *troff*. Control words must be placed on a new line and start with a period to be distinguished from other text. *troff* does not print the control words.

groff provides wide range of low level operations for formatting text (font control, line length, indenting, etc.), however it might be difficult to use by itself. To simplify usage, *groff* provides a *macro* facility for routine operations (printing paragraphs, adding footnote, etc.). Macros can be combined into *package*. Most common packages are: *man*, *mdoc*, *me*, *ms*, and *mm*. [5]

Device	Short description
ascii	Text output using the ascii character set.
cp1047	Text output using the EBCDIC code page IBM cp1047 (e.g., OS/390 Unix).
dvi	TeX DVI format.
html	HTML output.
latin1	Text output using the ISO Latin-1 (ISO 8859-1) character set.
pdf	PDF files.
utf8	Text output using the Unicode (ISO 10646) character set with UTF-8 encoding.
xhtml	XHTML output.

Table 2.1: groff output devices [5]

2.2 Output

As [5] describes, in *groff* output targets are called *devices*. A device can be hardware or a software file format. List of devices shortened to file formats only is presented in table 2.1.

Simple text formats, such as ascii, utf8, and cp1047, are also extended with overstriking. So they don't miss the font style, which may be useful considering rules described in section 2.4.

2.3 Overstriking

Overstriking is a method to achieve bold and underlined font styles in a limited environment (terminal, for example) by using a backspace character. Earlier it was also used to print diacritics and other unavailable characters in ASCII, but with introducing UTF-8 encoding this feature is no longer needed.

To achieve bold font style, character is repeated again after backspace. For example, `a\ba`, where `\b` is a backspace, will produce **a**.

To achieve underlined font style, sequence underline, backspace, character is used. For example, `_\ba` will produce a.

When the whole word needs to be written in bold font style, each character is written in that way. For a sentence, white spaces are written as usual, without overstriking.

2.4 Man page structure

As [3] says, manuals are divided into sections. Those sections are traditionally defined as follows:

1 User commands (Programs)

2 System calls

3 Library calls

4 Special files (devices)

5 File formats and configuration files

6 Games

7 Overview, conventions, and miscellaneous

8 System management commands

In this work we will primarily focus on section *1 User commands (Programs)*. Now let's take a look at man page structure described in [3], specifically for section 1:

1. Man page title
2. **NAME** section
3. **SYNOPSIS** section
4. **DESCRIPTION** section
5. **OPTIONS** section (optional)
6. **EXIT STATUS** section (optional)
7. **SEE ALSO** section

Synopsis section according to [3] should describe the syntax of the command and its arguments (including options); boldface is used for as-is text and italics are used to indicate replaceable arguments. Brackets ([]) surround optional arguments, vertical bars (|) separate choices, and ellipses (...) can be repeated.

In practice, italics is also often duplicated with underlined font style.

Name section contains one or more command names and a short description that follows the name and a dash (-) character.

The synopsis section contains a symbolic description of all syntax that a command supports.

Description and/or option sections usually contain a description of all supported options. There are also a few exceptions, where options are described in different sections.

Chapter 3

Analysis and design

3.1 Input

Parsing and analyzing raw *groff* can be quite challenging due to a variety of ways to achieve the same result. Also, the resulting solution will heavily depend on the concrete version of the macros package and if a new package will be introduced, the solution will no longer work. This leads us to the analysis of output formats, which were described in section 2.2.

Though formats html and xhtml have a clear specification and are easy to parse, they still are too rich in possible ways to achieve the same result.

On the other hand, simple text formats (ascii, utf8) are quite straightforward, all advanced formats' ambiguity is represented here only in the number of white spaces and newlines between sentences/paragraphs. We are also aware, that those formats don't miss bold and underline font styles, which can be useful considering rules discovered in section 2.4.

Finally, comparing between ascii and utf8, we may find ascii more suitable for the scope of this work. While looking at all utf8 outputs generated by groff for different manual pages we have encountered at least 4 different ways to represent a dash symbol. While in ascii there are only two of them: `-` and `--`. ASCII encoding itself is quite poor, but luckily man pages are written in English, without diacritics, so it should be just enough.

3.1.1 Generating input

Manual pages are written with the usage of groff typesetting system. To analyze the input, at first, we need to prepare it. To find the sources of a manual page, we can use option `-w` of command `man`: `man -w command-name`. It will print the location of the source file of the manual page for the given command:

```
$ man -w tar
/usr/share/man/man1/tar.1.gz
```

In some cases, the source file may be compressed. With help of gunzip command we can decompress it:

```
$ gzip -cd /usr/share/man/man1/tar.1.gz > tar.1
```

Having *groff* source in `tar.1` file, let's now convert it to ASCII with overstriking. Accordingly to the examples section in [5] we can do it with the following command:

```
$ groff -P -c -man -T ascii tar.1 > tar.1.txt
```

This should be enough for input for the tool. However, for the sake of good readability in this work, we will also remove overstriking from the input. In cases where font style matters we will mention and describe it explicitly. To remove overstriking we can use `col` and it's option `-b`:

```
$ cat tar.1.txt | col -b > tar-plain.1.txt
```

Now as we set the pipeline for finding and converting man page contents to the desired format, let's take a closer look at concrete parts, that we are interested in.

3.1.2 Name

The name section contains one or more names of the command and a short description for each of them. Let's take look at the most common formats of the name section we may encounter.

- Simple with one command name. A separator between name and description also may be single or double dash. Most of commands are written in this format (`ls`, `mv`, `tar`, etc.).

  ```
  tar -- The GNU version of the tar archiving utility
  ```

- Multiple command names separated by a comma, that have the same description. For example, `login` and `logout`.

  ```
  login, logout - write utmp and wtmp entries
  ```

- Multiple command names, each having its own description. For example, `hostname`.

  ```
  hostname - show or set the system's host name
  domainname - show or set the system's NIS/YP domain name
  ypdomainname - show or set the system's NIS/YP domain name
  nisdomainname - show or set the system's NIS/YP domain name
  dnsdomainname - show the system's DNS domain name
  ```

This section is worth parsing as we can get names of all commands that are described in man page and also a short description.

Multiple types of formats can be generalized into the following one:

```
<NAME1>[, <NAME2>[, <NAME3>[, ...]] {-|--} <description>
[<possible continue of the multiline description>]
[...]
```

Specifically, one to many names are separated with comma and space. Followed by single or double dash (wrapped in spaces). Followed by a description. The description may be multi-line. We can distinguish between names-containing line and description-only line by checking if it contains name and description separator in it. All these can be repeated until the end of the section.

All command names will be useful for further parsing. As for our final output in CDL we will use the command name as a verb-name and description as a help string.

3.1.3 Synopsis

The synopsis section describes all possible usages in a short amount of text. We will break parsing this section into two problems: finding syntax line(s) per each command and parsing the syntax line itself. As a first step, we will look into defining the general format of the synopsis section and then look into ways of parsing tokens, their attributes, and meaning.

There are multiple formats that we encountered during the analysis. Those can be summarized as follows:

- Single syntax shown in figure 3.1. For example, `ls`.
- Multiple syntax related to one command shown in figure 3.2. For example, `mv`.
- Multiple syntax for different commands shown in figure 3.3. For example, `hostname`.

Now let's take a closer look at them.

```
ls [OPTION]... [FILE]...
```

Figure 3.1: Synopsis section content of ls command

Synopsis of `ls` command shown in figure 3.1 can be read as: command name (`ls`) followed by zero or more options and then zero or more files.

We should be able to recognize the command name as we know all of them from the name section. The following `OPTION` token is wrapped into brackets, which means it is optional according to [3]. The ellipsis tells us, that the preceding token can be repeated. Similarly, `FILE` token is also optional and can be repeated.

Understanding what exactly concrete token means may not be easy at this moment. Let's postpone the token understanding for now and focus on the overall structure of the synopsis section by taking a look at more examples.

```
mv [OPTION]... [-T] SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
mv [OPTION]... -t DIRECTORY SOURCE...
```

Figure 3.2: Synopsis section content of mv command

Differently to `ls`, `mv` command's synopsis shown in figure 3.2 contains multiple lines. Each line is describing syntax, i.e. unique usage. Syntaxes can differ from each other either the amount of parameter or its' types. Or it may include a special option, which triggers such specific syntax.

```
hostname [-a|--alias] [-d|--domain] [-f|--fqdn|--long]
[-A|--all-fqdns] [-i|--ip-address] [-I|--all-ip-addresses]
[-s|--short] [-y|--yp|--nis]
hostname [-b|--boot] [-F|--file filename] [hostname]
hostname [-h|--help] [-V|--version]

domainname [nisdomain] [-F file]
ypdomainname [nisdomain] [-F file]
nisdomainname [nisdomain] [-F file]

dnsdomainname
```

Figure 3.3: Synopsis section content of hostname command

`hostname` command synopsis section has even more syntaxes defined for more than one command: 3 for `hostname` and 1 for every other. This is

an example of one of the most complicated synopsis content that we have encountered. As we can also see, the syntax may extend to more than one line. It's easy to distinguish as those lines do not begin with the command name.

Generalization of formats described above will look as follows:

```
<COMMAND-NAME> <SYNTAX-LINE>
[<SYNTAX-LINE-CONTINUES>]
[...]
```

For each of the command names that we already know we should be able to parse corresponding one or more syntax lines. Syntax line may also be written on multiple lines. We should be able to recognize the end of multi-line syntax by checking the prefix of the line on containment of one of the command names. That will mean a new syntax line is starting.

Having prepared syntax lines per command name, we are now ready to proceed with syntax line parsing.

By examining `ls` command synopsis shown in figure 3.1 we have already identified several rules: brackets (`[]`) means token(s) wrapped in it is optional and ellipsis (`...`) means preceding token(s) can be repeated.

Synopsis of `hostname` also contains not yet covered symbol - vertical bar (|). Which represent separate choices. In other words, only one of the tokens separated by a vertical bar can be used.

Let's now step back and return to the problem of token understanding we mentioned at the beginning of this section.

We can be sure, that tokens that start with a double dash (for example, `--alias`) represent a long option, as well as tokens starting with a single dash and containing a single character (for example, `-a`) represent a short option.

Tokens, that start with a single dash, but are longer than one character (for example, `-aABcCdeEfFg`) we can treat as short options joined under a single dash. But unfortunately, some commands define long options with a single dash, for example, `gcc`. To resolve this ambiguity, we require additional information: a list of known options. Imagining that we have already a list of all known options, we can detect which case is it by searching option with name as a token (stripping the leading dashes, of course). This leads us to one of the requirements for the solution: options should be parsed earlier than the synopsis.

And the last type of token is a token that does not start with a dash. By default, those can be treated as an argument with few exceptions.

Token `OPTION`, that we saw in `ls` synopsis section, shown in figure 3.1 is one of them. Here we can assume, that `OPTION` token represents the replaceable token for all common options. By iterating through man pages of standard UNIX commands, we may also find several aliases for it: `option` and `options`.

Name	CLD type
FILE, *FILENAME*	`$FILE` or `$INFILE`
DIR, *DIRECTORY*	`$DIRECTORY`
NUM, *NUMBER*	`$NUMBER`

Table 3.1: Commonly used replaceable tokens and matching CLD type

The second exception arises from the fact, that there are valid options that do not need to be preceded by dash at all. For example, `A` in `tar` command. This problem again leads us to search the list of known options. If we are able to find an existing option with such name, it's an option, otherwise, we can be confident it's an argument.

When a token is treated as an argument, there is one more potential ambiguity. Let's take a look at token `DIRECTORY` in the third syntax of `mv` command's synopsis shown in figure 3.2. Until we don't know whether option `-t` has an argument or not, we are not able to say whether it's an option argument or a command argument. To identify, we will need to find a known option with the name of a preceding token if any. In case of `-t` it has an argument, so we say `DIRECTORY` is an option argument. But next token `SOURCE` is a command argument.

The next step we can do is to guess the type of the argument from its name, such as file, directory, user.

Iterating through manual files of all standard commands, we can make table 3.1 of commonly used replaceable tokens and match them with corresponding CLD types.

Our assumption may help detect a few types of arguments, but not all of them. In order to make a flexible solution, we will introduce the ability for the user to append additional input. We can accept optional input from the user in form of additional definitions file, where he can match replaceable tokens with correct CLD types. The user may often see repeatable definitions, so we probably want to accept this input in two forms:

- Global user definitions file, for generally applied definitions
- Local user definitions file, for command-specific definitions

The definitions file will be a primary method of how to adjust the output of the program to the desired one and is described in more detail in section 4.3.

To sum it up, synopsis section of command manual file includes useful information, such as:

- Syntax
- Options that trigger specific syntax

- Option constraints
- Arguments

3.1.4 Description

Description section includes, apart from the rest, definitions of command's options - name(s), description and arguments.

```
List  information  about  the FILEs (the current directory by
default).
Sort entries alphabetically if none of -cftuvSUX nor --sort
is specified.

Mandatory  arguments  to  long  options are mandatory
for short options too.

-a, --all
    do not ignore entries starting with .

-A, --almost-all
    do not list implied . and ..

--author
    with -l, print the author of each file

-b, --escape
    print C-style escapes for nongraphic characters

--block-size=SIZE
    scale sizes by SIZE before printing them; e.g.,
    '--block-size=M' prints sizes in units of
    1,048,576 bytes; see SIZE format below

-p, --indicator-style=slash
    append / indicator to directories
```

Figure 3.4: Part of description section of ls command

Figure 3.4 shows a part of `ls` description section content. At first sight, we may notice, that option definition starts with a line that starts with a dash.

Unfortunately, limiting criteria to only this condition may result in too many false-positive results.

That's why here we will also look at font style on an example of one of the definitions of the options:

−p, **−−indicator-style**=slash
append / indicator to directories

So, to reduce false positives we, additionally to starts with dash condition, may also check that it starts with the bold font style. This will reduce false positives to almost zero.

The first line contains a symbolic definition of an option and the rest of the lines in the paragraph are a human-readable description of the option.

The symbolic definition contains zero, one or more short names, and zero, one or more long names. Names are separated by comma and space.

If an option has an argument, after the last name follows an equal sign or space and argument name, written in underline font style, meaning replaceable text. To detect the argument's type we can use the definition file.

It may also happen, that option's argument is optional. Then whole argument part, starting from equal sign and ending with argument type is wrapped into brackets, for example **−−color**[=WHEN].

After the first line with the signature goes a human-readable multi-line description of the option. The description ends either right before the next option begins, after an empty line, or until the end of the file.

Pure output of description section is a list of options, where for each option we can parse:

- Short name(s)
- Long name(s)
- Argument name
- Whether argument is required
- Human-readable description

3.1.5 Combining output from synopsis and description sections

In the analysis of the synopsis section, we have already mentioned, that it requires a list of known options to resolve ambiguity in several cases. But at that moment we left a special replaceable token *OPTION* without attention. As a reminder, *OPTION* token we treat as a placeholder for all common options. We saved its attributes and now that we have all syntaxes parsed, we may proceed further and replace them with common options.

Common options, or in other words, options that are not syntax-specific can be derived from a set of all known options subtracting a set of all syntax-specific options.

3.2 Output on example of mv command

In this section, we will focus on the problem of writing CLD command definition from outputs of the parsing stage. We will analyze the process on the example of concrete command and then will think of more generic steps to achieve the same result for any type of command introduced in the previous section. We will also rely on CLD specification introduced in section 1.2.

3.2.1 Name and verb definition

```
NAME
    mv - move (rename) files
```

Figure 3.5: Name section of mv command

The output from the name section of the manual page of `mv` shown in figure 3.5 is a short command description and command's name. We can now start defining verb, as shown in figure 3.6. The verb has the name of the command and a short command description as a help string. It also contains additional information, such as image (can be retrieved with help of builtin `which` command) and image type (hardcoded to unix for now).

```
DEFINE VERB mv
    IMAGE "/bin/mv"
    IMAGETYPE unix
    HELP "mv - move (rename) files"
```

Figure 3.6: Name section of mv command

3.2.2 Synopsis

Synopsis section shown in figure 3.2 contains description of three syntaxes. All three contain optional, replaceable generic placeholder *OPTION*. First syntax

then defines optional *-T* option and two required, non-repeatable arguments: *SOURCE* and *DEST*. Seconds syntax only defines two required arguments: repeatable *SOURCE* and non-repeatable *DIRECTORY*. The third syntax line is ambiguous, there are two possible variants. It defines required option *-t*, required argument *DIRECTORY* and required repeatable argument *SOURCE*. But are not able to say whether argument *DIRECTORY* belongs to option *-t* or command itself without a list of known options. We need to look *-t* option up in the description section and find out that it has an argument.

Now we can extend our CLD definition with this information as shown in figure 3.7.

There are few conventions we agreed upon to proceed through the limits of the CDL.

- First, verb definition itself in CDL counts as a syntax, and *syntax* clause in CDL is taken as an alternative usage definition. So our first syntax is defined in the verb clause and left two syntaxes with help of the syntax clause.

- Secondly, options with only short name are named with prefix *OPTION_* and as CDL is a case insensitive language we agreed that upper case letters have naming prefix *OPTION_UPPER*. The short name of the option is defined in a special *UNIXOPT* clause.

- Thirdly, the type of values will be the same as its corresponding replaceable token. Later, the user may alternate its type with help of a definitions file by adding an entry defining a type for a given replaceable token.

It also worth mentioning, that even if an option belongs to a single syntax, CDL requires that it's also defined in verb clause and then repeated in syntax clause. That is why, for example, *-t* in this case is defined in two places.

3.2.3 Description

Finally, by processing a shortened version of the description section shown in figure 3.8 and replacing types with builtin in CDL ones, we end up with definition listed in appendix C.1.

3.3 Architecture

This section covers the architecture of the application. We will go through current processes, requirements, and use-cases. The output of this section should be the first architecture draft.

```
DEFINE VERB mv
    IMAGE "/bin/mv"
    IMAGETYPE unix
    HELP "move (rename) files"

    QUALIFIER OPTION_UPPERT
        UNIXOPT "T"

    PARAMETER P1, LABEL=SOURCE
        VALUE (REQUIRED, TYPE=SOURCE)

    PARAMETER P2, LABEL=DEST
        VALUE (REQUIRED, TYPE=DEST)

    QUALIFIER OPTION_t, SYNTAX=syntax3
        UNIXOPT "t"
        VALUE (REQUIRED, TYPE=DIRECTORY)

DEFINE SYNTAX syntax2
    PARAMETER P1, LABEL=SOURCE
        VALUE (UNIXLIST, REQUIRED, TYPE=SOURCE)
    PARAMETER P2, LABEL=DIRECTORY
        VALUE (REQUIRED, TYPE=DIRECTORY)

DEFINE SYNTAX syntax3
    QUALIFIER OPTION_t
        UNIXOPT "t"
        VALUE (REQUIRED, TYPE=DIRECTORY)
    PARAMETER P1, LABEL=SOURCE
        VALUE (UNIXLIST, REQUIRED, TYPE=SOURCE)
```

Figure 3.7: Syntax definition of mv in CLD

3.3.1 Process

The current state can be described as follows. When a user wants to use the CLD command line utility he manually prepares the CLD definitions and builds the utility from the source codes.

Manual preparation of CLD definitions involves the following steps:

- finding a manual file for the command that is being integrated

```
DESCRIPTION
    Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

    Mandatory  arguments  to  long  options are mandatory
    for short options too.

    ...

    -b     like --backup but does not accept an argument

    -f, --force
        do not prompt before overwriting

    ...

    -S, --suffix=SUFFIX
        override the usual backup suffix

    -t, --target-directory=DIRECTORY
        move all SOURCE arguments into DIRECTORY

    -T, --no-target-directory
        treat DEST as a normal file

    -u, --update
        move  only  when the  SOURCE file is newer than the
        destination file or when the destination file is
        missing

    ...
```

Figure 3.8: Shortened version of description section of mv command

- reading a manual file
- writing CLD definition

Our goal is to enhance this by automating part of the process of writing CLD definitions.

3.3.2 Requirements

The result of this work should be a tool that fulfills the following requirements:

- Tool should generate CLD definitions of commands from corresponding Linux manual page in SUSE Linux distribution
- User should be able to alternate output by providing additional input with tokens to replace or type definitions to include

3.3.3 Use-cases

In the scope of this work, we will target a single use-case: the user wants to create a CLD definition of command from the manual file.

This use-case can be though split into multiple:

- user wants to generate CLD definition from manual file for the first time without additional input
- user already have generated CLD definition and now want to apply some changes to produce better output by providing additional input

3.3.4 Design

Application will have streamline flow with following steps:

1. Parsing arguments
2. Reading user definitions file if any
3. Reading man page contents
4. Splitting man page contents into sections
5. Parsing **NAME** section to retrieve information about all command names described in man page
6. Selecting one command name for the output
7. Parsing options from predefined sections (**DESCRIPTION**, **OPTIONS**)
8. Parsing **SYNOPSIS** section and splitting it's content into syntax lines per command name
9. Parse syntax lines of selected command name
10. Merging syntaxes and options. Creating command definition
11. Writing command definition in CLD to file

Chapter 4

Realisation

This section describes implementation details, such as technology choices, limitations, and algorithms.

4.1 Technologies

The existing command line utility, which takes CLD definitions as input is written in C. Because the output of this work may be later integrated into CLI, we have chosen to write CLD generator tool also in C with C99 standard.

The solution has no external dependencies. The only dependencies are standard libraries: stdio, stdlib, string, unistd.

To build it from source codes `make` utility and GNU C compilers are required.

4.1.1 dclsh shell prototype

The output of the solution will be tested via one of the components of dclsh shell.

dclsh shell consists of two components:

1. CDU (Command Definition Utility) - a compiler of command description in CLD to internal form. At the moment of the start of this work, it supported extensions described in section 1.2.8. During the implementation of the solution of this work, CDU was twice extended: to support dashes in identifiers and multi-line string literals, later to support alternative syntaxes and further special features of several Unix commands discovered to be necessary after the first usage of the tool.

2. dclsh itself is a shell, so far in its early stages of development. Originally created for realization of a unified command line interface of FC network simulation. Consist of three sub-components:

a) Editor for interactive command input

b) Command parser, which converts it into internal form according to the compiled version of CLD specification

c) Library routines for the interfaces of internal commands

So far, there are no modules available for running external commands, redirecting input and output, working with variables or helper functions for expanding the file specification, automatic completion of incomplete options and text constants, or sensitive hints. These will emerge from the follow-up to this project.

Therefore, the following was used to test the correctness of the function:

- CDU utility for syntax and completeness checks (also detects referenced missing parts)
- Manual checks for factual correctness

4.2 Supported options and arguments

The application expects one required argument - path to the text file with contents of man page. The text file should be in ASCII with overstriking encoding. Steps on how to generate such file were described in section 3.1.1.

The application also supports few non-required options that are shown in table 4.1.

4.3 Reading user definitions files

User may optionally provide two definition files: global and command-specific, or just any one of them.

By default, the application expects the global definitions file to be located in the working directory and have a name `global.def`. Command specific definitions file should be located in the working directory and have the name in format *command-file-name*`.def`. So, for example, if input file argument is `/usr/share/man/man1/ls.1`, it will search for `ls.1.def` file. Default values can be overwritten by specifying it explicitly via option `--global-definitions` or `--definitions` respectively.

4.3.1 User definitions file format

User definitions file was introduced for two reasons:

- to solve human-readable description ambiguity problem for enumeration types

Option	Description
`--generate-definitions FILE`	Generates definitions file template.
`-c, --command-name STRING`	Specify command name to generate output for in case when multiple commands were found in man page. Useful, for example, for command `hostname` or `logout`.
`-g, --global-definitions FILE`	Specify global user definitions file. Default value is `./global.def`.
`-d, --definitions FILE`	Specify command-specific user definitions file. Default value is `<INPUT-FILE>.def`.
`-o, --output FILE`	Specify output file. Default value is `<INPUT-FILE>.cld`.
`-l, --log-level {DEBUG,INFO,WARN,ERROR}`	Specify log level. Default value is `WARN`.
`-b, --debug`	Alias for `--log-level DEBUG`.
`-v, --verbose`	Alias for `--log-level INFO`.
`-h, --help`	Shows help.

Table 4.1: Supported options

- to replace occurrences of non-existing types from man page with supported by CLD types

These reasons lead to the following requirements:

- user should be able to change type in generated CLD
- user should be able to define enumeration type

Additionally, we wanted to keep the ability for the user append to manually prepared CLD blocks.

Finally, requirements analysis resulted in the following format of definitions file. A definitions file is a text file containing an unlimited amount of definitions of three types: enumeration, type replace rule or CLD append rule.

Enumeration definition is written in format: *enum-name* = (*enum-value1* ["*enum-value1-description*"] *enum-value2* ["*enum-value2-description*"] ...). In other words, it starts with a name, followed by equal sign, followed by enumeration values optionally followed by description in quotes separated by whitespace(s) and wrapped in parenthesis. Examples:

```
FORMAT = (gnu "GNU tar 1.13.x format"
    oldgnu "GNU format as per tar <= 1.12)"
    pax "POSIX 1003.1-2001 (pax) format"
```

```
    posix "same as pax"
    ustar "POSIX 1003.1-1988 (ustar) format"
    v7 "old V7 tar format")

DATE-OR-FILE = ($DATE $FILE)
```

Type replace rule is written in format: *type-name-to-be-replaced* = *type-name-to-replace-with*. Example: `color_WHEN = WHEN`.

In case that a type represents a list of values, `[]` suffix can be used. For example, `userlst = $UID[]`.

CLD append rule has a beginning, defining a place where CLD should be appended. Then follows the piece of CLD, wrapped into curly brackets. Possible formats are shown below:

```
QUALIFIER <qualifier-name> {
    <CLD to append to the end of qualifier statement>
}
SYNTAX <syntax-name> {
    <CLD to append to the end of syntax statement>
}
VERB_HEADER {
    <CLD to append to the beginning of verb statement>
}
VERB {
    <CLD to append to the end of verb statement>
}
END {
    <CLD to append to the end of tar statement>
}
```

Lines that start with an exclamation mark are treated as comments. Example: `! This is a comment`.

4.3.2 Parsing

Parsing such file is pretty straightforward. We iterate through lines of a file and:

- skip the line if it starts with an exclamation mark because it's a comment
- search for equal sign position
- if an equal sign is presented on the line
  - if there is an opening parenthesis after equal sign then it's an enumeration definition, otherwise, type replacement rule

  - for type replacement rules we need two tokens - before the equal sign and after the equal sign, both with whitespace trimmed
  - for enumeration we take token on the left side of the equal sign as a name, now we need the values part
  - we search for a closing parenthesis in the right part of string and track quotes, so that we don't treat parenthesis in the description as a closing parenthesis of enumeration values
  - if we iterated through the whole string and didn't reach closing parenthesis we read next line from the file and do the previous step again until we find it
  - when we have found bounds of enumeration values part we begin extracting values and optionally their description from it by iteration over characters with known rules: value goes first, then whitespace, then description if the first character is a quote, otherwise it's next value

- If equal sign is not presented on the line then case insensitively compare prefix of the line
  - If the prefix is `VERB_HEADER`, `VERB` or `END`, read the block of text until closing curly parenthesis and store this block in corresponding property of definitions structure.
  - If the prefix is `SYNTAX` or `QUALIFIER`, read next word (skipping leading whitespaces) - this is a name. Then read the block of text until closing the curly parenthesis. Store this block and a name in the corresponding key-value pair array.

4.3.3 Merging

When both command-specific and global files are present, we merge definitions into one for the sake of simpler further usage. After reading command-specific definitions, we append rules from global definitions, that are not already present.

Presence is given by the condition that definitions have an element with the same name. This results in command-specific definition having a higher priority so that rules located there can overwrite rules from global definitions.

4.4 Reading man page contents with overstriking

For further steps we will need text content of a manual page in two forms:

- Text with font style attribute
- Plain text

As discussed in section 3.1.4, font style attribute will improve results of option parsing by making more specific criteria on how to find an option and so reducing false positives. To have a font style attribute alongside the text block that it belongs to, we introduce rich text blocks and a utility function to parse overstriking, with an array of rich text blocks as it's output.

A rich text block is a simple structure, that stores plain text content and font style: normal, bold, or underline.

Parsing text with overstriking is achieved with the algorithm described in the following steps:

1. Begin with an empty array of rich text blocks and current block that have font style set to normal and empty content

2. Iterate through all character in the input string with steps 3-6

3. Detect font style of the current character

   - *bold* if next character is a backspace followed by the same character as current
   - *underline* if the current character is an underscore and the next character is a backspace
   - *normal* otherwise

4. If currently saved font style is not equal to detected character's font style and current block's content is not empty, push it to the resulting array of rich text blocks and reset current block to empty

5. Copy the character to the current block set block's font style to the font style of the character

6. In case current font style is not *normal*, skip 2 more characters

For the rest of the parsing stages, it will be enough to have plain text. Converting text with overstriking to plain text is as simple as removing all occurrences of a backspace character and it's preceding character.

4.5 Parsing man page contents and creating command definition

This section will go through implementation details of steps 4-10 of application flow described in section 3.3.4.

4.5.1 Splitting content into sections

The next stages require the content of a certain section as an input. Therefore, we need to process the content of man page and split it into sections. As an output, we will have an array of key-value pairs, where the key is a section name and the value is a section content. Having an output in such form will fulfill a prerequisite for necessary operation: to get content of a certain section by its name.

Every line in the input file can be classified as one of the following three: header, a section name, or section content. The header is always located on the first line, this we will skip. Section name has no indentation apart from the section content, which follows the name and has an indentation.

We iterate through lines of the input file, skipping the header. We keep a record of the current section name and its content. If a line has an indentation, we append it to the current section content. Otherwise, we push the current section if it's not empty into the resulting key-value array, reset the current section, and set the name of the current section to be the content of the current line. At the end of the cycle, we push last time the current section if it's not empty.

4.5.2 Name section

As the output from the name section, we expect to have an array of command names and corresponding description to each command name. Name section format was described in section 3.1.2. Font style doesn't matter, so we remove overstriking from the whole section content as described in section 4.4.

Each line in the name section can be classified as one of two: a line containing command name(s) and description or a continuation of the description of command names from the lines above. To distinguish between those two types of the line we check for a separator: dash wrapped with spaces (" `-` ") or double dash wrapped with spaces (" `--` ").

When a separator was found, meaning it's a line that contains command names, we split the string into two parts using the found separator: command names and beginning of the description. The first part then is being split again with separator ", " and the result is taken as an array of command names. We push each command name to the resulting array and as a description of each of them assign the second part of the string.

When a separator was not found, we append the whole line to the description of each of the command names that were found last time.

4.5.3 Selecting command name

When multiple command names were found in the previous section, we need to select one of them to process only parts related to it in the next steps. For example, `hostname` man page content contains multiple command names and

also multiple syntaxes in the synopsis section. Not every syntax described in the synopsis relates to all command names.

Command name specified via `--command-name` option is used if it is presented in parsed command names array.

Otherwise, we try to find such command name, that is contained in the input file name. If nothing passes that criteria, we select the first command name in the array.

4.5.4 Parsing options

Options may be found in *DESCRIPTION* or *OPTIONS* sections. For each of those sections, we repeat the same process, merging the results.

The strategy of parsing section content will rely on the format of the option described in section 3.1.4. In practice, we also discovered, that there are exceptions from the previously described format. For example, in `find` command where the option is written in normal font style instead of bold. To support such a few exceptions, we have removed the font style condition.

We iterate through lines of section content, trim whitespaces from both sides for each line and look at the prefix of the line. If it starts with a dash and the previous line was empty, we treat this line as the line with option definition.

There are cases though when even the option definition line contains a description of an option. To distinguish between help text and argument, we additionally check the number of whitespaces as in most cases helper text is separated with multiple whitespaces. If it's not the case, like for example, in option `--help` in `ls` command, the user still can tell that it's not an argument via special type replacement rule described in 4.3.1. In such case, the argument will be removed during the type replacing phase and its name will be inserted at the beginning of the help string.

Processing option definition is done in two steps:

1. Extracting and classifying symbols
2. Creating option entities from classified symbols

There are 3 types of symbols: short name, long name, and argument name. There can be zero, one, or more short names as well as long names. Argument name can be only one.

The first step of processing is a continuous iteration of characters of every block. We collect regular characters into a symbol. Then, when reached the end of the rich text block or reached special characters, such as brackets, comma, whitespace, and equal sign, we finalize the current symbol by detecting its type and moving its content to the corresponding array of symbols of that type.

Symbol classification uses the following rules:

- symbol is a *long option* there are two leading dashes or one leading dash and its length including leading dashes is bigger than 2
- symbol is a *short option* if either there is one leading dash and its length including leading dashes is 2 or if font style is *bold*, it has no leading dash and it's length is 1
- otherwise symbol is an *argument*

Apart from other special characters, reaching brackets also marks argument as optional - that's the only one use case discovered when brackets are used.

The second step of processing is creating option entities from symbols. From all of the long name symbols an option entity is created with the first short name symbol if any. From the rest of the short name symbols are then created options with short name only, without a long name. Finally, all created options are pushed into the resulting array.

The rest of the blocks that were not marked as option definition are taken as an option description. Also, the rest of the lines until an empty line (end of a paragraph) or until a line that starts with a dash are treated as an option description. Those lines are being converted to plain text and copied to the description property of the last created options if any.

The output of this stage is an array of known options. Each option may have a short name, long name, description, argument, and argument attributes (optional, repeating).

4.5.5 Synopsis section

The synopsis section consists of syntax lines that follow command names. Its format is described in more detail in section 3.1.3. From this section we need to get syntax lines per command name. Uses plain text content, without overstriking.

Having command names prepared we can easily parse the synopsis section so that for each command name we have an array of syntax lines. We iterate through lines of synopsis section content, looking at the beginning. If the line has some of the command names as a prefix, it's treated as a beginning of the syntax line for that command. A syntax line can be written on multiple lines, so we continue reading until we find another line with the command name as a prefix, or until an empty line. Then the process repeats until we reach the end of the section.

4.5.6 Syntax line

Now we have selected a command name and syntax lines per command name. We ignore syntax lines of other command names and only parse syntax lines

related to the selected command name. We also use an array of known options as an input to resolve ambiguity, as was discussed in section 3.1.3. The algorithm described above is repeated for each of the syntax lines.

Parsing syntax line is divided into two problems:

- Extracting symbols from the line and storing it in a suitable data structure
- Building a syntax from the result of the previous step

To store symbols a tree data structure was chosen. It allows being more flexible in defining rules on how to build a syntax out of a tree of symbols and symbol classification rules. It will be further referred to as a syntax tree.

Syntax tree node can be one of the following types:

- Symbol. A leaf node, containing a symbol.
- Optional. A node, which marks all its children as optional. Expected to have only one child.
- Repeating. A node, which marks all its children as repeating. Expected to have only one child.
- Sequence. A container node for multiple tokens on the same level.

Following steps describe how to build a syntax tree out of syntax line:

1. We start by creating a root node of type *sequence* and setting pointer current node to a root node
2. Iterate through characters of the syntax line with steps 3-
3. Examine the current character:
   - If it's opening bracket (`[`)
     a) If current node is *symbol*, close it by setting current node pointer to the parent of current node. This operation will be further referenced as *close symbol node*.
     b) If current node is not of type *sequence*, create a *sequence* node, replace current node in it's parent with the *sequence* node, set current node as a child of the *sequence* node. Finally, change current node pointer to the *sequence* node. We will further reference this operation as *insert a sequence node*.
     c) Create an *optional* node.
     d) Add *optional* node as a child of current node.
     e) Set current node pointer to the *optional* node.

   f) Create a *sequence* node.
   g) Add *sequence* node as a child of current node.
   h) Set current node pointer to the *sequence* node.
- If it's a closing bracket (`]`)
   a) Do *close symbol node*.
   b) Traverse up until *optional* node is found.
   c) Set current node pointer to the parent of *optional* node.
- If it's a comma or a dot and together with following characters it makes string `,...` or `...`
   a) Do *close symbol node*.
   b) If the current node is a *sequence* create and insert a *repeating* node instead of the last child, that last child put as a child of the *repeating* node, don't change the current node pointer.
   c) If current node is *optional* or *symbol* create a *repeating* node, replace current node as child in it's parent with *repeating* node, and insert current node as a child of *repeating* node. Set current node pointer to *repeating* node.
- If it's a whitespace do *close symbol node*, then *insert a sequence node*.
- If it's a character in a certain range of allowed characters (alphanumeric, `+-_~#?=:/@`)
   a) If the current node is a *symbol*, append the character to the symbol.
   b) Otherwise create a *symbol* node, set the character as a symbol, add as a child to the current node.
   c) For both cases set current node pointer to *symbol* node.
- Otherwise do *close symbol node*.

Example of a syntax tree build from third syntax line of `mv` command (see figure 3.2) is shown on figure 4.1.

Next, we build a syntax out of the syntax tree. The syntax is a structure for the internal representation of command syntax to be further used for the output writing in CDL. It contains an array of options and an array of command arguments. So the goal is to extract symbols from the tree, classify them, convert them to corresponding representation, set the attributes, and append them to the syntax.

We traverse the tree recursively from the root to the leaves. While traversing, we might find *repeating* or *optional* node. At that moment we set the corresponding flag (is repeating, is optional) so that when we reach a leaf, which is always a *symbol*, we can set those values as attributes.

When reached a symbol, we classify it by following rules:

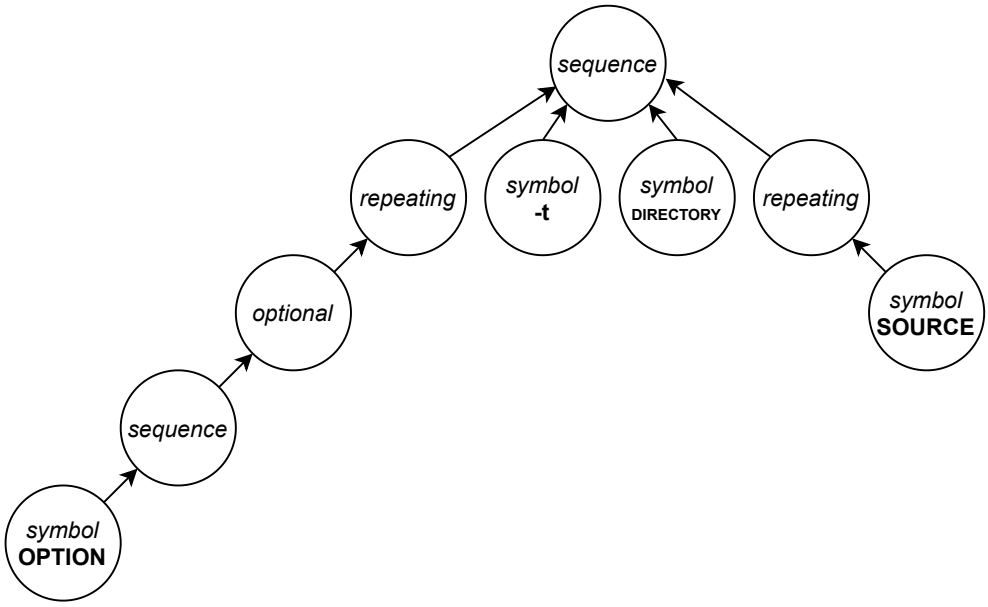

Figure 4.1: Syntax tree of third syntax of mv command

- It's an option if there is an option from known options input array, which short or long name matches the symbol without leading dashes.
- It's a special placeholder for any common options if its name without leading dashes matches string OPTION, option or options.
- It's an array of short options if it has one leading dash and its length without leading dashes is more than 1.
- It's an argument if it has no leading dashes and:
  - It's an option's argument if a right-most symbol to the left of the current node is a known option and it has an argument.
  - It's a command argument otherwise.

Symbols that were classified as an option or command argument are converted into the corresponding internal structure. Their name is set to the symbol value without leading dashes. Their attributes' values are set to be corresponding flag values. Finally, they are appended to the syntax.

4.5.7 Merging options and syntaxes

At beginning of this stage, we have extracted syntaxes from the synopsis and an array of options. For each option, we now also need to detect option type:

whether it's a common option or a syntax-specific option. Then we can replace placeholder options with common options.

This stage is focused on merging the array of options into individual syntaxes with the following rules:

- Syntax, that contains placeholder option should receive all common options which attributes copied from the placeholder option, placeholder itself should be removed
- Syntax specific options should receive attributes (like argument, description, name, short name) from options array
- Each option should be marked as syntax specific or common
- Only primary (1st) syntax should contain common options, secondary syntax contain only specific options

To do that we follow the algorithm described below:

1. Detect all syntax specific options and store it in an array as references, also store information to which syntax it is related
2. Iterate over all syntax with steps 3-8
3. Initialize new array of options for the current syntax
4. Search for a placeholder option in current syntax options and store a reference to it
5. Iterate over all options array
6. If we reach syntax-specific option, we check whether it's related to current syntax and if yes, append this option to the resulting array and copy all valid attributes from synopsis version
7. If we reach common option and the current syntax is primary syntax (i.e. first), we append this option to the resulting array and copy all attributes from the placeholder option if it exists
8. Replace syntax options array with newly created, continue to next syntax from step 3, if no syntax left continue to the next step
9. Iterate over all syntax and remove arguments that have the same name as arguments of its options.

As a next step, we give each syntax a name, so it's easier to reference it later in CLD. Following rules are used:

- Primary syntax is always named *syntax1*

- Secondary syntaxes are named after their unique options. Long name if exist, otherwise short name. For example, syntax that have it's unique option *target-directory* will be named *target-directory*.

- If secondary syntax has no unique options name is *syntaxN*, where N is order number.

4.5.8 Arguments type detection

Now that we have syntax, options, and arguments we want to proceed to the next step - type detection and type matching with the user definitions file.

As a first step, we iterate over all arguments (command's and options') and collect information about how many occurrences of the same name we have. Then we iterate over all arguments again and set a type to its name if it only appears once or if it's a command argument. If it appears more times and it's an argument of an option, we set a type to the format *optionName_argumentName*. Doing it this way we allow a user to specify different types for different options depending on a context. It's quite usual, that options have the same argument name, but their types are different.

4.5.9 Replacing types from user definitions file

User definitions file allows a user to specify simple type replacement rules. This step is implementing it. For each argument type of all arguments and all options which have arguments in all syntaxes we search for a rule in user definitions and if there is, we replace the argument type with the one that rule defines, if there are no rules we skip it and go to the next argument type.

4.6 Writing the CLD command definition to a file

Command definition is parsed and stored in an internal data structure. Now it's time to write it to the output file in CDL. Default output file name format is *inputFileName*`.cld`. We begin with opening the file in write mode, checking for errors, and proceed to the actual output.

4.6.1 Enumeration types definition

At first, enumeration types from user definitions are being written to the file. Enumeration declaration begins with `DEFINE TYPE typeName`. Then follow values with `KEYWORD value, NONNEGATABLE`. Optionally followed by value description: `HELP "help string"`. We use qualifier clause `NONNEGATABLE` here to disallow negating by adding `NO` to the name as it's not what is supported by default.

4.6.2 Verb definition

Next, we proceed to the verb definition which is mostly based on information from the name section. We define verb by using statement
`DEFINE VERB commandShortName`. Then we try to get the command's location by calling system command `which`. If we receive output without errors we specify an image location by using verb clause `IMAGE "pathToExecutable"`. After this, we specify the image type as unix with clause `IMAGETYPE unix` and escaped command description as help: `HELP "escapedCommandDescription"`. The header of the verb definition is now ready. We may proceed to the next step - defining syntaxes.

4.6.3 Syntax definition

As we know from section 1.2.7, CDL treats syntax as an alternative way to use verb. We on the other side, have an array of equal syntaxes rather than having one primary and all others as an alternative. It is required in CDL to define a verb and all possible qualifiers even syntax-specific ones. Alternative syntaxes are defined later and contain qualifier definition again, but only of those specific. So we will treat the first syntax in the array as a primary and all others as an alternative. When merging options to syntaxes described in section 4.5.7 we already did most of the preparation phase, so that we have common options contained only in the first syntax, and now it should be clear why we did it that way.

Syntax definition starts with `DEFINE SYNTAX syntaxName` statement.
Which we will skip for the first (primary) syntax so that it's content belongs to the verb definition we started previously. Then we iterate over options in current syntax and define CDL qualifiers out of them. And after we iterate over arguments and define CDL parameters. Finally, we append CLD that user definitions have for current syntax or verb if it's primary syntax.

There is also one exception in syntax definition: CDL does not support syntaxes without qualifiers as there is no direct reference from the verb to the syntax. To resolve the problem, we have introduced `ALTSYNTAX syntaxName` statement, which allows us to reference syntax without qualifiers from the verb definition. This way the first syntax without options is referenced from the verb, the next one from the previous one, and so on.

4.6.4 Qualifier definition

To define qualifier we use clause `QUALIFIER name`. Due to case insensitivity we have agreed to rename options by applying rules described in section 3.2.2: options with short names only will be renamed to `OPTION_`l or `OPTION_UPPER`L where l is a lowercase letter as a short name and L is a uppercase letter as a short name. If current option is a syntax-specific we specify a syntax name with a `SYNTAX=syntaxName` clause. If the option has a short name we specify

it with `UNIXOPT "shortName"` clause. Then we specify argument if any by using `VALUE (REQUIRED, DEFAULT="defaultValue", TYPE="typeName")`, depending on attributes of an argument we may skip the required and/or default value. Next, if the option has a description we specify it with statement `HELP "description"`. Lastly, we append CLD that contains user definitions have for the current qualifier.

4.6.5 Parameter definition

To define parameter `PARAMETER Pn, LABEL=paramName` is used, where agreed naming convention is that parameters are named Pn, where n is an order number of the parameter and original parameter name is used as a label. Then follows value specification `VALUE (REQUIRED, DEFAULT="defaultValue", TYPE="argumentType")`, where required and/or default value again skipped depending on attributes of the argument. If the argument is marked as repeated, `UNIXLIST` statement is also written in `VALUE` clause.

Using steps described above we have specified verb containing parameters of the first syntax and all qualifiers: common and syntax-specific. Then we specified all other syntaxes with their parameters and their syntax-specific qualifiers. That should be it, so lastly we close the file.

4.7 Testing

Testing correctness of the generator is done via unit tests and manual tests. Makefile define a target `test`, which builds an application with different main function declared in `main.tests.c`. Running tests is as easy as calling `make test`. There is also a `memory_test` target to execute all tests under `valdrind` environment with leak checking enabled.

4.7.1 Splitting content into sections tests

Splitting man page content into sections logic is tested on input data, that contains a header, 3 sections with a single line, multi-line content, and empty lines and empty section at the end of the file. The unit test validates, that header is skipped, 3 sections in between contain all content, including empty lines, and that empty section at the end of the file is not included in the result.

4.7.2 Name section tests

The logic that extracts command names and their description from the name section is tested on input data, that contains both dash and double dash as a command - description separator, two command names on the same line separated by a comma and one command name on a separate line. Input data

also includes a multi-line description. The unit test validates that all possible variations of the general format of the name section are parsed correctly.

4.7.3 Synopsis parser tests

Synopsis parsing logic is tested with two unit tests:

- Splitting synopsis content into syntax lines per command name. Input data contains lines that do not start with command name and lines that do. There are also single-line and multi-line syntax occurrences.
- Building a syntax tree out of syntax line. Input syntax line contains repeating, optional placeholder option, a short option, a long option with argument, and two command arguments. The unit test validates that option's argument was recognized as the option's argument, the correctness of returned options and arguments and its attributes.

4.7.4 Description parser tests

Description parser logic is tested with a single unit test, which input data contains:

- short-only option
- long-only option
- option with short and long names and an optional argument
- option with multiple short and multiple long names and a required argument
- option with a description that is starting on the same line as option definition

Test validate count of parsed options, attributes of each option, such as type (short, long, short and long), names, description, argument presence, argument name, argument attributes.

4.7.5 Merging synopsis and description output tests

The logic of merging output from synopsis and description sections parsing is tested by a single test case. That test case covers common options placeholder substitution, primary syntax, and syntax-specific options.

The test validates that syntaxes contain only syntax-specific options, that belong to current syntax, except the first syntax, which also contains all common options. Apart from it, there is also validation that all occurrences of common option placeholder were removed and that the first syntax contains

common options with correct attributes taken from the placeholder. At last, the test validates, that syntaxes were named after its first syntax-specific options if any, otherwise syntaxN, where N is an order number.

4.7.6 Type detection tests

Type detection tests are covered by a single test case and validate that all naming conventions are applied: command argument type is the same as its name, option's argument type is its name following option's name and that for short name only option `OPTION_` or `OPTION_UPPER` prefix is used.

4.7.7 User definitions tests

User definitions tests are covered by two use cases: parsing and merging.

Parsing test case validates correct parsing of two possible statements and their variations: type replacement rules and enumerations. As input we prepared a string containing: a line with a comment; type replacement rules containing not only alphabet characters but also dashes, brackets, dollar signs, underscores; single line enumeration definition and multi-line enumeration definition containing value descriptions. Test code validates total count and each of the type replacements rules and enumerations and all of its' attributes.

Merging test case validates merging of two definitions that have enumeration with the same name, type replacement rule with the same name, and also each of them has its unique content. It validates, that all unique content of both definitions present, and in case of definitions with the same name those from the second file are present.

4.7.8 Memory usage tests

Makefile contains two additional targets to check memory leaks under `valgrind` environment:

- `memory_test` to run all unit tests
- `memory_test_tar` to run the tool with `tar` man page as an input

Running both test targets proved, that no memory errors were detected.

4.7.9 Manual testing

In this section, we will look into the process of defining valid CLD definitions for a few commands. We'll start from scratch by running the tool against a formatted manual page and let it generate a user definitions file template. Then we will edit the template, check and fill all types. After that, we will generate the final CLD and validate it with help of `cdu` utility described in section 4.1.1.

For all tests below we have used man pages from OpenSUSE Leap version 15.1.

Let's start with `ls` command. It's an example of a more or less simple command, which consists of one syntax only. The formatted plain text content of ls man page can be found in appendix C.2.1.

```
$ make
$ man -w ls
/usr/share/man/man1/ls.1.gz
$ gzip -cd $(man -w ls) > test-data/ls.1
$ groff -P -c -man -T ascii test-data/ls.1 > test-data/ls.1.txt
$ ./man2cld --generate-definitions-template test-data/ls.def \
  -o test-data/ls.cld test-data/ls.1.txt
```

At this moment we should see that `test-data` folder has four files in it: raw man page source in groff, formatted man page, definitions template, and first output. Making ourselves familiar with the output and looking into options description, we should be able to notice, that there are few keyword types that the tool wasn't able to parse. For example, argument `WHEN` of `color` option. According to the description it accepts one of following values: `always`, `auto` or `more`.

In the user definitions template then we have prepared a type replacement rule for it: `WHEN = $STRING`. So our goal right now is to improve it by specifying, that `WHEN` is a keyword by changing the type replacement rule to keyword type definition rule: `WHEN = (always auto more)`.

One more issue we can notice in CLD is that tool has wrongly understood, that option `help` has an argument `display`. This happened due to relaxed criteria in option parsing logic to support more formats of man pages. This issue is solved easily by appending a special type replacement rule: `display = NOT_AN_ARGUMENT`.

By iterating all of the types we can end up with a definition file similar to the one listed in appendix C.2.2.

Now we are going to run the tool again to apply the latest version of definitions. Then, validating the result via `cdu`.

```
$ ./man2cld --definitions test-data/ls.def -o test-data/ls.cld \
  test-data/ls.1.txt
$ /home/cdu/cdu3 test-data/ls.cld
Openning file test-data/ls.cld
$ echo $?
0
```

No errors were printed, also to confirm a successful finish we check the exit code, which is 0 - success. We have now valid CLD definition of `ls` command, which can be found in appendix C.2.3.

The next command that we will generate a definition for is `mv`. This in difference to the `ls` have multiple syntaxes. We will check that the tool's multi-syntax output is valid. Formatted plain text contents of man page can be found in appendix C.3.1.

```
$ man -w mv
/usr/share/man/man1/mv.1.gz
$ gzip -cd $(man -w mv) > test-data/mv.1
$ groff -P -c -man -T ascii test-data/mv.1 > test-data/mv.1.txt
$ ./man2cld --generate-definitions-template test-data/mv.def \
  -o test-data/mv.cld test-data/mv.1.txt
```

Now, we should be able to improve argument types by manual edit of `test-data/mv.def` file. We can end up with content listed in appendix C.3.2.

Let's run the tool again and validate it.

```
$ ./man2cld --definitions test-data/mv.def -o test-data/mv.cld \
  test-data/mv.1.txt
$ /home/cdu/cdu3 test-data/mv.cld
Openning file test-data/mv.cld
$ echo $?
0
```

This confirms that the multi-syntax definition is valid. Full output in CLD can be found in appendix C.3.3.

Next command that we are going to test tool on is `tar`. `tar` contains more syntax lines as listed in appending C.4. All of them are contained in three sections, separated by usage type:

- Traditional usage
- UNIX-style usage
- GNU-style usage

It became clear in practice, that so many similar syntaxes confuse the tool. In order to cover all possible syntaxes, we are going to remove traditional and UNIX-style usage sections, leaving only GNU-style section. This section covers all possible modes of `tar` command. As for the short option style, there is nothing to worry about, because each long option will have a corresponding short name parsed from the description section, so it also will be covered.

Disallowing options from different syntax will require manual definition with help of *append CLD* feature, that user definitions file format supports. Moreover, `tar` allows specifying options without a leading dash, in CLD this is achieved by manually adding `OPTBROCADE` statement to the verb definition. Repeating the same steps as with previous commands, we get to the point where we can check the final output.

```
$ ./man2cld --definitions test-data/tar.def \
  -o test-data/tar.cld \
  test-data/tar-edited.1.txt
$ /home/cdu/cdu3 test-data/tar.cld
Openning file test-data/tar.cld
Line 117:       OPTPARAMETER
%cdu-E-invitem, Invalid item "OPTPARAMETER" encountered.
Some text will be skipped
generate_table_blocks: 0
%cdu-F-intinvnode, Internal error: invalid node encountered
```

Here, `cdu` did not recognized the OPTPARAMETER statement. After discussion, we have concluded that this problem occurred due to not complete implementation of the defined extension in the current version of `cdu`. We have manually verified, that OPTPARAMETER in this case is placed correctly. To verify the rest of the definition we removed this statement and run `cdu` again.

```
$ /home/cdu/cdu3 test-data/tar-no-optparameter.cld
Openning file test-data/tar-no-optparameter.cld
$ echo $?
0
```

There are several more commands for which we have generated definitions and successfully validated them using the established process:

- `find`
- `test`
- `ps`

All of them have a unique format of the description section, where options are described. Due to the large size, they are not included in the appendix but can be found in attachments to this work.

4.8 Further required CLD extensions

During a deeper analysis of some commands (`tar`, `test` and `ps`) we found out next extension of the CDL language is required to describe possible options. New statements are the following

- `OPTPARAMETER` to specify option without leading dash, for example, `A` in `tar`
- `OPTBROCADE` to specify long option name with single leading dash, for example, `-follow` in `find`

- `UNIXNEGATE` to negate option by using `+` instead of `-`
- `NOEQSIGN` to forbid using equal sign as a separator between option and it's argument
- `ALTSYNTAX syntax-name` to be able to specify alternative syntax, which does not have syntax-specific options

Furthermore, from the analysis it became clear that we need to extend supported types with the following:

- `$SIZE` - size with units (KB, MB, etc.)
- `$TEST_EXPRESSION` - Unix `test` command expression
- `$UNIXPROT` - Unix protection specifier
- `$UID` - user name or UID
- `$GID` - group name or GID
- `$MAJORMINOR` - version specification, `number.number`
- `$FREE` - any value delimited by a space or a string in quotes

The next required extension originates from a specific argument type of `ls` command `DATE-OR-FILE`. CDL does not allow to combine values of different types. Here we introduce a new syntax for defining such data types, which is similar to keyword type definition except that `TYPE` is used instead of `KEYWORD`:

```
DEFINE TYPE DATE-OR-FILE
    TYPE $DATE
    TYPE $FILE
```

Conclusion

The theoretical part of this work was focused on studying Command Line Description language and Linux manual pages. We described the syntax of CLD, terminology, and the possibilities it provides. We also defined an extension to be able to describe Unix commands. Then, we described the format and structure of Linux manual pages, possible alternative formats, and have selected the most suitable one.

In the analytical part of this work, we have looked more closely at the manual page in the selected format. We have chosen parts of input containing valuable information, how to find and extract it. Then, we have verified how extracted information would be described in CLD on an example of mv command. Afterward, we have designed the solution by stating the current manual process, enhancement we were willing to do, and requirements. We have specified possible use-cases and finally based on all previously collected information defined how the solution would work.

In the practical part of this work, we have focused on the implementation of the designed solution. We have defined a way to proceed through limitations of the automatic process and its possible errors by providing additional input in form of a user definitions file. Furthermore, we have tested the solution with unit tests, memory tests, and manual tests. Manual tests included writing a complete definition for commands, such as ls, mv, tar, find, test, and ps. Those definitions were afterward successfully tested on correctness via cdu utility - one of the components of dclsh shell.

During implementation, we have discovered several missing features in CLD to be able to describe Unix commands. We have defined and specified necessary extensions.

The implemented solution can be used to automate the process of writing CLD definitions for Unix commands, therefore allowing dclsh shell to use these definitions.

Bibliography

[1] Kristaps Dzonsons. *History of UNIX Manpages*. URL: `http://manpages.bsd.lv/history.html`. [cit. 2019-12-15].

[2] *HP OpenVMS Command Definition, Librarian, and Message Utilities Manual*. Version 7.3. Hewlett-Packard Company. Jan. 2005.

[3] *man-pages - conventions for writing Linux man pages*. Version 4.04. The Linux Foundation. Dec. 2015. [cit. 2019-12-15].

[4] Eric Raymond. *The Jargon File*. URL: `http://www.catb.org/jargon/html/S/shell.html`. [cit. 2019-12-15].

[5] *The GNU Troff Manual*. Free Software Foundation, Inc. Nov. 2008. URL: `https://www.gnu.org/software/groff/manual/groff.htm`. [cit. 2019-12-15].

Appendix A

Acronyms

DOS Disk operating system

DCL DIGITAL Command Language

CDL Command definition language

CLD Command line description

ASCII American standard code for information interchange

FC Fibre channel

CDU Command definition utility

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `man2cld` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format

Appendix C

Man pages, user definitions and outputs

This chapter includes contents of plain text man pages, user definitions, and CLD outputs generated by the tool for different commands.

C.1 Manual definition of mv command

```
DEFINE VERB mv
    IMAGE "/bin/mv"
    IMAGETYPE unix
    HELP "mv - move (rename) files"

    QUALIFIER OPTION_b
        UNIXOPT "b"
        HELP "like --backup but does not accept an argument"
    QUALIFIER force
        UNIXOPT "f"
        HELP "do not prompt before overwriting"
    QUALIFIER suffix
        UNIXOPT "S"
        VALUE (REQUIRED, TYPE=$STRING)
        HELP "override the usual backup suffix"
  QUALIFIER no-target-directory
        UNIXOPT "T"
        HELP "treat DEST as a normal file"
    QUALIFIER update
        UNIXOPT "u"
        HELP "move only when the SOURCE file is newer
than the destination file or when the
destination file is missing"
    PARAMETER P1, LABEL=SOURCE
        VALUE (REQUIRED, TYPE=$INFILE)
    PARAMETER P2, LABEL=DEST
        VALUE (REQUIRED, TYPE=$FILE)
```

```
    QUALIFIER target-directory, SYNTAX=target-directory
        UNIXOPT "t"
        VALUE (REQUIRED, TYPE=$DIRECTORY)
        HELP "move all SOURCE arguments into DIRECTORY"

DEFINE SYNTAX syntax2
    PARAMETER P1, LABEL=SOURCE
        VALUE (UNIXLIST, REQUIRED, TYPE=$INFILE)
    PARAMETER P2, LABEL=DIRECTORY
        VALUE (REQUIRED, TYPE=$DIRECTORY)

DEFINE SYNTAX target-directory
    QUALIFIER target-directory
        UNIXOPT "t"
        VALUE (REQUIRED, TYPE=$DIRECTORY)
        HELP "move all SOURCE arguments into DIRECTORY"
    PARAMETER P1, LABEL=SOURCE
        VALUE (UNIXLIST, REQUIRED, TYPE=$INFILE)
```

C.2 ls command

C.2.1 Man page

```
LS(1)        User Commands         LS(1)



NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory
     by default).
       Sort entries alphabetically if none of -cftuvSUX nor --
    sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for
    short
       options
       too.

       -a, --all
        do not ignore entries starting with .

       -A, --almost-all
        do not list implied . and ..

       --author
```

```
     with -l, print the author of each file

    -b, --escape
     print C-style escapes for nongraphic characters

    --block-size=SIZE
     with  -l,  scale  sizes  by  SIZE  when  printing  them;
   e.g.,
     '--block-size=M'; see SIZE format below

    -B, --ignore-backups
     do not list implied entries ending with ~

    -c     with -lt: sort by, and show, ctime (time of last
modification of
     file status information); with -l: show ctime and sort by
  name;
     otherwise: sort by ctime, newest first

    -C     list entries by columns

    --color[=WHEN]
     colorize  the output; WHEN can be 'always' (default if
omitted),
     'auto', or 'never'; more info below

    -d, --directory
     list directories themselves, not their contents

    -D, --dired
     generate output designed for Emacs' dired mode

    -f     do not sort, enable -aU, disable -ls --color

    -F, --classify
     append indicator (one of */=>@|) to entries

    --file-type
     likewise, except do not append '*'

    --format=WORD
     across -x, commas -m, horizontal -x, long -l, single-
column  -1,
     verbose -l, vertical -C

    --full-time
     like -l --time-style=full-iso

    -g     like -l, but do not list owner

    --group-directories-first
     group directories before files;
```

```
     can   be   augmented   with   a   --sort   option,   but   any
use  of
     --sort=none (-U) disables grouping

    -G, --no-group
     in a long listing, don't print group names

    -h, --human-readable
     with -l and -s, print sizes like 1K 234M 2G etc.

    --si   likewise, but use powers of 1000 not 1024

    -H, --dereference-command-line
     follow symbolic links listed on the command line

    --dereference-command-line-symlink-to-dir
     follow each command line symbolic link

     that points to a directory

    --hide=PATTERN
     do not list implied entries matching shell  PATTERN  (
overridden
     by -a or -A)

    --hyperlink[=WHEN]
     hyperlink file names; WHEN can be 'always' (default if
omitted),
     'auto', or 'never'

    --indicator-style=WORD
     append indicator with style WORD to entry names: none (
default),
     slash (-p), file-type (--file-type), classify (-F)

    -i, --inode
     print the index number of each file

    -I, --ignore=PATTERN
     do not list implied entries matching shell PATTERN

    -k, --kibibytes
     default  to  1024-byte  blocks for disk usage; used only
with -s
     and per directory totals

    -l     use a long listing format

    -L, --dereference
     when showing file information for a symbolic link, show
informa-
     tion  for  the file the link references rather than for
the link
     itself
```

```
    -m     fill width with a comma separated list of entries

    -n, --numeric-uid-gid
     like -l, but list numeric user and group IDs

    -N, --literal
     print entry names without quoting

    -o     like -l, but do not list group information

    -p, --indicator-style=slash
     append / indicator to directories

    -q, --hide-control-chars
     print ? instead of nongraphic characters

    --show-control-chars
     show nongraphic characters as-is (the default, unless
program is
     'ls' and output is a terminal)

    -Q, --quote-name
     enclose entry names in double quotes

    --quoting-style=WORD
     use  quoting style WORD for entry names: literal, locale,
 shell,
     shell-always,  shell-escape,  shell-escape-always,   c,
escape
     (overrides QUOTING_STYLE environment variable)

    -r, --reverse
     reverse order while sorting

    -R, --recursive
     list subdirectories recursively

    -s, --size
     print the allocated size of each file, in blocks

    -S     sort by file size, largest first

    --sort=WORD
     sort  by  WORD instead of name: none (-U), size (-S),
time (-t),
     version (-v), extension (-X)

    --time=WORD
     with -l, show time as WORD instead of default
modification time:
     atime  or  access  or  use  (-u); ctime or status (-c);
also use
     specified time as sort key if --sort=time (newest first)
```

```
   --time-style=TIME_STYLE
    time/date format with -l; see TIME_STYLE below

   -t     sort by modification time, newest first

   -T, --tabsize=COLS
    assume tab stops at each COLS instead of 8

   -u     with -lt: sort by, and show, access time; with -l:
show  access
    time  and  sort  by name; otherwise: sort by access time,
newest
    first

   -U     do not sort; list entries in directory order

   -v     natural sort of (version) numbers within text

   -w, --width=COLS
    set output width to COLS.  0 means no limit

   -x     list entries by lines instead of by columns

   -X     sort alphabetically by entry extension

   -Z, --context
    print any security context of each file

   -1     list one file per line.  Avoid '\n' with -q or -b

   --help display this help and exit

   --version
    output version information and exit

   The SIZE argument is an integer and  optional  unit  (
example:  10K  is
   10*1024).   Units  are  K,M,G,T,P,E,Z,Y  (powers  of 1024)
 or KB,MB,...
   (powers of 1000).

   The TIME_STYLE argument can be  full-iso,  long-iso,  iso,
  locale,  or
   +FORMAT.   FORMAT  is  interpreted  like in date(1).  If
FORMAT is FOR-
   MAT1<newline>FORMAT2, then FORMAT1 applies to non-recent
files and FOR-
   MAT2  to  recent files.  TIME_STYLE prefixed with 'posix-'
 takes effect
   only outside the POSIX locale.  Also the TIME_STYLE
environment  vari-
   able sets the default style to use.
```

```
       Using  color  to distinguish file types is disabled both
    by default and
       with --color=never.  With --color=auto, ls emits color
    codes only  when
       standard  output is connected to a terminal.  The
    LS_COLORS environment
       variable can change the settings.  Use the dircolors
    command to set it.

   Exit status:
       0      if OK,

       1      if minor problems (e.g., cannot access subdirectory
    ),

       2      if serious trouble (e.g., cannot access command-
    line argument).

AUTHOR
       Written by Richard M. Stallman and David MacKenzie.

REPORTING BUGS
       GNU coreutils online help: <https://www.gnu.org/software/
    coreutils/>
       Report ls translation bugs to <https://translationproject.
    org/team/>

COPYRIGHT
       Copyright (C) 2017 Free Software Foundation, Inc.  License
     GPLv3+:  GNU
       GPL version 3 or later <https://gnu.org/licenses/gpl.html
    >.
       This  is  free  software:  you  are free to change and
    redistribute it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       Full documentation at: <https://www.gnu.org/software/
    coreutils/ls>
       or available locally via: info '(coreutils) ls invocation'



GNU coreutils 8.29     February 2018         LS(1)
```

C.2.2 User definitions

```
FILE = $INFILE
SIZE = $SIZE
WHEN = (always auto never)
color_WHEN = WHEN
format_WORD = (across commas horizontal long single-column
verbose vertical)
```

```
none_OR_WORD = $STRING
hide_PATTERN = $QUOTED_STRING
hyperlink_WHEN = WHEN
WORD_OR_slash = (none slash file-type classify)
ignore_PATTERN = $QUOTED_STRING
quoting-style_WORD = (literal locale shell shell-always
shell-escape shell-escape-always c escape)
time_WORD = (atime access ctime status)
TIME = (full-iso long-iso iso locale)
TIME_STYLE = (TIME $STRING)
tabsize_COLS = $NUMBER
width_COLS = $NUMBER
display = NOT_AN_ARGUMENT
```

C.2.3 Output CLD

```
DEFINE TYPE WHEN
  KEYWORD always, NONNEGATABLE
  KEYWORD auto, NONNEGATABLE
  KEYWORD never, NONNEGATABLE

DEFINE TYPE format_WORD
  KEYWORD across, NONNEGATABLE
  KEYWORD commas, NONNEGATABLE
  KEYWORD horizontal, NONNEGATABLE
  KEYWORD long, NONNEGATABLE
  KEYWORD single-column, NONNEGATABLE
  KEYWORD verbose, NONNEGATABLE
  KEYWORD vertical, NONNEGATABLE

DEFINE TYPE WORD_OR_slash
  KEYWORD none, NONNEGATABLE
  KEYWORD slash, NONNEGATABLE
  KEYWORD file-type, NONNEGATABLE
  KEYWORD classify, NONNEGATABLE

DEFINE TYPE quoting-style_WORD
  KEYWORD literal, NONNEGATABLE
  KEYWORD locale, NONNEGATABLE
  KEYWORD shell, NONNEGATABLE
  KEYWORD shell-always, NONNEGATABLE
  KEYWORD shell-escape, NONNEGATABLE
  KEYWORD shell-escape-always, NONNEGATABLE
  KEYWORD c, NONNEGATABLE
  KEYWORD escape, NONNEGATABLE

DEFINE TYPE time_WORD
  KEYWORD atime, NONNEGATABLE
  KEYWORD access, NONNEGATABLE
  KEYWORD ctime, NONNEGATABLE
  KEYWORD status, NONNEGATABLE

DEFINE TYPE TIME
```

```
  KEYWORD full-iso, NONNEGATABLE
  KEYWORD long-iso, NONNEGATABLE
  KEYWORD iso, NONNEGATABLE
  KEYWORD locale, NONNEGATABLE

DEFINE TYPE TIME_STYLE
  TYPE TIME
  TYPE $STRING

DEFINE VERB ls
  IMAGE "/bin/ls"
  IMAGETYPE unix
  HELP "list directory contents"

  QUALIFIER all
    UNIXOPT "a"
    HELP "do not ignore entries starting with ."

  QUALIFIER almost-all
    UNIXOPT "A"
    HELP "do not list implied . and .."

  QUALIFIER author
    HELP "with -l, print the author of each file"

  QUALIFIER escape
    UNIXOPT "b"
    HELP "print C-style escapes for nongraphic characters"

  QUALIFIER block-size
    VALUE (REQUIRED, TYPE=$SIZE)
    HELP "with  -l,  scale  sizes  by  SIZE  when  printing  them
    ;   e.g.,
'--block-size=M'; see SIZE format below"

  QUALIFIER ignore-backups
    UNIXOPT "B"
    HELP "do not list implied entries ending with ~"

  QUALIFIER OPTION_c
    UNIXOPT "c"
    HELP "with -lt: sort by, and show, ctime (time of last
    modification of
file status information); with -l: show ctime and sort by  name;
otherwise: sort by ctime, newest first"

  QUALIFIER OPTION_UPPERC
    UNIXOPT "C"
    HELP "list entries by columns"

  QUALIFIER color
    VALUE (TYPE=WHEN)
    HELP "colorize  the output; WHEN can be 'always' (default if
    omitted),
```

```
'auto', or 'never'; more info below"

  QUALIFIER directory
    UNIXOPT "d"
    HELP "list directories themselves, not their contents"

  QUALIFIER dired
    UNIXOPT "D"
    HELP "generate output designed for Emacs' dired mode"

  QUALIFIER OPTION_f
    UNIXOPT "f"
    HELP "do not sort, enable -aU, disable -ls --color"

  QUALIFIER classify
    UNIXOPT "F"
    HELP "append indicator (one of */=>@|) to entries"

  QUALIFIER file-type
    HELP "likewise, except do not append '*'"

  QUALIFIER format
    VALUE (REQUIRED, TYPE=format_WORD)
    HELP "across -x, commas -m, horizontal -x, long -l, single-
   column  -1,
verbose -l, vertical -C"

  QUALIFIER full-time
    HELP "like -l --time-style=full-iso"

  QUALIFIER OPTION_g
    UNIXOPT "g"
    HELP "like -l, but do not list owner"

  QUALIFIER group-directories-first
    HELP "group directories before files;"

  QUALIFIER sort
    VALUE (REQUIRED, TYPE=$STRING)
    HELP "(-U) disables grouping
sort  by  WORD instead of name: none (-U), size (-S), time (-t),
version (-v), extension (-X)"

  QUALIFIER no-group
    UNIXOPT "G"
    HELP "in a long listing, don't print group names"

  QUALIFIER human-readable
    UNIXOPT "h"
    HELP "with -l and -s, print sizes like 1K 234M 2G etc."

  QUALIFIER si
    HELP "likewise, but use powers of 1000 not 1024"
```

```
  QUALIFIER dereference-command-line
    UNIXOPT "H"
    HELP "follow symbolic links listed on the command line"

  QUALIFIER dereference-command-line-symlink-to-dir
    HELP "follow each command line symbolic link"

  QUALIFIER hide
    VALUE (REQUIRED, TYPE=$QUOTED_STRING)
    HELP "do not list implied entries matching shell  PATTERN  (
    overridden
by -a or -A)"

  QUALIFIER hyperlink
    VALUE (TYPE=WHEN)
    HELP "hyperlink file names; WHEN can be 'always' (default if
    omitted),
'auto', or 'never'"

  QUALIFIER indicator-style
    VALUE (REQUIRED, TYPE=WORD_OR_slash)
    HELP "append indicator with style WORD to entry names: none (
    default),
slash (-p), file-type (--file-type), classify (-F)
append / indicator to directories"

  QUALIFIER inode
    UNIXOPT "i"
    HELP "print the index number of each file"

  QUALIFIER ignore
    UNIXOPT "I"
    VALUE (REQUIRED, TYPE=$QUOTED_STRING)
    HELP "do not list implied entries matching shell PATTERN"

  QUALIFIER kibibytes
    UNIXOPT "k"
    HELP "default  to  1024-byte  blocks for disk usage; used
    only with -s
and per directory totals"

  QUALIFIER OPTION_l
    UNIXOPT "l"
    HELP "use a long listing format"

  QUALIFIER dereference
    UNIXOPT "L"
    HELP "when showing file information for a symbolic link, show
     informa-
tion  for  the file the link references rather than for the link
itself"

  QUALIFIER OPTION_m
    UNIXOPT "m"
```

```
    HELP "fill width with a comma separated list of entries"

  QUALIFIER numeric-uid-gid
    UNIXOPT "n"
    HELP "like -l, but list numeric user and group IDs"

  QUALIFIER literal
    UNIXOPT "N"
    HELP "print entry names without quoting"

  QUALIFIER OPTION_o
    UNIXOPT "o"
    HELP "like -l, but do not list group information"

  QUALIFIER hide-control-chars
    UNIXOPT "q"
    HELP "print ? instead of nongraphic characters"

  QUALIFIER show-control-chars
    HELP "show nongraphic characters as-is (the default, unless
    program is
'ls' and output is a terminal)"

  QUALIFIER quote-name
    UNIXOPT "Q"
    HELP "enclose entry names in double quotes"

  QUALIFIER quoting-style
    VALUE (REQUIRED, TYPE=quoting-style_WORD)
    HELP "use  quoting style WORD for entry names: literal,
    locale, shell,
shell-always, shell-escape, shell-escape-always,  c,  escape
(overrides QUOTING_STYLE environment variable)"

  QUALIFIER reverse
    UNIXOPT "r"
    HELP "reverse order while sorting"

  QUALIFIER recursive
    UNIXOPT "R"
    HELP "list subdirectories recursively"

  QUALIFIER size
    UNIXOPT "s"
    HELP "print the allocated size of each file, in blocks"

  QUALIFIER OPTION_UPPERS
    UNIXOPT "S"
    HELP "sort by file size, largest first"

  QUALIFIER time
    VALUE (REQUIRED, TYPE=time_WORD)
    HELP "with -l, show time as WORD instead of default
    modification time:
```

```
atime  or  access  or  use  (-u); ctime or status (-c); also use
specified time as sort key if --sort=time (newest first)"

  QUALIFIER time-style
    VALUE (REQUIRED, TYPE=TIME_STYLE)
    HELP "time/date format with -l; see TIME_STYLE below"

  QUALIFIER OPTION_t
    UNIXOPT "t"
    HELP "sort by modification time, newest first"

  QUALIFIER tabsize
    UNIXOPT "T"
    VALUE (REQUIRED, TYPE=$NUMBER)
    HELP "assume tab stops at each COLS instead of 8"

  QUALIFIER OPTION_u
    UNIXOPT "u"
    HELP "with -lt: sort by, and show, access time; with -l:
     show  access
time  and  sort  by name; otherwise: sort by access time, newest
first"

  QUALIFIER OPTION_UPPERU
    UNIXOPT "U"
    HELP "do not sort; list entries in directory order"

  QUALIFIER OPTION_v
    UNIXOPT "v"
    HELP "natural sort of (version) numbers within text"

  QUALIFIER width
    UNIXOPT "w"
    VALUE (REQUIRED, TYPE=$NUMBER)
    HELP "set output width to COLS.  0 means no limit"

  QUALIFIER OPTION_x
    UNIXOPT "x"
    HELP "list entries by lines instead of by columns"

  QUALIFIER OPTION_UPPERX
    UNIXOPT "X"
    HELP "sort alphabetically by entry extension"

  QUALIFIER context
    UNIXOPT "Z"
    HELP "print any security context of each file"

  QUALIFIER OPTION_1
    UNIXOPT "1"
    HELP "list one file per line.  Avoid '\n' with -q or -b"

  QUALIFIER help
    HELP "display this help and exit"
```

```
  QUALIFIER version
    HELP "output version information and exit"

  PARAMETER P1, LABEL=FILE
    VALUE (UNIXLIST, TYPE=$INFILE)
```

C.3 mv command

C.3.1 Man page

```
MV(1)        User Commands         MV(1)



NAME
       mv - move (rename) files

SYNOPSIS
       mv [OPTION]... [-T] SOURCE DEST
       mv [OPTION]... SOURCE... DIRECTORY
       mv [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION
       Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

       Mandatory  arguments  to  long  options are mandatory for
   short options
       too.

       --backup[=CONTROL]
        make a backup of each existing destination file

       -b     like --backup but does not accept an argument

       -f, --force
        do not prompt before overwriting

       -i, --interactive
        prompt before overwrite

       -n, --no-clobber
        do not overwrite an existing file

       If you specify more than one of -i, -f, -n, only the
   final  one  takes
       effect.

       --strip-trailing-slashes
        remove any trailing slashes from each SOURCE argument

       -S, --suffix=SUFFIX
```

```
       override the usual backup suffix

      -t, --target-directory=DIRECTORY
       move all SOURCE arguments into DIRECTORY

      -T, --no-target-directory
       treat DEST as a normal file

      -u, --update
       move  only  when  the  SOURCE file is newer than the
   destination
       file or when the destination file is missing

      -v, --verbose
       explain what is being done

      -Z, --context
       set SELinux security context of destination file to
   default type

      --help display this help and exit

      --version
       output version information and exit

      The  backup  suffix  is  '~',  unless  set  with  --
   suffix  or  SIM-
      PLE_BACKUP_SUFFIX.  The version control method may be
   selected via the
      --backup option or through the  VERSION_CONTROL
   environment  variable.
      Here are the values:

      none, off
       never make backups (even if --backup is given)

      numbered, t
       make numbered backups

      existing, nil
       numbered if numbered backups exist, simple otherwise

      simple, never
       always make simple backups

AUTHOR
       Written by Mike Parker, David MacKenzie, and Jim Meyering.

REPORTING BUGS
       GNU coreutils online help: <https://www.gnu.org/software/
   coreutils/>
      Report mv translation bugs to <https://translationproject.
   org/team/>
```

```
COPYRIGHT
       Copyright  (C) 2017 Free Software Foundation, Inc.
    License GPLv3+: GNU
       GPL version 3 or later <https://gnu.org/licenses/gpl.html
    >.
       This is free software: you are free  to  change  and
    redistribute  it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       rename(2)

       Full documentation at: <https://www.gnu.org/software/
    coreutils/mv>
       or available locally via: info '(coreutils) mv invocation'


GNU coreutils 8.29     February 2018         MV(1)
```

C.3.2 User definitions

```
SOURCE = $FILE
DEST = $FILE
CONTROL = (none off numbered t existing nil simple never)
SUFFIX = $STRING
display = NOT_AN_ARGUMENT
DIRECTORY = $DIRECTORY
target-directory_DIRECTORY = $DIRECTORY
```

C.3.3 Output CLD

```
DEFINE TYPE CONTROL
  KEYWORD none, NONNEGATABLE
  KEYWORD off, NONNEGATABLE
  KEYWORD numbered, NONNEGATABLE
  KEYWORD t, NONNEGATABLE
  KEYWORD existing, NONNEGATABLE
  KEYWORD nil, NONNEGATABLE
  KEYWORD simple, NONNEGATABLE
  KEYWORD never, NONNEGATABLE

DEFINE VERB mv
  IMAGE "/bin/mv"
  IMAGETYPE unix
  HELP "move (rename) files"

  ALTSYNTAX=syntax2

  QUALIFIER backup
    VALUE (TYPE=CONTROL)
```

```
    HELP "make a backup of each existing destination file"

  QUALIFIER OPTION_b
    UNIXOPT "b"
    HELP "like --backup but does not accept an argument"

  QUALIFIER force
    UNIXOPT "f"
    HELP "do not prompt before overwriting"

  QUALIFIER interactive
    UNIXOPT "i"
    HELP "prompt before overwrite"

  QUALIFIER no-clobber
    UNIXOPT "n"
    HELP "do not overwrite an existing file"

  QUALIFIER strip-trailing-slashes
    HELP "remove any trailing slashes from each SOURCE argument"

  QUALIFIER suffix
    UNIXOPT "S"
    VALUE (REQUIRED, TYPE=$STRING)
    HELP "override the usual backup suffix"

  QUALIFIER no-target-directory
    UNIXOPT "T"
    HELP "treat DEST as a normal file"

  QUALIFIER update
    UNIXOPT "u"
    HELP "move  only  when  the  SOURCE file is newer than the
    destination
file or when the destination file is missing"

  QUALIFIER verbose
    UNIXOPT "v"
    HELP "explain what is being done"

  QUALIFIER context
    UNIXOPT "Z"
    HELP "set SELinux security context of destination file to
    default type"

  QUALIFIER help
    HELP "display this help and exit"

  QUALIFIER version
    HELP "output version information and exit"

  PARAMETER P1, LABEL=SOURCE
    VALUE (REQUIRED, TYPE=$FILE)
```

```
  PARAMETER P2, LABEL=DEST
    VALUE (REQUIRED, TYPE=$FILE)

  QUALIFIER target-directory, SYNTAX=target-directory
    UNIXOPT "t"
    VALUE (REQUIRED, TYPE=$DIRECTORY)
    HELP "move all SOURCE arguments into DIRECTORY"


DEFINE SYNTAX syntax2

  PARAMETER P1, LABEL=SOURCE
    VALUE (UNIXLIST, REQUIRED, TYPE=$FILE)

  PARAMETER P2, LABEL=DIRECTORY
    VALUE (REQUIRED, TYPE=$DIRECTORY)


DEFINE SYNTAX target-directory

  QUALIFIER target-directory
    UNIXOPT "t"
    VALUE (REQUIRED, TYPE=$DIRECTORY)
    HELP "move all SOURCE arguments into DIRECTORY"

  PARAMETER P1, LABEL=SOURCE
    VALUE (UNIXLIST, REQUIRED, TYPE=$FILE)
```

C.4 tar command synopsis section

```
SYNOPSIS
   Traditional usage
        tar {A|c|d|r|t|u|x}[GnSkUWOmpsMBiajJzZhPlRvwo] [ARG...]

   UNIX-style usage
        tar -A [OPTIONS] ARCHIVE ARCHIVE

        tar -c [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -d [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -t [-f ARCHIVE] [OPTIONS] [MEMBER...]

        tar -r [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -u [-f ARCHIVE] [OPTIONS] [FILE...]

        tar -x [-f ARCHIVE] [OPTIONS] [MEMBER...]

   GNU-style usage
        tar {--catenate|--concatenate} [OPTIONS] ARCHIVE ARCHIVE
```

```
    tar --create [--file ARCHIVE] [OPTIONS] [FILE...]

    tar {--diff|--compare} [--file ARCHIVE] [OPTIONS] [FILE
...]

    tar --delete [--file ARCHIVE] [OPTIONS] [MEMBER...]

    tar --append [-f ARCHIVE] [OPTIONS] [FILE...]

    tar --list [-f ARCHIVE] [OPTIONS] [MEMBER...]

    tar --test-label [--file ARCHIVE] [OPTIONS] [LABEL...]

    tar --update [--file ARCHIVE] [OPTIONS] [FILE...]

    tar --update [-f ARCHIVE] [OPTIONS] [FILE...]

    tar {--extract|--get} [-f ARCHIVE] [OPTIONS] [MEMBER...]
```