SCAPAR: A Scalable Text Parser for Data Sciences

VERSION 0.1

Hui Liu

2018

Contents

Chapter 1

Introduction

1.1 Overview

In data sciences industry, there are huge amounts of data to process. A text parser should be flexible enough to understand multiple data types and efficient enough to read large data volume. Here SCAPAR is developed to handle data files. Data formats are designed and proper reading routines are implemented. The library is designed for Linux, Unix and Mac systems. It is also possible to compile under Windows. The code is written by C, and it is serial.

1.2 License

The package uses GPL license. If you have any issue, please contact: hui.sc.liu@gmail.com

1.3 Citation

If you use the Scapar library, you may cite it like this,

```
@misc{scapar-library,
    author="Hui Liu",
    title="SCAPAR: A Scalable Text Parser for Data Sciences",
    year="2018",
    note={\url{https://github.com/huiscliu/scapar/}}
}
```

1.4 Website

The official website for Scapar is `https://github.com/huiscliu/scapar/`.

Chapter 2

Installation

2.1 Configuration

The configuration sets system parameters and data types. The simplest way is to run the following command:

```
./configure
```

2.2 Options

The script configure has many options, if user would like to check, run command:

```
./configure --help
```

Output will be like this,

```
'configure' configures this package to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print 'checking ...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for '--cache-file=config.cache'
  -n, --no-create         do not create output files
```

```
      --srcdir=DIR        find the sources in DIR [configure dir or ‘..’]

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local/scapar]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [PREFIX]

By default, ‘make install’ will install all the files in
‘/usr/local/scapar/bin’, ‘/usr/local/scapar/lib’ etc.  You can specify
an installation prefix other than ‘/usr/local/scapar’ using ‘--prefix’,
for instance ‘--prefix=$HOME’.

For better control, use the options below.

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --disable-assert        turn off assertions
  --enable-big-int        use long int for INT
  --disable-big-int       use int for INT (default),
  --with-int=type         integer type(long|long long)
  --enable-long-double    use long double for FLOAT
  --disable-long-double   use double for FLOAT (default)

Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  LIBS        libraries to pass to the linker, e.g. -l<library>
  CPPFLAGS    (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  CPP         C preprocessor
```

The default integer and floating point number are `int` and `double`. However, user can change the type of integer, such as `long long int`, by using options `--enable-big-int --with-int="long long"`, and the type of floating point number, such as `long double`, by using `--enable-long-double`. The integer number has three choices, `int`, `long int` and `long long int`, and the floating point number has two choices, `double` and `long double`.

2.3 Compilation

After configuration, Makefile and related scripts will be set correctly. A simple `make` command can compile the package,

```
    make
```

2.4 Installation

Run command:

```
    make install
```

The package will be installed to a directory. The default is /usr/local/scapar/. A different directory can be set by `--prefix=DIR`.

Chapter 3

Scapar Basics

3.1 Conventions

In a data file, backslash (\) means to continue. Comment line is ignored. In a vector or table, comment line also means to continue. The comment line starts with # by default, and it can be changed to another charactor. slash (/) has special meanings, which ends a table and a section.

3.2 Data Types

The Scapar library has many internal types, including basic types and complicated types.

3.2.1 Basic Data Types

`SCAPAR_FLT` is the floating point number type, which could be `double` or `long double` depending on the configuration options. `SCAPAR_INT` is the integer type, which could be `int`, `long int` or `long long int` depending on the configuration. `char *` is for string management.

`SCAPAR_ADDR` and `SCAPAR_BOOL` are internal data types for address and boolean. `SCAPAR_IDX` is for index, which is an integer type.

3.2.2 Info

When creating a vector or a table, some addtional information should be passed to notify Scapar the type of vector member and the type information of a table.

The member of a vector can be an integer, a floating point number, another vector, a table, a section and user-defined types. A column of a table can be a vector of integer, floating point number and string. The info type is defined as follows:

```
typedef struct SCAPAR_INFO_
{
    SCAPAR_TYPE type;
    SCAPAR_INT tsize;
    SCAPAR_INT msize;
    SCAPAR_TYPE *mtype;
} SCAPAR_INFO;
```

The `type` is used for vector, which is the type of a vector member, such as integer. `tsize` is the size of a member, such as 4 for `int` (most computer architectures) and 8 for `double` (most computer architectures). Sometimes the `tsize` is known for internal types and basic types, but sometimes `tsize` has to be set by user. For example, user can define a `struct` and Scapar does not know the size of the `struct`. `mtype` is the number of columns of a table and `mtype` is the type for each column. Legal types for mtype are `SCAPAR_T_FLT`, `SCAPAR_T_INT`, and `SCAPAR_T_STR`. For example, the following table has four columns.

```
Table:
1. 2. 4 a
1. 2. 5 b
1. 3. 4 e
1. 2. 7 por
2. 2. 4 ml
1. 2. 4 point
1. 2. 9 score
```

The first column and the second column are floating point numbers, the third column is integer and the forth column is string. In this case,

```
Table:
msize = 4, mtype = {SCAPAR_T_FLT, SCAPAR_T_FLT, SCAPAR_T_INT, SCAPAR_T_STR}
```

Scapar has some pre-defined info variable, such as `SCAPAR_INFO_INT` for integer, `SCAPAR_INFO_FLT` for floating point number, `SCAPAR_INFO_VEC_INT` for vector of integer, `SCAPAR_INFO_VEC_FLT` for vector of floating point number, and `SCAPAR_INFO_VEC_STR` for vector of string.

3.2.3 Vector

The vector (`SCAPAR_VEC`) is the same as arrary in C language, whose member could be integer, floating point number, string, another vector, table and `struct`. Its length is arbitrary, only limited by memory and operating system.

Examples:

```
The following vectors of integer are equivalent.

vec: 1 1 1 1 3 777 222 9 9 9 9 9

vec: 1 1 1 \
1 3 777 222 \
########
9 \
######
```

```
###### \
#############################
9 \
9 \
9 9

vector of vector of integer:
vc: 1 1 1 1 3
vc: 1 1
vc: 1 1 3
vc: 1 1 1 1 3
vc: 1 1
vc: 1 1 3
vc: 1 3
vc: 3
vc: 1 1 1 1 3 9 9 8 888 555 0 9 -3
vc: 1 1 1 1 3 3 3 3 3
```

3.2.4 Table

The following examples show how to define tables. `stb` and `stc` have four columns. When creating a table, its column size is required, such as four columns or eight columns. However, it can contain arbitrary rows. Each table is terminated by either an empty row or a slash (/) in the end of a legal row.

```
Table examples:
stb:
1. 2. 4 a
1. 2. 5 b
1. 3. 4 e
1. 2. 7 por
2. 2. 4 ml
1. 2. 4 point
1. 2. 9 score

stc:
1. 2. 4 a
1. 2. 5 b
1. 3. 4 e
1. 2. 7 por
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
2. 2. 4 ml
1. 2. 4 point
1. 2. 9 score /
```

3.2.5 Keyword Types

The Scapar supports the following data types: integer, floating-point number, string, table, vector of these basic types, user-defined types, and section.

```
typedef enum SCAPAR_TYPE_
{
    /* basic */
    SCAPAR_T_STR,    /* string */

    SCAPAR_T_INT,    /* int */

    SCAPAR_T_FLT,    /* float */

    SCAPAR_T_TABLE,  /* table */

    /* vector */
    SCAPAR_T_VEC_INT, /* int vector */
    SCAPAR_T_VEC_FLT, /* float vector */
    SCAPAR_T_VEC_STR, /* string vector */

    /* user defined */
    SCAPAR_T_USER0,  /* user defined type 0 */
    SCAPAR_T_USER1,  /* user defined type 1 */
    SCAPAR_T_USER2,  /* user defined type 2 */
    SCAPAR_T_USER3,  /* user defined type 3 */
    SCAPAR_T_USER4,  /* user defined type 4 */
    SCAPAR_T_USER5,  /* user defined type 5 */
    SCAPAR_T_USER6,  /* user defined type 6 */
    SCAPAR_T_USER7,  /* user defined type 7 */
    SCAPAR_T_USER8,  /* user defined type 8 */
    SCAPAR_T_USER9,  /* user defined type 9 */
    SCAPAR_T_USER10, /* user defined type 10 */
    SCAPAR_T_USER11, /* user defined type 11 */
    SCAPAR_T_USER12, /* user defined type 12 */
    SCAPAR_T_USER13, /* user defined type 13 */
    SCAPAR_T_USER14, /* user defined type 14 */
    SCAPAR_T_USER15, /* user defined type 15 */
    SCAPAR_T_USER16,
    SCAPAR_T_USER17,
    SCAPAR_T_USER18,
    SCAPAR_T_USER19,
    SCAPAR_T_USER20,
    SCAPAR_T_USER21,
    SCAPAR_T_USER22,
    SCAPAR_T_USER23,
    SCAPAR_T_USER24,
    SCAPAR_T_USER25,
    SCAPAR_T_USER26,
    SCAPAR_T_USER27,
    SCAPAR_T_USER28,
    SCAPAR_T_USER29,
    SCAPAR_T_USER30,
    SCAPAR_T_USER31,

    SCAPAR_T_VEC,   /* vector, general */
    SCAPAR_T_SEC,   /* section, may contain all types above */

} SCAPAR_TYPE;
```

Here are explanations:

- `SCAPAR_T_STR` defines a string, such as

  ```
  str: hello
  ```

- `SCAPAR_T_INT` defines an integer, such as

  ```
  id: 2037889
  ```

- `SCAPAR_T_FLT` defines a floating point number, such as

  ```
  price: 3.7
  ```

- `SCAPAR_T_TABLE` defines a table, such as

  ```
  tbl_1:
  0.1 2 3
  0.2 3 4.7
  0.3 8 9
  0.4 1 0.3

  tbl_2:
  0.1 2 3
  0.2 3 4.7
  0.3 8 9
  1.4 1 0.3
  1.5 1 0.3
  0.8 1 0.3 /
  ```

- `SCAPAR_T_VEC_STR` defines a vector of strings, such as

  ```
  str: hello jack and jill
  ```

- `SCAPAR_T_VEC_INT` defines a vector of integers, such as

  ```
  sn: 203788 99 222222 134447
  ```

- `SCAPAR_T_VEC_FLT` defines a vector of floating point numbers, such as

  ```
  price: 3.7 2. 3.1e-4 0.7778 1e-9
  ```

- SCAPAR_T_USERx defines a user-defined type. When this type is defined, certain functions must registered, such as conversion function, cleanup function to release allocated memory and printing function. Their prototypes are:

```
typedef void (*SCAPAR_CONV)(void *, FILE *, char *, SCAPAR_ITEM *);
typedef void (*SCAPAR_DESTROY)(void *, SCAPAR_ITEM *);
typedef void (*SCAPAR_PRINT)(void *, SCAPAR_ITEM *);

typedef struct SCAPAR_UT_INFO_
{
    SCAPAR_CONV conv;
    SCAPAR_DESTROY destroy;
    SCAPAR_PRINT print;

} SCAPAR_UT_INFO;
```

- SCAPAR_T_SEC defines a section, which is used to parse struct in C/C++. It can contain all defined data types, such as SCAPAR_T_INT, SCAPAR_T_VEC_INT, SCAPAR_T_VEC, and SCAPAR_T_SEC. A section ends with slash (/), such as

```
# key sec_example:
# vector of floating point number
price: 3.7 2. 3.1e-4 0.7778 1e-9

# string
name: Jack

# integer
birth_year: 1999
# vector of int
year: 1010 2001 2018 2111
# ending mark
/
```

- SCAPAR_T_VEC defines a vector, which can contain all defined data types, such as SCAPAR_T_INT, SCAPAR_T_VEC_INT, SCAPAR_T_TABLE, and SCAPAR_T_SEC, such as

```
# vector of vector of integer
vi: 1 2 3 4
vi: 1 2 3 4 5
vi: 8 2 3 4 5
vi: 2221 2 3 4 5 1 1 1 1 1 3.3
```

3.3 Work Flow

The following code sample shows how to use the Scapar library.

```
{
    char *in = "scalar.dat";
    SCAPAR scapar;
    SCAPAR_FLT km = -1.;

    /* 1. init */
    scapar_init(&scapar, in);

    /* settings, optional */
    scapar_set_dup_check(&scapar, 1);

    /* 2. register scalar key words */
    scapar_register_float(&scapar, "km", &km);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

Four steps must be taken:

1. The first step is to call `scapar_init`, which initializes the parser and accepts a file name. After this step, some setting functions can be invoked to control the behavoir of the library.
2. The second step is to register keyword. User can register as many as needed, such as 2000 keywords. All keywords must be registered here.
3. The third step is to call `scapar_parse`, which will read the file and parse its content according to those registered keywords.
4. The last step is to call `scapar_finalize`, which finalizes the parser and will clean up internal memories and status.

Chapter 4

Registration

A keyword must be registered to be parsed. Scapar provides many subroutines to manage different data types.

4.1 Integer

scapar_register_int registers an integer, key is the name of the keyword, and addr is the address where the integer is stored.

```
void scapar_register_int(SCAPAR *scapar, const char *key, SCAPAR_INT *addr);
```

Sample code:

```
{
    char *in = "keyword.dat";
    SCAPAR scapar;
    SCAPAR_INT verb = -1;

    /* init */
    scapar_init(&scapar, in);

    /* register */
    scapar_register_int(&scapar, "verb", &verb);

    /* parse */
    scapar_parse(&scapar);

    /* clean up */
    scapar_finalize(&scapar);
}
```

The keyword file contains the value, which is optional,

```
# keyword.dat
# legal
```

```
verb: 3
verb: 3
verb: -13
verb: 13000

# illegal
verb: 3.1
verb: 3a
verb: a
verb: 1e-3
```

If the keyword has multiple entries, the last one is applied. If no entry is provided, the value of the keyword is not changed. When parsing keyword file, Scapar will verify if the value and the key are legal. If not, error information will be printed and program will quit.

The `key` can be any legal string, which should not contain any space, or ":". The following keys are legal:

```
scapar_register_int(&scapar, "verb", &verb);
scapar_register_int(&scapar, "v", &verb);
scapar_register_int(&scapar, "Verb", &verb);
scapar_register_int(&scapar, "V", &verb);
scapar_register_int(&scapar, "verbosity", &verb);
scapar_register_int(&scapar, "level", &verb);
scapar_register_int(&scapar, "k", &verb);
scapar_register_int(&scapar, "print_level", &verb);
scapar_register_int(&scapar, "HiJack", &verb);
scapar_register_int(&scapar, "hello-world", &verb);
```

4.2 Floating Point Number

`scapar_register_float` registers a floating point number, `key` is the name of the keyword, and `addr` is the address where the value is stored.

```
void scapar_register_float(SCAPAR *scapar, const char *key, SCAPAR_FLT *addr);
```

Sample code:

```
{
    char *in = "keyword.dat";
    SCAPAR scapar;
    SCAPAR_FLT b = -1.3;

    /* init */
    scapar_init(&scapar, in);

    /* register */
    scapar_register_float(&scapar, "rhs", &b);
    /* parse */
    scapar_parse(&scapar);

    /* clean up */
    scapar_finalize(&scapar);
}
```

The keyword file contains the value,

```
# keyword.dat
rhs: 3.1e-2
rhs: 3.
rhs: 3.1
rhs: 0.000001
rhs: 21
```

4.3 String

`scapar_register_string` registers a string, `key` is the name of the keyword, and `addr` is the address where the value is stored.

```
void scapar_register_string(SCAPAR *scapar, const char *key, char **addr);
```

Sample code:

```
{
    char *in = "keyword.dat";
    SCAPAR scapar;
    char *name = "jill";

    /* init */
    scapar_init(&scapar, in);

    /* register */
    scapar_register_float(&scapar, "first_name", &name);

    /* parse */
    scapar_parse(&scapar);

    /* clean up */
    scapar_finalize(&scapar);
}
```

The keyword file contains the value,

```
# keyword.dat
first_name: Jenny

first_name: Jack
```

4.4 Table

`scapar_register_table` registers a string, `key` is the name of the keyword, and `addr` is the address where the value is stored. `info` stores the table information, such as number of columns, and type of each column. The information is defined by `SCAPAR_INFO`.

```
void scapar_register_table(SCAPAR *scapar, const char *key, SCAPAR_TABLE *addr, void *info);
```

The following code shows how to register a table and how to set table information.

```
{
    char *in = "table.dat";
    SCAPAR scapar;
    /* table */
    SCAPAR_TABLE tab;

    /* info */
    SCAPAR_INFO info;

    /* 1. init */
    scapar_init(&scapar, in);

    /* 2. register table key words */
    scapar_info_init(&info);
    info.msize = 3;
    info.mtype = NULL;  /* float */
    scapar_register_table(&scapar, "tbl", &tab, &info);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

The following data files show legal table data.

```
# table.dat
tbl:
# first row 1.21 2. 3
#second row
2 11 5.3
2 11 5.3
2 11 0
2 11 .3 2 11 5.
2 11 5.3
2 11 5.3
111 1e-3 22.6
# ending with empty line

# table.dat
tbl:
```

```
# first row 1.21 2. 3
#second row
2 11 5.3
2 11 5.3
2 11 0
2 11 .3 2 11 5.
2 11 5.3
2 11 5.3
# ending with slash (/)
111 1e-3 22.6 /
```

Another example shows how to register a table and how to set table information.

```
{
    char *in = "table.dat";
    SCAPAR scapar;

    /* table */
    SCAPAR_TABLE tab;

    /* info */
    SCAPAR_INFO info;
    SCAPAR_TYPE type[2];

    /* 1. init */
    scapar_init(&scapar, in);

    /* 2. register table key words */
    scapar_info_init(&info);
    type[0] = SCAPAR_T_STR;
    type[1] = SCAPAR_T_INT;
    info.msize = 2;
    info.mtype = type;
    scapar_register_table(&scapar, "tbl", &tab, &info);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

A legal data file:

```
# table.dat
tbl:
# first row
Emma 8
#second row
Jack 23
Jill 16
Mark 3 /
```

4.5 User-defined Type

scapar_init_user_type initializes one user-defined type. The type should be SCAPAR_T_USER0 to SCAPAR_T_USER15.

```
void scapar_init_user_type(SCAPAR *scapar, SCAPAR_TYPE type, SCAPAR_UT_INFO ut);
```

ut carries required info, including conversion function, cleanup function and printing function. The conversion function should be responsible for all user defined operations, such as memory allocation, initialization and marking the item parsed. The printing outputs parsed values. However, if the ”parsed” mark isn’t set to true, the printing function won’t be called. The cleanup function should free all allocated memory and set correct status. They are defined as,

```
typedef void (*SCAPAR_CONV)(void *, FILE *, char *, SCAPAR_ITEM *);
typedef void (*SCAPAR_DESTROY)(void *, SCAPAR_ITEM *);
typedef void (*SCAPAR_PRINT)(void *, SCAPAR_ITEM *);
```

scapar_register_user_type registers a user type. info is optional and is determined by user.

```
void scapar_register_user_type(SCAPAR *scapar, const char *key, void *addr,
     SCAPAR_TYPE type, void *info);
```

The following code shows how to use user-defined type.

```
{
    char *in = "user-mat.dat";
    SCAPAR scapar;
    mat_csr A;
    SCAPAR_UT_INFO ut; /* info for user-defined types */

    /* 1. init */
    scapar_init(&scapar, in);

    /* 2a. define user type */
    ut.conv = func_conv;
    ut.destroy = func_des;
    ut.print = func_print;
    scapar_init_user_type(&scapar, SCAPAR_T_USER0, ut);

    /* 2b. register user defined type */
    scapar_register_user_type(&scapar, "mat_csr", &A, SCAPAR_T_USER0, NULL);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

The data file format is determined by user. When the keyword is found, Scapar will pass control to the conversion function, which should decide when to allocate memory, to set correct value, and to finish. There is one example (`user-mat.c`)that shows more details.

4.6 Section

Section is a special type designed to handle `struct` in C/C++. `addr` is the address of a `struct` object. We should mention that there are two ways for handling struct.

```
void scapar_register_section(SCAPAR *scapar, const char *key, void *addr);
```

`scapar_sec_register_offset` registers a member of a `struct` object. `key` is the keyword, `offset` is the offset in the `struct`, `type` is the type of the member, and `info` is optional information required by the member.

```
void scapar_sec_register_offset(SCAPAR_SEC *s, SCAPAR *scapar, const char *key,
    SCAPAR_ADDR offset, SCAPAR_TYPE type, SCAPAR_INFO *info);
```

`scapar_get_section` gets the section by its keyword.

```
SCAPAR_SEC * scapar_get_section(SCAPAR *scapar, const char *key);
```

4.6.1 First Method

This method treats each member of a `struct` as a regular variable. The following sample code shows to how to register.

```
typedef struct Student_
{
    char *name;
    SCAPAR_INT id;
} Student;

{
    char *in = "keyword.dat";
    SCAPAR scapar;
    Student one;

    /* init */
    scapar_init(&scapar, in);

    /* register */
    scapar_register_int(&scapar, "id", &one.id);
    scapar_register_string(&scapar, "name", &one.name);

    /* parse */
    scapar_parse(&scapar);

    /* clean up */
    scapar_finalize(&scapar);
}
```

The keyword file looks like,

```
# keyword.dat
name: Jenny
id: 100036178
```

4.6.2 Second Method

The second method is to treat the `struct` as a section. Three steps must be taken to register all members:

1. To register the section using `scapar_register_section`.
2. To get the internal section address using `scapar_get_section`.
3. To register each member of the struct using `scapar_sec_register_offset`.

```
typedef struct SEC_STT_
{
    SCAPAR_FLT km;
    SCAPAR_INT dim;
    SCAPAR_INT verb;
    char *fn;

    SCAPAR_VEC vd;  /* vector of int */
    SCAPAR_VEC vf;  /* vector of floating point number */
    SCAPAR_VEC vs;  /* vector of string */
    SCAPAR_VEC vvi; /* vector of vector of int */

} SEC_STT;

int main(void)
{
    char *in = "keyword.dat";
    SCAPAR scapar;
    SEC_STT s;
    SCAPAR_SEC *sec;

    /* 1. init */
    scapar_init(&scapar, in);

    /* 2a. register a section */
    scapar_register_section(&scapar, "sec-stt", &s);

    /* 2b. get section and handle member */
    sec = scapar_get_section(&scapar, "sec-stt");
    assert(sec != NULL);
    /* 2c. register each member */
    scapar_sec_register_offset(sec, &scapar, "dim", offsetof(SEC_STT, dim),
        SCAPAR_T_INT, NULL);

    scapar_sec_register_offset(sec, &scapar, "km", offsetof(SEC_STT, km),
        SCAPAR_T_FLT, NULL);

    scapar_sec_register_offset(sec, &scapar, "verb", offsetof(SEC_STT, verb),
        SCAPAR_T_INT, NULL);

    scapar_sec_register_offset(sec, &scapar, "fn", offsetof(SEC_STT, fn),
        SCAPAR_T_STR, NULL);

    /* vector of int */
    scapar_sec_register_offset(sec, &scapar, "vd", offsetof(SEC_STT, vd),
        SCAPAR_T_VEC, SCAPAR_INFO_INT);

    /* vector of floating point number */
    scapar_sec_register_offset(sec, &scapar, "vf", offsetof(SEC_STT, vf),
        SCAPAR_T_VEC, SCAPAR_INFO_FLT);

    /* vector of string */
    scapar_sec_register_offset(sec, &scapar, "vs", offsetof(SEC_STT, vs),
        SCAPAR_T_VEC, SCAPAR_INFO_STR);

    /* vector of vector of int */
    scapar_sec_register_offset(sec, &scapar, "vvi", offsetof(SEC_STT, vvi),
        SCAPAR_T_VEC, SCAPAR_INFO_VEC_INT);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

The keyword file looks like,

```
# keyword.dat
# section starts here
sec-stt:
# scalar: dimension
dim: 3

# scalar: verb
verb: 2

# scalar: km = 2.2
km: 2.2

# scalar: file name
fn: hello_scapar
```

```
# int vector
vd: 3 \
#
#
3 \
###
3 4

# float vector
vf: 3.1 33 4.3e-2 0.3

# string vector
vs: hello world from \
here

# vector of vector of int
vvi: 5 5
vvi: 5 5 6
vvi: 5 5 6 1 1 2
vvi: 5 5 6 0 1 1 11 1 1 2
vvi: 0 1 1 11 1 1 2
# section ends here, ending mark /
/
```

The usage of vector will be introduced. The advantage of the second method is that vector of section could be defined.

4.7 Vector

Vector is useful for handling array, such as integer array, string array and array of some `struct`. `scapar_register_vec` registers a vector (array). `key` is the keyword, `addr` is the address of the vector, and `info` has the member information for data conversion and memory management. The `info` has the type of `SCAPAR_INFO`, which is defined as,

```
typedef struct SCAPAR_INFO_
{
    SCAPAR_TYPE type; /* type of a vector member */
    SCAPAR_INT tsize; /* size of a vector member */

    SCAPAR_INT msize;
    SCAPAR_TYPE *mtype;

} SCAPAR_INFO;

void scapar_register_vec(SCAPAR *scapar, const char *key, SCAPAR_VEC *addr,
    void *info);
```

Scapar has a few pre-defined info objects, such as,

```
SCAPAR_INFO_INT,        /* member is an integer */
```

```
SCAPAR_INFO_FLT,      /* member is floating point */
SCAPAR_INFO_STR,     /* member is string */
SCAPAR_INFO_VEC_INT,    /* member is a vector of int */
SCAPAR_INFO_VEC_FLT,  /* member is a vector of floating point number */
SCAPAR_INFO_VEC_STR, /* member is a vector of string */
```

The following sample code shows how to register a vector. Pre-defined info is used. However, users can define theirs too.

```
{
    char *in = "vec.dat";
    SCAPAR scapar;

    /* vector */
    SCAPAR_VEC vi;
    SCAPAR_VEC vf;
    SCAPAR_VEC vs;

    /* 1. init */
    scapar_init(&scapar, in);

    /* 2. register vector key words (vector of a type) */
    scapar_register_vec(&scapar, "vi", &vi, SCAPAR_INFO_INT);
    scapar_register_vec(&scapar, "vf", &vf, SCAPAR_INFO_FLT);
    scapar_register_vec(&scapar, "vs", &vs, SCAPAR_INFO_STR);

    /* 3. parse */
    scapar_parse(&scapar);

    /* 4. detroy */
    scapar_finalize(&scapar);
}
```

The data file has the form,

```
# vec.dat
# int vector
vi: 3 \
#
#
    3 \
###
3 4

# float vector
vf: 3.1 33 4.3e-2 \
    0.3

#string vector
vs: hello world from \
    here
```

There is one example shows how to use vector of section. Scapar also has three special functions to handle commonly used vector of integer, vector of floating point number, and vector of string.

```
void scapar_register_vec_int(SCAPAR *scapar, const char *key, SCAPAR_VEC *addr);
void scapar_register_vec_float(SCAPAR *scapar, const char *key, SCAPAR_VEC *addr);
void scapar_register_vec_string(SCAPAR *scapar, const char *key, SCAPAR_VEC *addr);
```

Chapter 5

Subroutines

Scapar has implemented various subroutines to management the parsing. Some are for internal use and some are for users.

5.1 Scapar

`scapar_init` initializes the library by passing the keyword data file name. This subroutine is mandatory.

```
void scapar_init(SCAPAR *scapar, const char *fn);
```

`scapar_finalize` destroys the parser and cleans up memories it allocated.

```
void scapar_finalize(SCAPAR *scapar);
```

`scapar_set_comment_mark` changes the default comment mark ('#').

```
void scapar_set_comment_mark(SCAPAR *scapar, char m);
```

`scapar_set_dup_check` enables the parser check duplicated keywords or not. If duplication check is enabled, performance will be slowed down.

```
void scapar_set_dup_check(SCAPAR *scapar, SCAPAR_BOOL check);
```

`scapar_set_case_sensitive` sets if the parser is sensitive to letter cases, and the default is not case sensitive. In this case, keywords, tbl and TBL, are treated as the same.

```
void scapar_set_case_sensitive(SCAPAR *scapar, SCAPAR_BOOL cs);
```

`scapar_set_max_line_length` sets the maximal line length of the input data file. The default is 4096.

```
void scapar_set_max_line_length(SCAPAR *scapar, size_t m);
```

`scapar_set_sec_ending_mark` sets the ending mark of a section. The default is slash (/).

```
void scapar_set_sec_ending_mark(SCAPAR *scapar, char *m);
```

`scapar_print_by_key` outputs values by the provided key.

```
void scapar_print_by_key(SCAPAR *scapar, const char *key);
```

`scapar_print` outputs all parsed values.

```
void scapar_print(SCAPAR *scapar);
```

`scapar_get_section` gets internal address of a section by its key.

```
SCAPAR_SEC * scapar_get_section(SCAPAR *scapar, const char *key);
```

`scapar_query` is an internal subroutine, and it gets the internal address of the object stores its information, such as name and type.

```
SCAPAR_ITEM * scapar_query(SCAPAR *s, const char *key);
```

5.2 Table

The following three subroutines are internal, which create a table, destroy a table and add a row to a table.

```
void scapar_table_create(SCAPAR_TABLE *v, const char *key, SCAPAR_IDX cols,
     const SCAPAR_TYPE *type);

void scapar_table_destroy(SCAPAR_TABLE *v);
int scapar_table_add_entry(SCAPAR_TABLE *v, char *buf, SCAPAR_INT lno, char cmt);
```

`scapar_table_get_num_cols` gets the number of columns.

```
SCAPAR_IDX scapar_table_get_num_cols(SCAPAR_TABLE *t);
```

`scapar_table_get_num_entries` gets the number of rows (entries).

```
SCAPAR_IDX scapar_table_get_num_entries(SCAPAR_TABLE *t);
```

`scapar_table_get_key` gets the keyword of a table.

```
char * scapar_table_get_key(SCAPAR_TABLE *t);
```

`scapar_table_get_data_type` gets the data type of the n-th column.

```
SCAPAR_TYPE scapar_table_get_data_type(SCAPAR_TABLE *t, SCAPAR_IDX n);
```

`scapar_table_get_value` gets the address of the object from certain row and certain column. If the data type is integer, the returned value is pointer to integer. If the data type is floating point number, the returned value is pointer to it.

```
void * scapar_table_get_value(SCAPAR_TABLE *t, SCAPAR_IDX row, SCAPAR_IDX col);
```

`scapar_table_get_col` gets address of certain column.

```
void * scapar_table_get_col(const SCAPAR_TABLE *t, SCAPAR_IDX col);
```

`scapar_table_print` outputs a table.

```
void scapar_table_print(const SCAPAR_TABLE *v);
```

5.3 Vector

`scapar_vec_initialized` checks if a vector is initialized or not.

```
SCAPAR_BOOL scapar_vec_initialized(SCAPAR_VEC *vec);
```

`scapar_vec_init` initializes a vector, type is the member type, tsize is the size of the member, and key is the keyword.

```
void scapar_vec_init(SCAPAR_VEC *vec, SCAPAR_TYPE type, SCAPAR_INT tsize, const char *key);
```

`scapar_vec_destroy` destroys a vector.

```
void scapar_vec_destroy(SCAPAR_VEC *vec);
```

`scapar_vec_add_entry` adds an entry to a vector and e is the pointer to the entry.

```
void scapar_vec_add_entry(SCAPAR_VEC *v, void *e);
```

`scapar_vec_get_size` gets the size of a vector.

```
SCAPAR_IDX scapar_vec_get_size(SCAPAR_VEC *v);
```

`scapar_vec_int_get_value` gets the n-th entry from a vector of integer.

```
SCAPAR_INT scapar_vec_int_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_float_get_value` gets the n-th entry from a vector of floating point number.

```
SCAPAR_FLT scapar_vec_float_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_string_get_value` gets the n-th entry from a vector of string.

```
char * scapar_vec_string_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_table_get_value` gets the n-th table from a vector of table.

```
SCAPAR_TABLE scapar_vec_table_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_vec_get_value` gets the n-th vector from a vector of vector.

```
SCAPAR_VEC scapar_vec_vec_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_get_value` is a general getting value subrouting, which gets the n-th entry of a vector.

```
void * scapar_vec_get_value(SCAPAR_VEC *v, SCAPAR_IDX n);
```

`scapar_vec_print` outputs a vector.

```
void scapar_vec_print(SCAPAR_VEC *v);
```

5.4 Section

`scapar_sec_print_by_key` outputs the value of a member by key from a section.

```
void scapar_sec_print_by_key(const SCAPAR_SEC *s, const char *key);
```

`scapar_sec_print` outputs parsed value of a section.

```
void scapar_sec_print(SCAPAR_SEC *s);
```

5.5 Utilities

Memory management subroutines:

```
void * scapar_malloc(size_t size);
void * scapar_calloc(size_t nmemb, size_t size);
void * scapar_realloc(void *ptr, size_t size);
void scapar_free(void *ptr);
```

`scapar_string_to_lower` converts string to lower case.

```
void scapar_string_to_lower(char *str);
```

`scapar_atof` and `scapar_atoi` convert string to integer and floating point number.

```
SCAPAR_FLT scapar_atof(const char *ptr);
SCAPAR_INT scapar_atoi(const char *ptr);
```

`scapar_info_init` initializes an info object.

```
void scapar_info_init(SCAPAR_INFO *in);
```

`scapar_info_copy` copies an info object.

```
SCAPAR_INFO * scapar_info_copy(SCAPAR_INFO *in);
```

`scapar_info_destroy` destroys an info object.

```
void scapar_info_destroy(SCAPAR_INFO *in);
```