Petunia 项目开发记录

陆巍

2023 年 10 月 5 日

前言

目录

前言 i

第一章 需求分析 1

第二章 设计 2

2.1 流程图 2

2.2 词法分析 3

2.2.1 状态转换图 3

2.2.2 用于 TOML 状态转换表解析的状态转换图 6

附录 A PCRE 的简单使用方法 7

A.1 在 Ubuntu 22.04 上安装 7

A.2 头文件与构建参数 7

A.3 基础使用方法 8

A.3.1 pcre_compile() 函数 9

A.3.2 pcre_exec() 函数 10

附录 B PCRE 的正则表达式规则 12

附录 C SQLite 的简单使用方法 14

C.1 在 Ubuntu22.04 上安装 SQLite 14

附录 D 开发日记 **15**

D.1 2023 年 10 月 15

D.1.1 10 月 5 日 15

D.1.2 10 月 6 日 15

D.1.3 10 月 12 日 16

D.2 2023 年 11 月 16

D.2.1 11 月 13 日 16

D.2.2 11 月 16 日 17

D.2.3 11 月 18 日 17

D.2.4 11 月 21 日 17

D.2.5 11 月 28 日 18

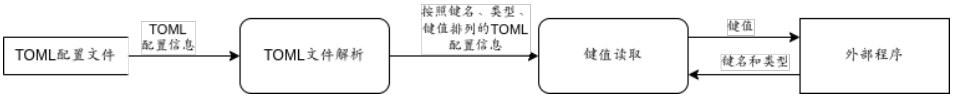
D.3 2023 年 12 月 18

D.3.1 12 月 02 日 18

D.3.2 12 月 03 日 19

第一章　需求分析

本项目的需求分析方面很简单，也很明确。这里选用数据流图（DFD）来表达。



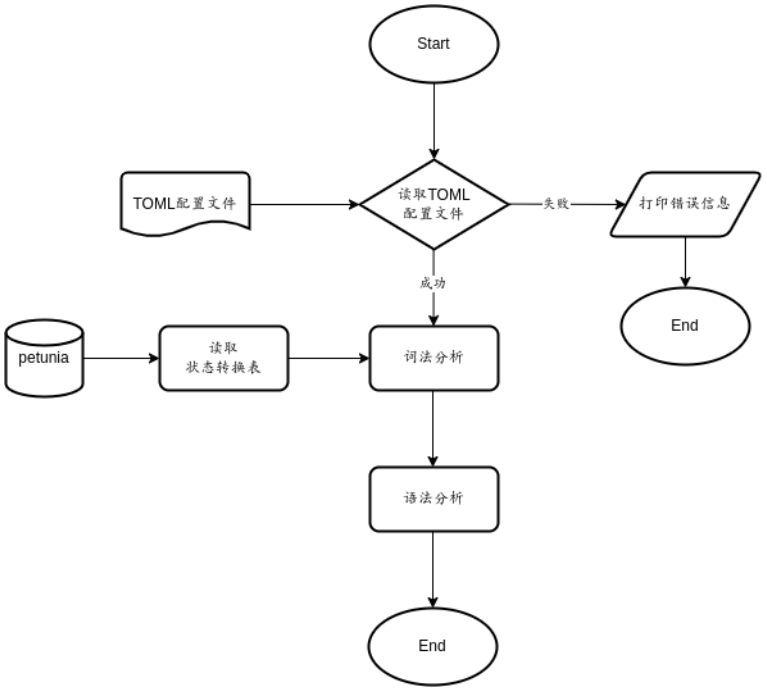
此图由 draw.io 绘图工具绘制后导出。

第二章　设计

2.1　流程图

本项目规模很小，并且使用的编程语言是面向结构的 C 语言，所以使用流程图来设计。

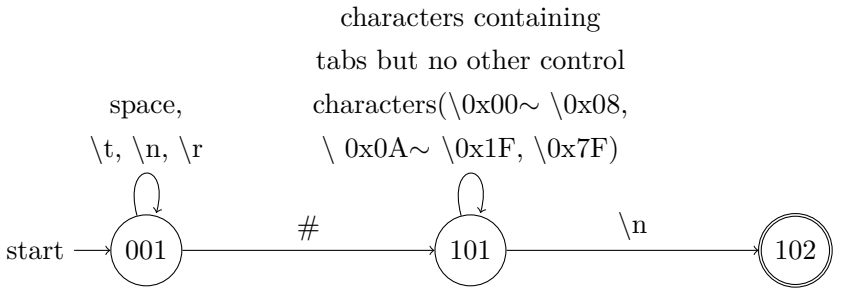
下面的流程图是针对本项目中的示例程序，在实际使用中，读取文件方面的操作由相关调用代码自行处理。



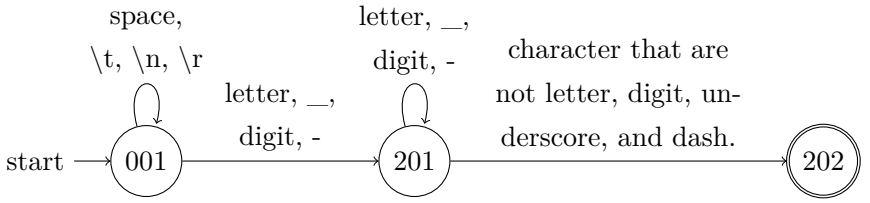
2.2　词法分析

2.2.1　状态转换图

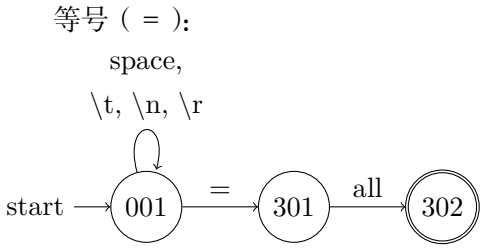
注释（comment）



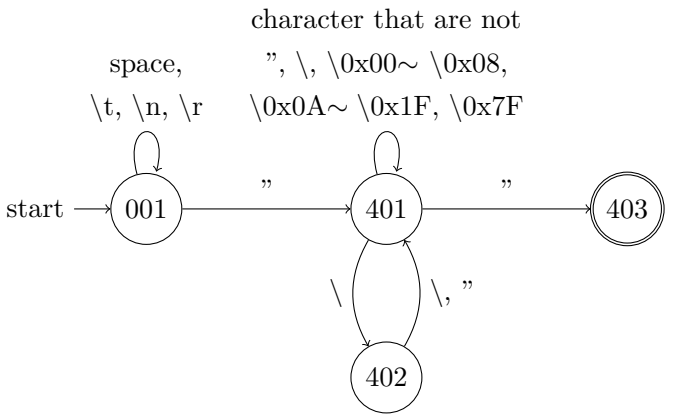
裸键（bare key）



运算符

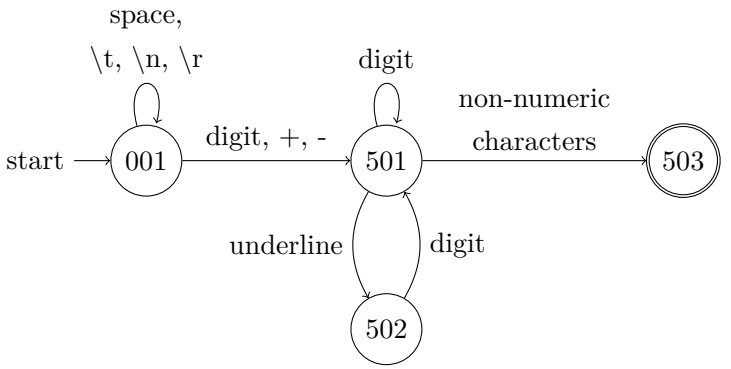


字符串（string）



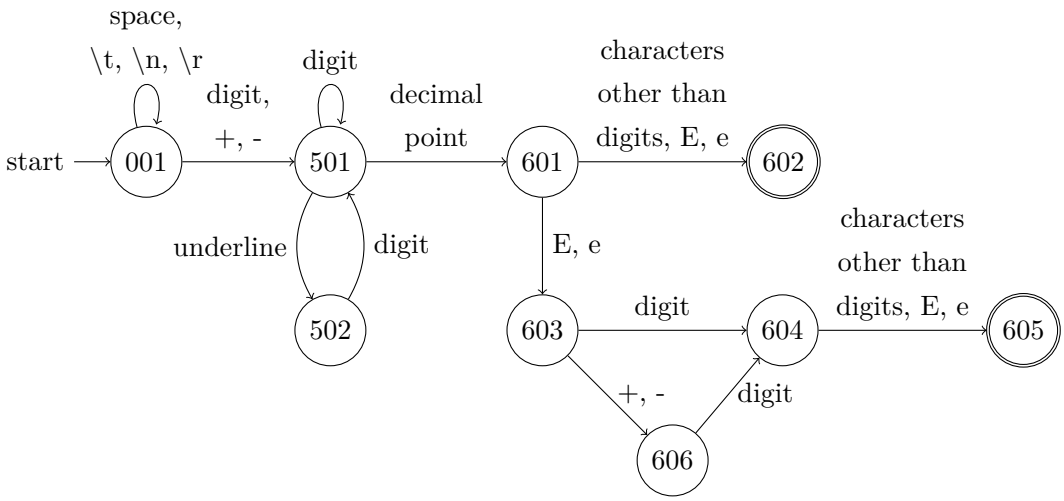
按照当前的实际需要，暂时不支持多行字符串，另外转义字符目前也只支持反斜杠和双引号。

整数（integer）



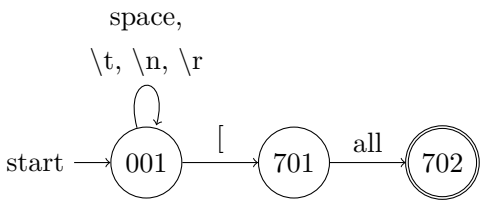
目前暂时不支持 16 进制、8 进制。

浮点数（fractional）

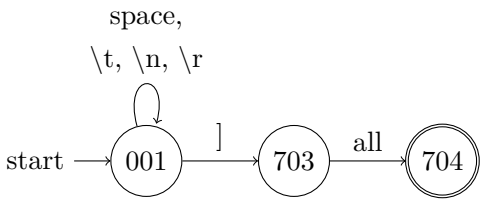


界符

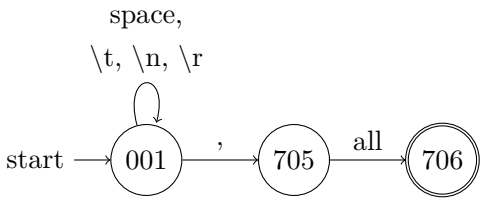
左中括号：



右中括号：



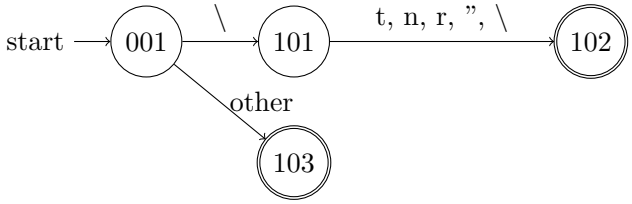
逗号：



2.2.2　用于 TOML 状态转换表解析的状态转换图

这个标题有些绕，实际对应的是上一节的内容。上一节的状态转换图最后是要用一张状态转换表（这里使用 CSV 文件）来实现的，那么这张表在程序读入时，需要把表文件中的内容转换保存在 C 语言的数组中。但是，因为这张表文件的内容会包含不能被直接转换的内容，比如一些不可见的控制字符，所以就需要做一些加工转换。这样一来，在我们的程序中就又增加了一个词法分析，只是比较简单，并且在具体实现时将不再把这个转换规则以外部文件的形式存放，而是直接编写在程序中。

这里的状态转换，核心部分就是转义字符的处理。



附录 A　PCRE 的简单使用方法

C 语言处理正则表达式时一般可以选择 POSIX（Portable Operating System Interface，是一种跨平台标准）的 regex.h 或 PCRE（Perl Compatible Regular Expressions）的 pcre.h。因为 PCRE 易用、功能强大，性能超过了 POSIX，所以在本项目中使用 PCRE 来处理正则表达式。

A.1　在 Ubuntu 22.04 上安装

安装命令如下：

sudo apt install libpcre3 libpcre3-dev

A.2　头文件与构建参数

代码如下：

```
...
#include <pcre.h>
...
```

CMakeLists.txt 文件中的配置：

```
target_link_libraries(pcre_test
                      pcre)
```

A.3　基础使用方法

示例代码：

```c
// @copyright Copyright 2023 Willard Lu
//
// Use of this source code is governed by an MIT-style
// license that can be found in the LICENSE file or at
// https://opensource.org/licenses/MIT.

#include <stdio.h>
#include <string.h>
#include <pcre.h>

int main() {
  pcre *re; // 保存编译后的正则表达式
  const char *error; // 保存错误信息
  int erroffset; // 保存错误位置
  int rc; // 保存匹配串的偏移位置
  const char *pattern = "-.+[a]+."; // 示例用正则表达式
  // 1. 编译正则表达式
  re = pcre_compile(pattern, 0, &error, &erroffset, NULL);
  if (re == NULL) {
    // 输出错误信息
    fprintf(stderr, "PCRE compilation failed at offset %d:
↪  %s\n", erroffset, error);
    return 1;
  }
  // 2. 模式匹配
  const char *subject = "--===abcfooabcfoo";
  rc = pcre_exec(re, NULL, subject, strlen(subject), 0, 0,
↪  NULL, 0);
  if (rc < 0) {
    if (rc == PCRE_ERROR_NOMATCH) {
      printf("No match\n");
```

```
    }
    else {
      printf("PCRE execution failed at offset %d: %d\n",
↪   erroffset, rc);
    }
    pcre_free(re);
    return 1;
  }
  printf("Matched %d groups\n", rc);
  // 释放内存
  pcre_free(re);
  return 0;
}
```

从以上代码可以看到简单使用方法的步骤为：

1. 为了更高效地处理正则表达式，需要先把正则表达式字符串进行编译。为此要定义一个 pcre 结构的指针用于存放编译结果。
2. 按惯例定义相关变量，包括错误信息、错误位置、匹配串偏移位置。
3. 定义并赋值正则表达式字符串。在本项目中，这个由外部提供。
4. 使用 pcre_compile() 函数编译正则表达式。
5. 使用 pcre_exec() 函数进行模式匹配。
6. 释放第一步定义的 pcre 结构指针指向的内存空间。

上述步骤只是最基本的，实际上还应按照示例程序中所示添加一些错误判断处理的代码。

A.3.1　pcre_compile() 函数

原型：

```
pcre *pcre_compile(const char *pattern,
                   int options,
```

```
                    const char **errptr,
                    int *erroffset,
                    const unsigned char *tableptr);
```

功能：将一个正则表达式编译成一个内部表示，在匹配多个字符串时，可以加速匹配。

参数：

- pattern 正则表达式；
- options 为 0，或者其他参数选项；
- errptr 出错消息；
- erroffset 出错位置；
- tableptr 指向一个字符数组的指针，可以设置为空 NULL。

A.3.2　pcre_exec() 函数

原型：

```
int pcre_exec(const pcre *code,
              const pcre_extra *extra,
              const char *subject,
              int length,
              int startoffset,
              int options,
              int *ovector,
              int ovecsize);
```

功能：该函数使用与 Perl 类似的匹配算法，将已编译的正则表达式与给定主题字符串进行匹配。它会返回捕获子串的偏移量。

参数：

- code：指向提供的正则表达式编译后的模式；

- extra：指向相关的 pcre[16|32]_extra 结构、可为 NULL；
- subject：需要匹配的字符串；
- length：待匹配字符串的长度（字节）；
- startoffset：匹配的开始位置；
- options：选项位；
- ovector：指向一个结果的整型数组；
- ovecsize：数组大小（应为 3 的整数倍）。

附录 B　PCRE 的正则表达式规则

Regular Expression Basics	
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
\	Escapes a special character

Regular Expression Quantifiers	
*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more

Default is greedy. Append ? for reluctant.

Regular Expression Groups	
(...)	Capturing group
(?P<Y>...)	Capturing group named Y
(?:...)	Non-capturing group
(?>...)	Atomic group
(?|...)	Duplicate group numbers
\Y	Match the Y'th captured group
(?P=Y)	Match the named group Y
(?R)	Recurse into entire pattern
(?Y)	Recurse into numbered group Y
(?&Y)	Recurse into named group Y
\g{Y}	Match the named or numbered group Y
\g<Y>	Recurse into named or numbered group Y
(?#...)	Comment

Regular Expression Character Classes	
[ab-d]	One character of: a, b, c, d
[^ab-d]	One character except: a, b, c, d
[\b]	Backspace character
\d	One digit
\D	One non-digit
\s	One whitespace
\S	One non-whitespace
\w	One word character
\W	One non-word character

Regular Expression Assertions	
^	Start of string
\A	Start of string, ignores m flag
$	End of string
\Z	End of string, ignores m flag
\b	Word boundary
\B	Non-word boundary
\G	Start of match
(?=...)	Positive lookahead
(?!...)	Negative lookahead
(?<=...)	Positive lookbehind
(?<!...)	Negative lookbehind
(?()|)	Conditional

Regular Expression Escapes	
\Q..\E	Remove special meaning

Regular Expression Flags	
i	Ignore case
m	^ and $ match start and end of line
s	. matches newline as well
x	Allow spaces and comments
J	Duplicate group names allowed
U	Ungreedy quantifiers
(?iLmsux)	Set flags within regex

Regular Expression Special Characters	
\n	Newline
\r	Carriage return
\t	Tab
\0	Null character
\YYY	Octal character YYY
\xYY	Hexadecimal character YY
\x{YY}	Hexadecimeal character YY
\cY	Control character Y

Regular Expression Posix Classes	
[:alnum:]	Letters and digits
[:alpha:]	Letters
[:ascii:]	Ascii codes 0 - 127
[:blank:]	Space or tab only
[:cntrl:]	Control characters
[:digit:]	Decimal digits
[:graph:]	Visible characters, except space
[:lower:]	Lowercase letters
[:print:]	Visible characters
[:punct:]	Visible punctuation characters
[:space:]	Whitespace
[:upper:]	Uppercase letters
[:word:]	Word characters
[:xdigit:]	Hexadecimal digits

测试正则表达式的网站：DebuggexBeta

附录 C　SQLite 的简单使用方法

C.1　在 Ubuntu22.04 上安装 SQLite

使用如下命令进行安装：

```
1 sudo apt install sqlite3
2 sudo apt install libsqlite3-dev
```

注意，第 1 行命令实际上只是提供了一个终端调试界面；要想在程序中使用 SQLite，必须运行第 2 个命令来安装相关库，然后我们的 C 语言代码中的“#include <sqlite3.h>”才能找到所需文件。

为了更方便操作 SQLite，可以安装下面这个工具：

```
1 sudo apt install sqlitebrowser
```

附录 D　开发日记

D.1　2023 年 10 月

D.1.1　10 月 5 日

这个项目最初只是打算使用简单的判断方法来解决，但想到以后要开发其他编译器，所以先用这个微小项目来练练手。项目将按照编译器开发方法来实现，当然，本项目过于微小，大概只会用到词法分析与语法分析。

Windows 系统和 Linux 系统在文本处理上是有些差异的，其中的换行就不相同。Windows 系统中的换行实际上包含了两个字符，即 $\backslash r$（回车，0xD）与 $\backslash n$（换行，0xA），而 Linux 系统中只有 $\backslash n$（换行，0xA）。我现在主要使用的是 Linux 系统，但为了兼顾 Windows 系统，可能需要在读入配置文件后，先把其中的 $\backslash r\backslash n$ 替换成 $\backslash n$，然后才做词法分析。

目前暂时只解析裸键名，引号键名以后再考虑。

D.1.2　10 月 6 日

在绘制状态转换图时，我们看到在识别某些内容时，可以按照不同的权衡有不同的处理方式。例如在判断整数时，可以在出现非数字符号就截止，也可以规定必须要出现空格、换行符或 # 才截止，两种方式一个宽松，一个严格，各有各的好处与不足。前一种方式对 TOML 的书写格式比较宽松，但也因为过于宽松可能导致混乱，并增加后期处理的负担。后一种方式要求严格，书写时会有更多约束，但可以减少后期处理的工作量。这里说的后期处理主要是指语法分析阶段。

10 月 7 日

随着状态转移图绘制的深入，会让人感到越来越繁琐，或许应该创建一个专门的工具来绘制，并且在绘制完成后自动转换成相应表格直接供词法分析器调用。这个工具的原理并不复杂，麻烦的是图形操作方面的支持问题，这将涉及到图形库方面，这是一个老话题了，先放一放。

D.1.3　10 月 12 日

绘制状态转换图时，我曾经想到对于不合法的符号要如何在图中去处理，这是一种流程图的思维习惯。实际上，在状态转换图中并不需要显式指明如何处理不合法的符号，而是已经暗含了处理方式。合法的符号串可以从状态转换图的开始（start）处走到某一个终点，不合法的符号是没有路径的，在程序处理上会自动跳到错误处理模块，通常会向用户报告某行某列出现词法错误。通常情况下，每发现一个错误就退出程序并报告此错误，也可以把每一行视为一个单元，全部扫描后统一报告。全部扫描的方式还有一些细节问题需要考虑，并非简单的逐行处理就可以。

在把状态转换图映射为状态转换表时，需要把使用到的符号、状态都列出来，这项工作的繁琐程度会随着语言的复杂程度的增加而增加。对于本项目，即使只是简化版的，其状态转换图已经有些繁琐。

D.2　2023 年 11 月

D.2.1　11 月 13 日

在绘制状态转换图时，对于各个状态的编号，一开始我会习惯性的从 1 开始顺序编写。这样做对于后面的修改并不方便，因为每次修改中间的编号都要对后续的编号重新编写。因此，现在改为使用分段编号，例如注释是 100 开头，裸键使用 200 开头。这种方法就需要在状态转换表中增加一列参数来标明编号，以取代原来隐含的自然顺序编号。在实际的程序处理中，会多出一些用于判断编号的代码，虽然会增加一点开销，但有利于设计。

原本我打算在 LibreOffice Draw 中用一张图来完整描绘状态转换图，但发现这张图越来越大，不利于观看，因此将其拆分为各段，并使用 LaTeX 的 tikz 宏包来绘制。虽然也可以在 LibreOffice Draw 中分页绘制，但为了方便本

记录的查阅，就还是放在 LaTeX 中吧。

D.2.2　11 月 16 日

原来的状态转换图中，无意中把数组用词法分析的形式画上去了，这实际上是不对的，正确的情况应该只是有数组中使用的定界符，即左右中括号。

D.2.3　11 月 18 日

当编写状态转换表时，会面临以何种方式来实现的问题。最简单的方法是把这个表直接放在程序中，使用一个数组来保存，但这样做不灵活，因为每次修改都需要重新编译程序。如果我们把这个表用 CSV 格式存放在外部的话，灵活性确实有了，但增加了对这个表进行解析的工作。我们不能指望存放这张表的文件中全部是文本字符，应该考虑其中可能包含非文本字符的情况，因此从更通用的角度出发，需要引入转义字符的方式来可视的实现此表的编辑。毕竟不可见的符号编辑时需要使用二进制编辑工具来处理，不直观。这里我将按照 C 语言中转义字符的使用规则来处理，会在程序中添加一套词法解析的代码，只不过这个词法解析代码的规则将直接写在程序中，不再以外部文件的形式存放。

至此，我们可以看到，这个小项目中出现了两个词法解析，一个用于解析 TOML 配置文件，一个用于解析 TOML 的词法规则文件。

D.2.4　11 月 21 日

使用 CSV 格式来保存状态转换关系，还是存在一些符号方面的问题。通常情况下，CSV 分隔数据使用的是逗号，那么如果数据中包含逗号，那就需要做转义处理，一般是使用双引号。这样做感觉还是不算用户友好，所以我计划使用 SQLite 这一轻量级数据库来保存状态转换表。使用 SQLite 自然需要增加一些开销，包括引入相关源码文件和数据库管理工具，目前看来是可以接受的。这个就是数据符号与格式符号的混淆问题。

对于状态转换表中的实际内容，并不象我们绘制的图形中那样，让人想到的是一个一个的字符判断语句，具体的处理方法是使用正则表达式的规则字符串来实现。这样做可以实现通用性。例如，我们在判断一个字符是否属于字母时，一般的 C 语言代码中可以使用 ASCII 码的数值比较来判断，但这样做实

际上就把规则固化在程序中，以后做调整时需要修改程序。使用正则表达式的规则字符串的话，就把程序与具体的转换规则彻底分离，以后的规则调整不再需要修改程序重新编译了。不管是什么规则，在程序中都只是简单的与内容无关的一个判断语句。

D.2.5　11 月 28 日

开发这个工具的目的是读取 TOML 配置文件，那么应该向用户提供什么形式的数据输出呢？一般而言，读入 TOML 配置文件后，应该生成一个字符串，其中包含了数据类型、键名与键值。但仅仅是字符串的话，对于用户来说还是有些麻烦，因为需要把字符串转换成用户程序中的数据类型，这项工作应该交由本项目来完成。因此，我们还需要在本项目中添加各种数据类型的转换函数供用户调用。当然，不管最后是什么数据类型，都需要先把配置文件中的内容读取到一个字符串中保存以供后续的调用。

读取 TOML 配置文件后，返回给用户数据的方式有两种。一种是由用户指定键名来决定读取对应的键值，一种是把配置文件中的数据全部读取。第 1 种方式下，在用户程序中已经事先明确定义了相关变量；第 2 种方式对于 C 语言这样的程序，并不能直接使用，需要一些转换。第 1 种方式，可以先通过读取函数把 TOML 配置文件的内容读入到一个字符串变量中，然后再通过相应数据类型读取函数获得指定键名的键值。第 2 种方式因为需要使用一些转换，目前看来在用户程序中的后续调用上会增加许多开销，故此暂时不建议使用。

D.3　2023 年 12 月

D.3.1　12 月 02 日

我们知道软件项目的开发周期包括需求分析、设计、开发、测试、部署和维护，一共六个阶段。虽然本项目非常小，但也是如此。软件项目的开发周期中，会使用到很多工具，目的是提高整个开发过程的自动化程度，让我们可以把更多的精力放在项目真正的核心上。实际上对于大部分的软件项目，特别是进入机器人项目、人工智能项目时，项目的核心都不是计算机编程。因此在选择开发方法、工具时，一个重要的指标就是自动化程度。

本项目的开发过程确实可以使用脚本语言来逐步实现自动化，但是因为项目过小，并不具备代表性，所以暂时不做这样的处理，等开发相对完整独立的项目时再说。

D.3.2　12 月 03 日

对于本项目在需求分析阶段的工具，我选用了数据流图（DFD），原因大家一看就明白。另外，在绘制 DFD 时，我用的是 draw.io 工具，没有使用 tikz。这是因为 tikz 中并没有 DFD 方面的专用库，需要自己手动去绘制，稍显麻烦。

以前说过词法分析对应的状态转换表，表里面的字符在具体实现时用的是正则表达式的判断字符串。这样一来，实际上就不再需要针对状态转换表的进一步解析。