Petunia 项目开发记录

陆巍

2023 年 10 月 5 日

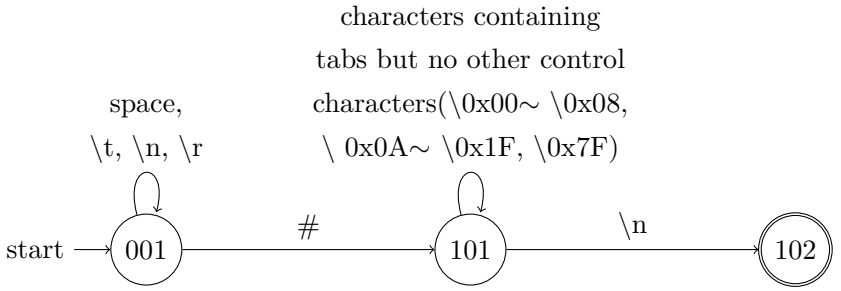
前言

目录

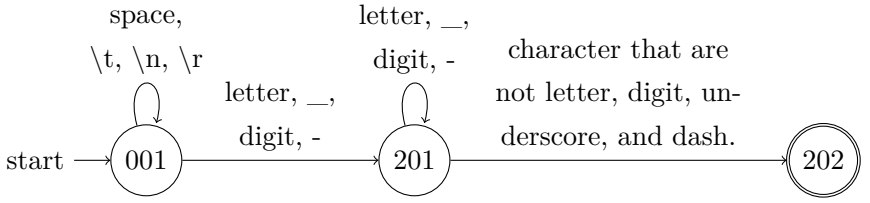
第一章　词法分析

1.1　状态转换图

1.1.1　注释（comment）

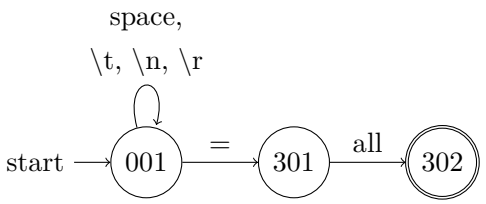


1.1.2　裸键（bare key）

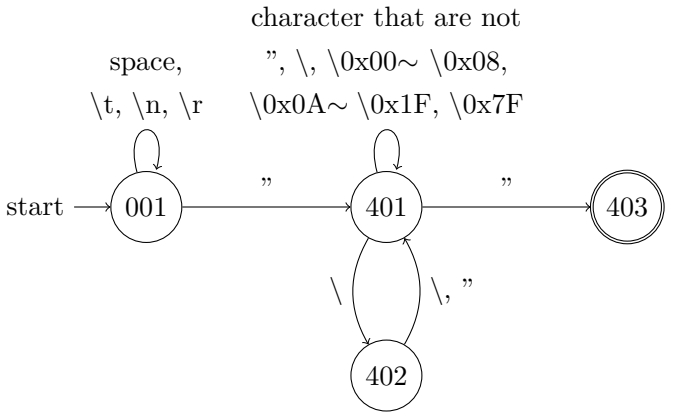


1.1.3　运算符

等号（ = ）:

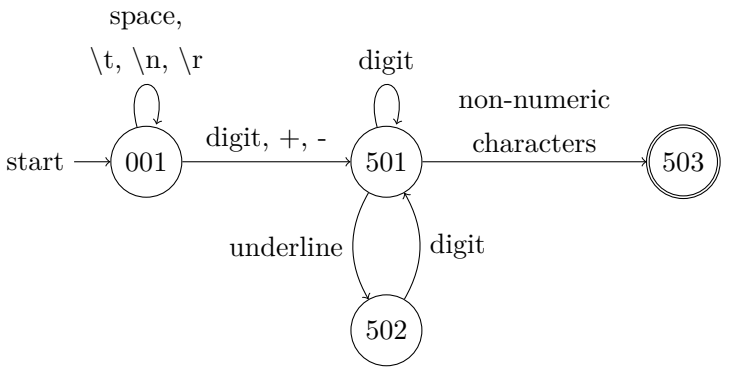


1.1.4　字符串（string）



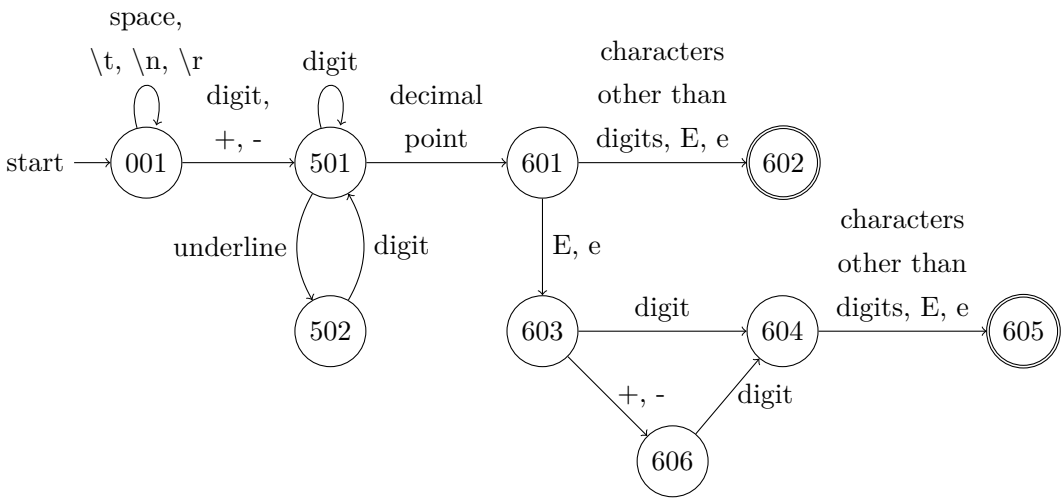
按照当前的实际需要，暂时不支持多行字符串，另外转义字符目前也只支持反斜杠和双引号。

1.1.5　整数（integer）



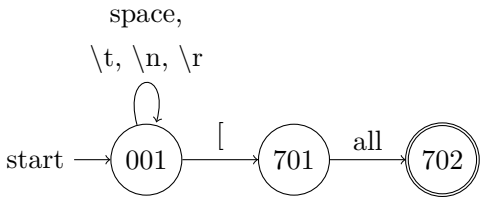
目前暂时不支持 16 进制、8 进制。

1.1.6　浮点数（fractional）

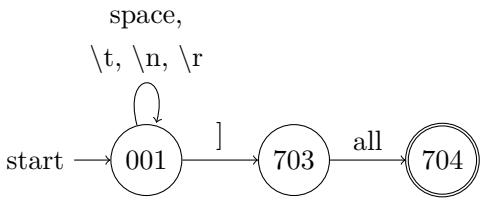


1.1.7　界符

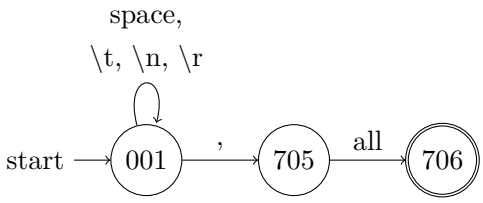
左中括号：



右中括号：



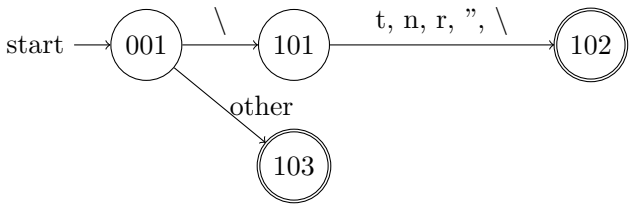
逗号：



1.2　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。



第二章　开发日记

2.1　2023 年 10 月

2.1.1　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。

2.1.2　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。

2.1.3　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。

2.2　2023 年 11 月

2.2.1　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。

2.2.2　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。

2.2.3　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。

2.2.4　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。

2.2.5　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。

2.3　2023 年 12 月

2.3.1　12 月 02 日