**概述**

对于词法分析器，一般来说，有限自动机做词法分析的方法比较常用，这里面也有很多自动进行词法分析的软件。

但对于一个小的编译器而言，例如ucc，采用的就是手工打造词法分析器的方式。

我的做法是，使用EBNF范式进行语法表达，然后进行递归。

**EBNF范式及代码格式表示**

* **输入的标识和规则文件格式**

参考lex或者类似的自动构造词法分析的软件，他们都需要由程序员书写好相应的匹配规则。

因此，相应的，我们在使用正则引擎进行匹配的时候，还需要设计好匹配的规则，需要约定好读取的格式。

在此，对于读取的规则，做如下的格式约定。

这里面应当包含两部分：

* + - 声明
    - 规则

实际上lex等等软件里面的规定更为复杂，还有第三部分的c代码balabala一堆东西。

出于简便考虑，将两者分开在两个文件当中进行存放。（一开始我使用的是txt，后来发现，对于换行符的处理非常麻烦，因此最后写进代码里，分别存储在两个文件中，分别是lex\_statement.c和rule.c中）

对于规则的正则表达式的错误处理，因为这是需要匹配的规则，数量是有限的，而不是文本本身，因此，这里默认规则和声明都是正确的，不做错误处理。由编译器设计者保证这些规则的书写是正确的，当然，实际写代码的时候，还是要做点错误处理的，因为会手误写错，到时候调试起来会很麻烦。

* + **声明的格式约定**

每一行一条声明。

声明名称和声明的内容使用单个冒号进行分隔，并且，在声明名称中不能存在冒号。

有两种模式，一个是枚举，一个是区间。

枚举的字符之间不使用任何符号进行分隔。区间的字符之间使用‘-’。如果在枚举的字符中存在‘-’符号，请不要放在第二个位置，因为这会无法分辨。

需要注意的是，这不能有多余的空格，因为空格本身就在执行字符集中。

例如，约定数字的枚举方式为：

|  |
| --- |
| digit:0123456789 |

而区间方式为：

|  |
| --- |
| digit:0-9 |

理论上，两种方式可以并存，比如0,1,3-8这种，但是，为了简单，如果需要这种情况，则统一用第一种方式列举就好。

* + **规则的格式约定**

这里面只考虑几种类型：

* + - a|b 这个字符可以是a或者b
    - ab a和b连接
    - a\* 0到无限个a
    - a? 表示匹配0或者1个

同样每一行都表示一条规则，并且每条规则，只包括一种类型，由于这个条件的存在，无需为几种类型设置优先级。

在规则中对于声明的引用，使用$开头，带大括号{}，括号内的内容是引用的声明名称。

同样的，使用冒号，将规则名称和规则内容进行切分，规则名称不应当存在冒号。

* + - * 对于子规则的引用

由于每条规则只包括一个类型，对于普通的EBNF范式能表达的语句，在实际代码中，需要多条规则组合才可以完成，这时候，子规则的引入是必然的。

在实际中，使用&{child\_rule}表示子规则。

例如，对于以下一条复合规则：

|  |
| --- |
| TK\_id:${letter}(${letter}|${digit})\* |

就是用于表示C代码中的id，该规则表示以letter开头，后面跟随若干个letter或者digit。（实际上还有下划线之类的开头）

然而在这个代码中，为了编程的便捷，这条复合规则，被改为如下若干条规则：

|  |
| --- |
| &TK\_id1:${letter}|${digit}  &TK\_id2:&{&TK\_id1}\*  TK\_id:${letter}&{&TK\_id2} |

这会一定程度上增加rule的复杂度，但是简化了编程。

另外，对于这个例子中以‘&‘符号开头的规则。

说明如下：这些&符号开头的规则，是作为某条规则的子规则来处理的，目的是为了考虑到后面会描述到的左递归问题，在实际匹配某个词串的时候，除非被上层的父规则所引用，否则这些规则会直接跳过，而不进行匹配。

以上面的这个例子为例，TK\_id是以letter开头的，如果不这么跳过前面两条规则，那么很可能digit也可以作为开头，去匹配到TK\_id1这个规则——然而这是不对的。

由于不限制自递归，因而这个问题不是采用最长匹配和规则顺序问题能够解决的，只能做这个规定。

* + - * **改写C代码中不同token类型的表示以消除左递归**

对于token的规则，标准中都给写完了，没必要画蛇添足。直接照抄就行了，所有的rule规则都可以在“defs/lex\_rule.c”当中查看。

但是，抄也有抄的问题所在，举个例子：

|  |
| --- |
| **identifier:**  **identifier-nondigit**  **identifier identifier-nondigit**  **identifier digit** |

这是我直接抄过来的，如果仅仅看他的第二条规则，显然就是一个左递归的文法。

好在，在C标准中，这样的递归文法永远都不是第一条文法规则，因此，至少不会有含义方面的问题，可以通过消除左递归等等的方式去完成。以上面的为例，改造成EBNF的语法描述就是：

|  |
| --- |
| Identifier: identifier-nondigit ( identifier-nondigit | digit ) \* |

其实就是前面在规则中的那个样例。

**匹配流程**

当预处理器对源文件映射进来的字节流（其实c中没这概念吧）进行处理之后，得到了源字符集构成的字节流。

对于该字节流，从头开始，按照规则逐条进行规则的匹配，并对于最长的匹配进行选择，如果一条也不符合则报错，如果存在长度相同的匹配则按照规则顺序选择更靠前的。

在匹配的过程中，对四种不同的规则分别进行处理，并使用递归方式进行处理。

随后，字节流中切除掉这部分，用切除后的字节流进行接下来的匹配。

**错误处理**

在mcc的错误处理中包括三个部分

* + 错误发现
  + 错误报告
  + 错误处理

**错误发现机制**

错误发现机制和规则匹配的过程应当是混合在一起的。

首先定义一点，报错和没匹配到是两回事，我这里定义的报错，是前面匹配了一部分，但是后面对不上号，并且后面需要匹配的内容是必须有的。没匹配上是压根从头开始就完全不一样，没对上。

前者，如果发生，那么在所有规则都匹配完成之后，仍然没有对的上的，该出错规则，应当作为代码词法出错的规则来处理，后者，在规则匹配过程中，直接跳过即可，在规则全部都匹配完成之后，匹配长度仍然为零，那么显然只有一种可能性，就是输入了不合法的字符，当做报错处理——但这应该在预处理器阶段就已经结束了，原因可以参考预处理器的多个阶段。

相关内容请参看“lex.c”。

这里面设计两种长度，match长度和error长度。

前者保证大于等于0，后者必然小于等于0。

在lex主函数中，对于match函数，因为只有一个返回类型int，所以error取负号表示错误。

Match长度表示成功匹配到的长度，error长度表示虽然出错了，但是在出错前匹配了多少长度。

如果前者大于0，表示成功匹配到了某条规则——否则必然为0。但是这时候error长度可以为小于0，表示别的规则没匹配上。这时候，无需考虑error长度。

如果前者等于0，则需要考虑error长度。error长度为0，则表示一条规则都没匹配上，这属于前面提到的，不合法字符，如果error长度小于0，说明在某条规则处，可能发生匹配，但是出现了错误。

Error长度供后续错误处理程序参考。

在match的递归设计中，对于不同type的处理如下：

* Link类型，如果子规则类型不是星号和问号，那么如果某个子规则缺失或者子规则报错，那么报错，报错长度分两种情况，子规则缺失，他就是匹配不上而已，那么在该缺失长度之前的长度取负号，作为error长度。如果是子规则报错，那报错之前的长度加上子规则返回的error长度，取负号。

如果子规则的类型是星号，无所谓长度，match长度可以为0，但如果返回长度为负数，则认为该星号子规则出错，匹配长度置为0，但不做报错处理。

如果类型是问号，返回0或者正数均不算错，但是如果返回负数，则认为出错。Error长度为子规则的返回error长度加上之前匹配的长度，取负号。

* 或类型，全都没匹配到，那么只是返回匹配为0，如果子规则里面报错，且其他都没匹配上，那么返回子规则的error长度。
* 星号，如果子规则报错，因为星号的特殊性，只考虑子规则报错之前的长度，不做报错处理
* 问号类型，如果子规则报错，那么直接返回子规则报错长度（因为理论上最多只匹配一次），如果子规则匹配长度为0，那么返回0，匹配上了，则返回匹配长度，如果发生多次匹配，大于1，那么返回总共的匹配长度取负号作为error长度，如果最后一次报错（不会存在中间）则还要加上最后一次报错的长度。

这些内容在lex.c的match函数中都有所体现。

**错误报告**

上面在错误发现机制的时候说过，有两种出错可能，一种是，压根一条规则都没匹配上，也就是非法字符。另一种可能是，拼写错误导致的，匹配了某条规则的一部分，但是出错了。

维护一个记录，如果已经匹配的长度大于0，且发生报错，则记录下该次报错的规则以及发生错误时，已经匹配到的内容长度。如果有多个符合的报错规则，则记录最长的那个。如果扫描完成后，如果有任何一条规则符合，则清空该数据结构，不做任何操作，否则输出那个最长的报错规则。

每个遇到的错误构成另一个vec，在lex阶段完成后，统一查看一下是否存在错误，如果有，则报错。

**错误处理机制**

默认的错误处理策略当然是恐慌模式——教材的经典嘛，但是，总感觉和现代编译器的错误处理有不小差距。

举个例子，在test样例中，给了一个0FF，0开头应该匹配oct八进制数，但是FF则不是八进制数，从而造成错误，采用恐慌模式，很简单，报个错，然后往后移动一格，于是FF就匹配完成了。

虽然这么说吧，还有一大堆没有识别的东西，但是，好歹是能识别的都识别了。

**测试代码及测试说明**

测试代码是一段正常的C代码，目的是测试能否将整个文档按照要求进行切分。因此，应当尽可能的覆盖C代码的所有关键字和类型。甚至于，可以不考虑代码是否符合后续的语法分析器的规则。

另外还应当包含一定的错误测试。

**预处理器的词法分析器和语法分析器**

这里面对于这几个部分的文法描述，以及文法匹配进行一个对照比较。

对于EBNF范式，是可以表达上下文无关文法的，因此，预处理器的词法分析器和语法分析器的文法，都可以是用本文档描述的规则进行表达的，因此，无需为了语法分析器或者预处理器，专门再设计一个语法。

但是在文法匹配上，则存在一定的差别。

在词法分析器中，使用的是递归方式进行的匹配。理论上而言，既然EBNF文法可以用递归方式进行匹配，并且词法分析器上面也成功应用了，那么语法分析器，也可以用同样的方式进行递归匹配。

不过，如果能够将词法分析器和语法分析器的语法树画出来，就会发现很大的不同。确切的说，词法分析器的树那是一个中间有大量树，但是树深度都相对小的森林，并且即使存在环路，环路深度也不多。而语法分析器的语法树则不同，总共就一棵树，树深度很深，并且环路众多，一旦走入环路，某个语法的语法树深度更为恐怖，并且，众多节点均为空，缺少实际的终结符，如果没有错误还好，一旦有错误，这个回溯就很烦了。

因此，在词法分析器中，可以使用简单的递归去寻找所有的token，但是语法分析器需要另外考虑。

* **后记**

在整个词法分析器的设计当中，我相信有很多更好，更为成熟的做法——但是我并不想依赖已经成熟的flex等自动词法分析器，因此自己设计了一个。

这篇doc文档，从开始敲代码，到结束，中间的代码基础数据结构和设计多次更改，本篇文档更是数易其稿甚至重构，肯定存在描述前后不连贯甚至矛盾的地方，对此我表示抱歉。如果发现错误并且有意愿修订这篇描述，我非常欢迎。