

语法分析器设计分析报告

1850845 虞枫毅

1852839 李培然

1 需求分析

1.1 程序任务输入范围

本程序用于针对符合语法规则的类 C 语言段进行语法分析，并辨别此段输入是否符合之前确定好的语义规则，因此，此程序针对的输入为一段属于类 C 语言的代码段，或一句符合类 C 语言语法规则的代码语句。

1.2 输出形式

本程序对符合程序要求输入的代码段或代码文件进行语法分析，可选择将分析过程重定向至文件，分析过程和语法分析树都将被输出至文件，使用一句简单的类 C 语言代码作为示例，基础输出文件情况如下（需要使用强大一点的文字编辑器打开避免自动换行破坏此树）：

```

61 ({actionState.Reduce: 1>, 149),
62 Token(type="'',", value="'',", index=10)
63 ({actionState.Reduce: 1>, 147),
64 Token(type="'',", value="'',", index=10)
65 ({actionState.Reduce: 1>, 139),
66 ({actionState.Reduce: 1>, 145),
67 Token(type="'',", value="'',", index=10)
68 ({actionState.Reduce: 1>, 138),
69 ({actionState.Reduce: 1>, 143),
70 Token(type="'',", value="'',", index=10)
71 ({actionState.Reduce: 1>, 140),
72 ({actionState.Reduce: 1>, 140),
73 Token(type="'',", value="'',", index=10)
74 ({actionState.Reduce: 1>, 139),
75 ({actionState.Reduce: 1>, 147),
76 Token(type="'',", value="'',", index=10)
77 ({actionState.Reduce: 1>, 136),
78 ({actionState.Reduce: 1>, 90),
79 Token(type="'',", value="'',", index=10)
80 ({actionState.Reduce: 1>, 138),
81 ({actionState.Reduce: 1>, 48),
82 Token(type="'',", value="'',", index=10)
83 ({actionState.Reduce: 1>, 171),
84 ({actionState.Reduce: 1>, 45),
85 Token(type="'',", value="'',", index=10)
86 ({actionState.Reduce: 1>, 170),
87 ({actionState.Reduce: 1>, 173),
88 Token(type="'G',", value="'G',", index=10)
89 ({actionState.Reduce: 1>, 173),
90 ({actionState.Reduce: 1>, 9),
91 Token(type="'G',", value="'G',", index=10)
92 ({actionState.Reduce: 1>, 4),
93 ({actionState.Reduce: 1>, 10),
94 Token(type="'G',", value="'G',", index=10)
95 ({actionState.Reduce: 1>, 1),
96 ({actionState.Reduce: 1>, 1),
97 Token(type="'G',", value="'G',", index=10)
98 ({actionState.Reduce: 1>, 0),
99 ({actionState.Reduce: 1>, 0),
100 SUCCESSFUL
101 translation_unit      external_decl      decl
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
```

Token 元组中的数据为当前词法分析器所返回的单词，里面包括单词种类，单词值，单词所在位置的索引，Token 下面的 list 列表为当前状态栈内的情况，下面的元组，前一个元素为 Action 表中此时应该执行的动作，如果下一步为移进，则右侧的数字表示下一个该被压进状态栈的状态，如果下一步为归约，则右侧数字代表归约时使用生成式在生成式列表中的索引。

最下方 SUCCESSFUL 之后的值，为语法分析成功之后输出的语法分析树，此语法分析树是逆时针旋转 90 度之后输出的树，我们正在寻找更好的表现方式来表现语法分析树。

1.3 程序功能

本程序是一个类 C 语言的语法分析器，其中包含词法分析程序和语法分析程序，

其中词法分析程序被语法分析程序调用。

本程序可将输入程序符合条件的文法转为 LR(1) 分析表，并可将其保存以便于下一次使用，或者加载符合程序要求的 LR(1) 文法分析表使用文法分析器进行分析。此程序中包含的语法分析程序为 LR(1) 分析程序，其调用词法分析器对源文件进行初步处理并提取词元素，之后根据程序自动生成或加载的 LR(1) 分析表对文法进行分析，分析的同时可输出分析过程，在分析成功后可以 PDF 或 ANSI 字符的形式输出源文件的语法分析树。

1.4 测试数据

1.4.1 正确数据：C 语言实现的二分查找

```
int main()
{
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements");
    scanf("%d", &n);
    printf("Enter %d integers", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter value to find");
    scanf("%d", &search);
    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    while (first <= last) {
        if (array[middle] < search)
            first = middle + 1;
        else if (array[middle] == search) {
            printf("%d found at location %d.", search,
middle+1);
            break;
        }
        else
            last = middle - 1;
        middle = (first + last)/2;
    }
    if (first > last)
        printf("Not found! %d isn't present in the list.",
search);
    return 0;
}
```

1.4.2 错误数据：非法定义

```
int main()
{
    int 334a;
    int 33qraf;
    void www;
```

2 概要设计

2.1 任务分解

一个 LR(1) 语法分析器主要由两部分构成，词法分析器和语法分析器，而要想使用语法分析器对源文件进行分析，则需要依赖一张由 Action 和 GOTO 表组成的 LR 分析表，本程序的设计要求我们完成自动生成此分析表的功能，因此整个任务最终被分为了三部分：

- (1) 词法分析器
- (2) 语法分析器
- (3) LR 分析表生成器

2.2 数据类型的定义

2.2.1 词法分析器类

词法分析器类是文法分析程序的重要组成部分，此类负责从文件中读取数据并封装成格式化的 Token，交由语法分析器进行分析。

此类包含以下属性：

属性名及其类型	作用及功能
Data:string	保存从文件中读取的数据
REs:List[Word(type, RE)]	保存正则表达式列表以及正则表达式所对应的词类型，例如 ID
Index:int	保存当前词的位置索引便于输出报错信息和正则匹配
SizeOfFile:int	保存文件大小，用于判断匹配停止条件

包含以下方法：

方法名	作用及功能
__init__(path)	类构造函数，接受一个 string 类型的路径输入，对文件进行读取并初步处理，初始化对象。
Scan()	使用正则表达式对文件内容进行匹配，返回匹配到的最长串及其类型
Next()	此方法为一个生成器，调用此对象可生成一个迭代器，每次返回一个 Token，其中调用 Scan() 取得其返回值做进一步处理

除这些属性和方法外，此类中还使用了两个简单定义的命名元组 Token 和 Word，Token 用于封装一个固定规格的返回值，其中包含三个值，type，用于指定此词的类

型，例如 ID，value 用于指定整个词的值，例如 “main”，index 用于指定当前词的位置索引。Word 用于帮助正则表达式确定匹配的类型，它只包含两个类型的值。Type 为当前正则表达式对应的词类型，RE 为当前匹配的正则表达式。

2.2.2 语法分析器类

语法分析器类是语法分析器的核心，此类负责根据从词法分析器送来的数据和 LR(1)分析表，对源文件输入的数据进行文法分析，并生成语法分析树。
此类包含以下属性：

属性名及其类型	作用及功能
S:Scanner	词法分析器实例
Token:List(Token)	保存从词法分析器实例传过来的 Token 队列
Status:List(int)	当前状态栈
Symboi:List(Node(type,Children))	当前符号栈

此类中也使用了一个简单定义的命名元组 Node，此元组是为了生成语法树而存在的，当 LR 分析表指示动作为归约时，Token 将被归约，形成一个新的 Node，原来的 Token 成为这个 Node 的一个 Children，被归约后的类型成为了这个 Node 的 type，如果之后这个 Node 也被归约了，那么这个 Node 将成为其他 Node 的 Children，按照这种规律，如果最终分析成功，那么符号栈将仅剩下一个节点，并且这个节点就是语法生成树的根节点，根据这样的设计，我们可以轻松地在分析完文件内容之后生成一棵语法生成树。

此类包含以下属性：

方法	作用及功能
__init__()	类构造函数，接受一个 string 类型的路径输入，完成对对象的初始化
parser(extendTable,actionTable,gotoTable)	核心函数，使用分析表，利用状态栈符号栈等对语法进行分析
showParserTree()	将语法分析树展现出来（打印）

2.2.3 LR 分析表构造器

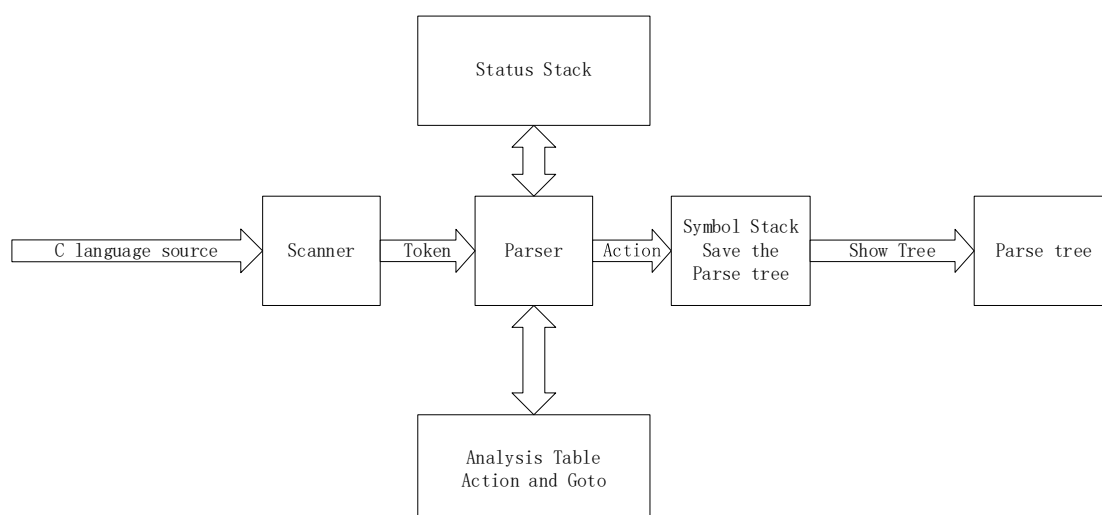
LR 分析表因为函数繁多，且联系不密切，因此并没有封装成为一个类，但放在了一个文件中，因此我们来查看这些函数的作用：

函数名	作用及功能
Nullable_Set(Grammar_Table)	求解 Nullable 集
Terminal_Set(Grammar_Table):	求解 Terminal 集
First_Closure(terms,FirstS,TerminalS,	求解单独一个式子的 First 集

NullS)	
First_Set(Grammar_Table, Terminal, NullS)	求解整个产生式的 First 集
Follow_Set(Grammar_Table, FirstS, Terminal, NullS)	求解整个产生式的 Follow 集
Nullable_Closure(terms, NullS)	求解单独一个式子的 Nullable 集
Calc_Closure(items, Extended_Table, FirstS, Terminals, NullS)	求出一个项集的闭包
LR1_Table(Extended_Table, FirstS, Terminals, NullS, Start_Symbol)	求出 LR 分析表

3 详细设计

3.1 顶层总体设计及函数调用关系



3.2 模块详细设计

本次的语法分析器主要分为了三个模块进行实现，有词法分析器，语法分析器和 LR 分析表生成器三个部分。以下将对此三个模块的设计分别进行描述。

3.2.1 词法分析器

词法分析器的功能主要是读取源文件，对源文件进行初步处理，并按照词法对其进行拆分，便于语法分析器进行语法分析。

词法分析器最主要的作用就是识别当前输入的字符串，识别其类型后发送给语法分析器供其进行语法分析，我使用了正则表达式匹配来完成此项功能，一方面，词法分析的实质即使用多个 DFA 进行匹配，找到最终能到达接受状态的一个 DFA，并将该 DFA 对应的词类型返回，正则表达式可看作是 DFA 的一种表达方式，并且 Python 对正则表达式提供了很好的支持，我们只需提前构造好正则表达式，并将所需进行匹配的正则表达式放进一个数组中，依次进行匹配，找到匹配长度最长的一个，此时被匹配

的词即可被分类为当前正则表达式所对应的类型。正则表达式的使用不仅可以帮助识别词的种类，也可以帮助我们进行错误识别、空白跳过和注释处理。在词法分析器类的 `Scan()` 方法中实现了上述算法，但只有此方法对于词法分析器还是不够的，上述方法仅能做出初步匹配，其中还包含了注释、空白等无用信息，因此我们需要用另一个函数调用它，对它的匹配值进行进一步的处理，并作为可供调用的接口供语法分析器使用，使用的正则表达式可见 `KeyWord.py` 文件。

本来认为，为便于语法分析器的调用，将词法分析器接口作为一个迭代器使用很合适，因为这个场景非常适合迭代器来发挥作用，每当语法分析器需要下一个词时，它调用 `next()` 方法即可轻松获得一个值，而词法分析器也根据需求按次执行，这样的实现方法不仅使得程序执行效率提高，也在一定程度上节省了很多空间，因此词法分析器类中的 `next()` 方法变成了一个生成器，此方法可以帮助进一步处理 `Scan()` 方法识别的词类型，返回符合规范的 `Token`，并且使用一个循环，使其成为一个源源不断识别词类型的生成器，我们只要调用此方法，即可获得一个迭代器，此迭代器的作用就是不断返回识别好词的 `Token`，具体实现如下：

```
def next(self):
    while self.index < self.sizeOfFile:
        res = self.scan()
        if res.type != SPACE:
            if res.type == 'KEY_WORDS':
                yield Token("'" + res.value + "'", res.value, res.index)
            elif res.type == ERR_ID:
                yield Token("ERROR", "ID_INVALID", res.index)
            else:
                yield res
        yield Token("$", "$", self.index)
```

3.2.2 LR(1)分析表生成

大多数用上下文无关文法描述其语法的程序设计语言都有一个 LR(1) 文法，构造 LR(1) 文法构造 LR(0) 类似。

LR(1)文法的意思是从左向右扫描，最右推导，往前多看一个字符。

LR(1)文法也需要构造要给预测分析表，但是 LR(1)的的预测分析表有两部分，分别是 Action 表和 Goto 表。

LR(1)状态是由 LR(1)的项组成的集合，并且存在着合并该超前符号的 LR(1)的 Closure 操作和 Goto 操作。

Closure 操作：

- (1) 假定 I 是一个项目集， I 的任何项目都属于 $CLOSURE(I)$ 。

(2) 若有项目 $A \rightarrow \alpha \cdot B\beta$, a 属于 $CLOSURE(I)$, $B \rightarrow \gamma$ 是文法中的产生式, $\beta \in V^*$, $b \in FIRST(\beta a)$, 则 $B \rightarrow \cdot \gamma, b$ 也属于 $CLOSURE(I)$ 中。

(3) 重复 b) 直到 $CLOSURE(I)$ 不再增大为止。

$Closure(I) =$

repeat

for I 中的任意项 $(A \rightarrow \alpha.X\beta, z)$

for 任意产生式 $X \rightarrow \gamma$

for 任意 ω 在 $First(\beta z)$ 中,

$I \leftarrow I \cup X \rightarrow \cdot \gamma, \omega$

until I 没有改变

Goto 操作:

$Goto(I, X) = Closure(J)$

$J \leftarrow \{$

for I 中的任意项 $(A \rightarrow \alpha.X\beta, z)$

$J \leftarrow (A \rightarrow \alpha.X.\beta, z)$

Action 和 Goto 表的构建:

假设已构造出 LR(1) 项目集规范族为: $C = \{I_0, I_1, \dots, I_n\}$, 其中 I_k 为项目集的名字, k 为状态名, 令包含 $S' \rightarrow \cdot S$ 项目的集合 I_k 的下标 k 为分析器的初始状态。

那么分析表的 ACTION 表和 GOTO 表构造步骤为:

- (1) 若项目 $A \rightarrow \alpha \cdot a\beta$ 属于 I_k 且转换函数 $GO(I_k, a) = I_j$, 当 a 为终结符时则置 $ACTION[k, a]$ 为 S_j 。
- (2) 若项目 $A \rightarrow \alpha \cdot$ 属于 I_k , 则对任何终结符 a 和 '#' 号置 $ACTION[k, a]$ 和 $ACTION[k, \#]$ 为 "rj", j 为在文法 G' 中某产生式 $A \rightarrow \alpha$ 的序号。
- (3) 若 $GO(I_k, A) = I_j$, 则置 $GOTO[k, A]$ 为 "j", 其中 A 为非终结符。
- (4) 若项目 $S' \rightarrow S \cdot$ 属于 I_k , 则置 $ACTION[k, \#]$ 为 "acc", 表示接受。
- (5) 凡不能用上述方法填入的分析表的元素, 均应填上 "报错标志"。为了表的清晰我们仅用空白表示错误标志。

3.2.3 语法分析器

语法分析器的实现是整个系统相对较简单的部分, 词法分析器和 LR 分析表已经替它负重前行了, 所以它要做的事仅仅是, 向词法分析器索要下一个词, 按照 action 和 GOTO 表的指令更新自己的符号栈和状态栈, 总体而言非常简单, 不过在我们的语法分析器中包含一个较为特殊的设定, 之前有提到, 我们在我们的符号栈中保存了语法树。这里有个蛮有趣的设计, 我们的符号栈会直接保存词法分析器发来的 Token, 此 Token 包含三个属性, type, value 和 index, 之前也有提到, 在进行归约时, 我们的符号不会出栈, 而是会成为一个 Node 的 Children, 此 Node 将成为新加入符号栈的元

由于时间的关系，我们仅采用了一种简单的树可视化方法，即将树逆时针旋转了 90 度使用 GBK 编码进行输出，而由于整棵语法树较大，所以需要使⤵用一些特殊的编辑器进行查看，例如 Notepad++或 VS code，这部分算法不重要，暂时略过不谈。

4.1 简单正确代码测试

```
int main()
{
    int a = 10;
    int b = 20;
    int c = a + b;
    return 0;
}
```

[illegible]

```
int main()
{
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements");
    scanf("%d",&n);
```

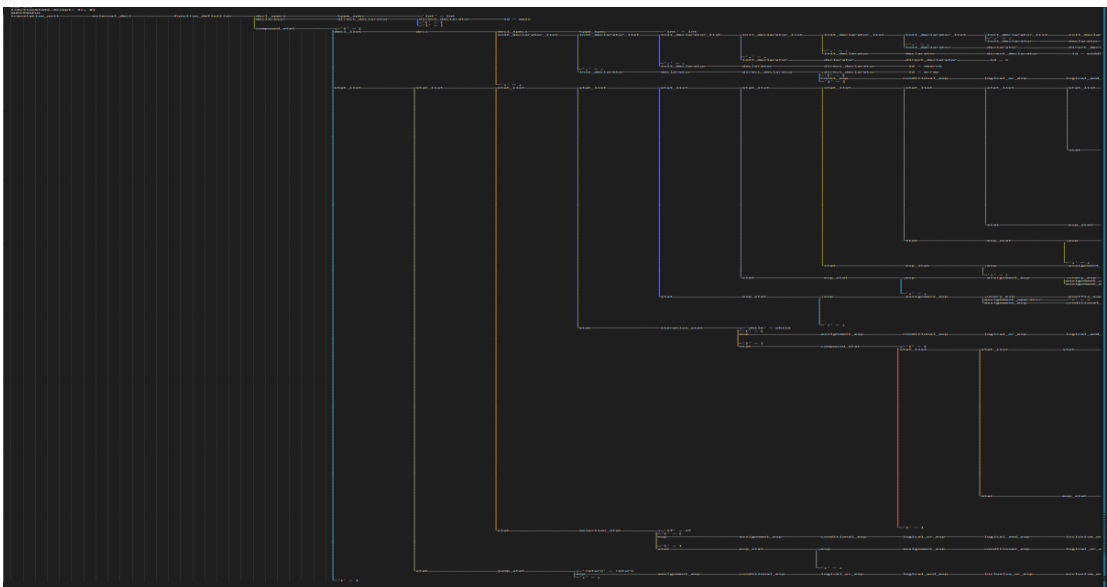


```

printf("Enter %d integers", n);
for (c = 0; c < n; c++)
    scanf("%d",&array[c]);
printf("Enter value to find");
scanf("%d", &search);
first = 0;
last = n - 1;
middle = (first+last)/2;
while (first <= last) {
    if (array[middle] < search)
        first = middle + 1;
    else if (array[middle] == search) {
        printf("%d found at location %d.", search, middle+1);
        break;
    }
    else
        last = middle - 1;
    middle = (first + last)/2;
}
if (first > last)
    printf("Not found! %d isn't present in the list.", search);
return 0;
}

```

测试结果：正确

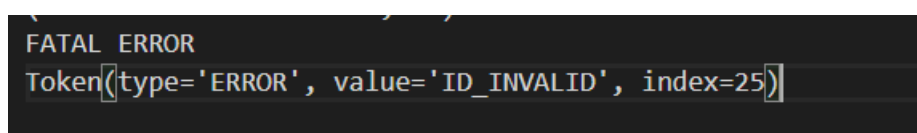


4.3 错误代码测试

源代码：定义不符合规范

```
int main()
{
    int 334a;
    int 33qraf;
    void www;
}
```

测试结果：正确



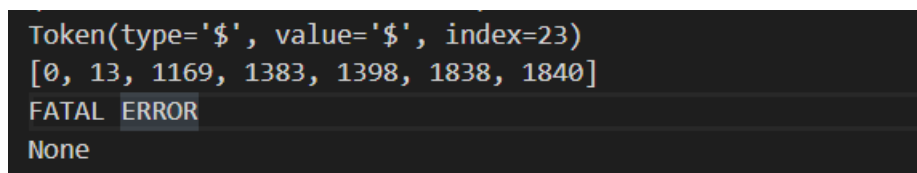
```
FATAL ERROR
Token(type='ERROR', value='ID_INVALID', index=25)
```

4.4 错误代码测试

源代码：无最后的大括号

```
int main()
{
    int a;
```

测试结果：正确



```
Token(type='$', value='$', index=23)
[0, 13, 1169, 1383, 1398, 1838, 1840]
FATAL ERROR
None
```

4.5 模块调用设计问题与思考

如果还有印象的话应该记得，在词法分析器中有个设计，我们将调用词法分析器的接口设计成了一个迭代器，并且期望这个迭代器在语法分析器中发挥其作用，但在实际的应用过程中，我们对这个迭代器的使用并未达到我们设计时的期望，主要是写代码的过程中未考虑清楚归约的情况，而迭代器每个值只能迭代一次，在哪里迭代，迭代器输出的内容该怎样保存，保存在何处，这些都是需要经过设计的，但在写语法分析器时，这些问题并未得到充分考虑，因此，这样一个迭代器最终被我们先变成了List，然后再使用，但我认为词法分析器作为子模块，迭代器的属性与其非常契合，因此需要重新编写的是语法分析器部分的代码，只要我们重复考虑迭代器的特殊性，并作出设计，就可以解决这一问题，只是因为时间的关系，重构计划将在完成课程设计的同时完成。

5 总结与收获

此次的任务是做出一个 LR(1) 语法分析器，在此过程中感觉自己收获颇丰，不仅锻炼了我们分析问题，写代码的能力，同时也从实践层面学习到了关于 LR(1) 文法的很多东西，尤其是 LR(1) 分析表的生成，此算法尤其花费时间，并且很难进行调试，此部分的代码是我们最后完成的部分，一方面是因为其实现较为复杂，另一方面是一个分析表的生成过程非常耗时，很难调试，在整个过程结束之前，我们都不能知晓此过程中是否发生了错误，甚至即使我们验证了几个源程序的正确性，依然不能确定整个分析表是完全正确的。

不过最终经过不断调试，我们终于完成了这个表现尚可的语法分析器，最大的收获就是彻底弄懂了语法分析器的工作原理，自己亲手实践过后，不管是理解还是记忆程度都大大加深了。

附件是源代码以及一些数据文件，源代码使用 Python 3.7 测试通过，具体使用方法请见文件中的 Usage.md 文件。