****PROGRAMMING LANGUAGES

**PROJECT NAME**

Writing a lexical analyzer for BigAdd Language

**STUDENT ID - NAME**

05190000108 – Hamit Can DAŞÇİ

05180000113 – Simge Merve YAŞBAY

**DUE :** 07.06.2021

**Contents**

****

Analysis ..................................................................................................3

Design ....................................................................................................4

Source Code...................................................................................4

Test Data and Screenshot ………….…………………......................................14

4.Limitations..........................................................................................16

**ANALYSIS**

Bizden, BigAdd dili için bir lexical analyzer yazmamız bekleniyor.

Bu lexical analyzer dosya adı alıp, bu dosyanın içinde bulunan sözcükleri türlerine göre ayırıp başka bir dosyaya kaydetmektedir.

Bu türler: keyword, identifier ve operatörlerdir.

****

Dilin başlıca özelliklerine değinecek olursak:

BigAdd dili, sadece int tipi verilerden oluşan bir dildir. Bu dilde atama, toplama , çıkarma işlemleri ve döngü , kod blokları ve yorum satırları bulunmaktadır.

Dil tek bir tane (-) operatörü kullanırken, (+) operatörüne kapalıdır. Dilimiz gerçek sayıları desteklememektedir.

Dildeki değişkenler sadece harfle başlayabilirken devamında (\_) veya rakam içerebilmektedir. Ayrıca değişkenler büyük ve küçük harflere karşı duyarlıdırlar.

Dilde döngü yapısı bulunur. Kod blokları sayesinde döngü içinde döngü veya döngü içinde kod bloğu bulunabilir.

Dilde yorum satırları bulunur ve lexical analyzer işletilirken bu satırlar görmezden gelinir. Dil iç içe yorum satırları içermemektedir.

Ayrıca dilde “ “ karakterleri içinde yazılmış string sabitleri de bulunmaktadır.

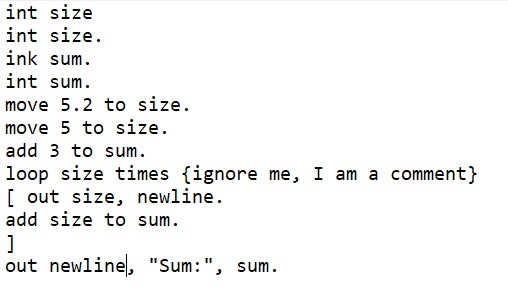
**DESIGN**

**Source Code:**

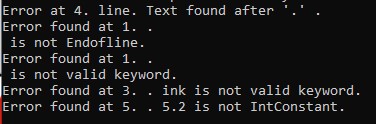
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <stdbool.h>
5. #include <ctype.h>
7. const char ENDOFLINE = '.';
8. const char SEPERATOR[] = "Seperator";
9. const char OPENBLOCK = '[';
10. const char CLOSEBLOCK = ']';
12. char \*KEYWORDS[11] = {"int", "move", "to", "loop", "times", "out", "newline", "add", "sub", "[", "]"};
14. char IdentifierNameList[100][20] = {"temp"};
15. int identifierOrder = 0;
17. int substring(char \*source, int from, int n, char \*target)
18. {
19. int length, i;
20. for (length = 0; source[length] != '\0'; length++)
21. ;
22. if (from > length)
23. {
24. printf("Starting index is invalid.\n");
25. return 1;
26. }
27. if ((from + n) > length)
28. {
29. n = (length - from);
30. }
31. for (i = 0; i < n; i++)
32. {
33. target[i] = source[from + i];
34. }
35. target[i] = '\0';
36. return 0;
37. }
39. void isValidEnd(char lastChar)
40. {
41. // eger en sonda nokta yoksa hatali statement
42. if (lastChar != ENDOFLINE)
43. {
44. printf("%s", "satir '.' ile bitmiyor.");
45. }
46. }
48. bool isKeyword(char \*kelime)
49. {
50. bool flag = false;
51. for (int i = 0; i < 11; i++)
52. {
53. if (strcmp(kelime, KEYWORDS[i]) == 0)
54. {
55. flag = true;
56. }
57. }
58. return flag;
59. }
61. bool isEndofline(char karakter)
62. {
63. return karakter == ENDOFLINE ? true : false;
64. }
66. bool isSeperator(char \*karakter)
67. {
68. return strcmp(karakter, SEPERATOR) == 0 ? true : false;
69. }
71. bool isStringConstant(char \*karakter)
72. {
73. return (karakter[0] == '"' && karakter[strlen(karakter) - 1] == '"') ? true : false;
74. }
76. //tam dogru degil,
77. bool isIntConstant(char \*karakter)
78. {
79. /\*
80. int temp\_int = \*karakter;
81. return isdigit(temp\_int) != 0 ? true : false;
82. \*/
84. int len = strlen(karakter);
86. if (len > 100)
87. {
88. return false;
89. }
91. for (int i = 0; i < len; i++)
92. {
93. if (i > 0)
94. {
95. if (karakter[i] == '-' || len == 1)
96. {
97. return false;
98. }
99. }
100. if (!(isdigit(karakter[i]) || karakter[i] == '-'))
101. {
102. return false;
103. }
104. }
106. return true;
107. }
109. bool isInIdentifierList(char \*kelime)
110. {
111. bool flag = false;
112. for (int i = 0; i < strlen(\*IdentifierNameList); i++)
113. {
114. if (strcmp(kelime, IdentifierNameList[i]) == 0)
115. {
116. flag = true;
117. }
118. }
119. return flag;
120. }
122. bool isIdentifier(char \*karakter, char \*lastToken)
123. {
124. bool isInList = isInIdentifierList(karakter);
125. int first\_letter = karakter[0];
126. bool isValid = ((strcmp(lastToken, "int") == 0 || strcmp(lastToken, "to") == 0 || strcmp(lastToken, "from") == 0 || strcmp(lastToken, "loop") == 0 || strcmp(lastToken, "add") == 0 || strcmp(lastToken, "out") == 0 || strcmp(lastToken, ",") == 0) && (strlen(karakter) <= 20) && (isalpha(first\_letter) != 0) && (isKeyword(karakter) == false)) ? true : false;
127. if (isInList)
128. {
129. return true;
130. }
131. else if (isValid)
132. {
133. strcpy(IdentifierNameList[identifierOrder], karakter);
134. identifierOrder++;
135. return true;
136. }
138. else
139. {
140. return false;
141. }
142. }
144. //alt satira sarkan yorum satirlari sikintili
145. void deleteComments(char \*satir)
146. {
147. int index1 = strcspn(satir, "{");
148. if (index1 + 2 > strlen(satir))
149. {
150. return;
151. }
152. else
153. {
154. int index2 = strcspn(satir, "}");
155. if (index2 + 2 > strlen(satir))
156. {
157. return;
158. }
159. else
160. {
162. for (int i = index1; i <= index2; i++)
163. {
164. satir[i] = '\0';
165. }
166. }
167. }
168. }
170. void splitDot(char \*satir)
171. {
172. static int satir\_no = 1;
173. int dotIndex = strcspn(satir, ".");
174. char \*p = strchr(satir, '.');
176. if (dotIndex == strlen(satir) - 2 || dotIndex == strlen(satir) - 1)
177. {
178. satir[dotIndex] = '\0';
179. strcat(satir, " .");
180. }
181. else
182. {
183. if (p == NULL)
184. {
185. return;
186. }
187. printf("Error at %i. line. Text found after \'.\' .", satir\_no);
188. }
189. satir\_no++;
190. }
192. void splitComma(char \*satir)
193. {
194. int commaIndex = strcspn(satir, ",");
195. while (commaIndex < strlen(satir))
196. {
197. char first[commaIndex];
198. char second[strlen(satir) - commaIndex];
199. substring(satir, 0, commaIndex, first);
200. substring(satir, commaIndex + 1, strlen(satir) - 1, second);
201. strcat(first, " Seperator");
202. strcat(first, second);
203. strcpy(satir, first);
204. commaIndex = strcspn(satir, ",");
205. }
206. }
208. int main(int argc, char \*argv[]) //icteki seyler cmd de parametre vermeye yariyor
209. {
211. char \*SourcefilePath = argv[1];
213. char satir[150];
214. int satir\_sayisi = 0;
215. char satirlar[150][150];
217. // okunacak dosya
218. FILE \*sourceFile = fopen(SourcefilePath, "r");
219. if (sourceFile == NULL)
220. {
221. printf("%s", "File not found. Please check the file.");
222. }
224. // yazilacak dosya
225. FILE \*destFile = fopen("myscript.lx", "w");
227. // satirlari tek tek alan dongu
228. while (!feof(sourceFile))
229. {
230. fgets(satir, 150, sourceFile);
231. deleteComments(satir); //yorum bulunuyorsa satirda siliyor
232. splitDot(satir); // sondaki yapisik noktalari da token almasi icin noktanin oncesine bosluk ekliyor
233. splitComma(satir); // seperator olarak tanimasi icin virgulleri ayiriyor
234. strcpy(satirlar[satir\_sayisi], satir);
235. satir\_sayisi++;
236. }
238. int anlik\_satir = 0;
239. char lastToken[] = "";
241. // satirlari kelimelerine ayiriyor
242. while (anlik\_satir <= satir\_sayisi)
243. {
244. const char ayirici[] = " ";
246. char \*token = strtok(satirlar[anlik\_satir], ayirici);
248. while (token != NULL)
249. {
251. //int state kontrolu
253. if (strcmp(token, "int") == 0)
254. {
255. fprintf(destFile, "Keyword %s\n", token);
256. strcpy(lastToken, token);
257. token = strtok(NULL, ayirici);
259. if (isIdentifier(token, lastToken))
260. {
261. fprintf(destFile, "%s %s\n", "Identifier", token);
262. strcpy(lastToken, token);
263. token = strtok(NULL, ayirici);
264. }
265. else
266. {
267. printf("Error found at %i. . %s is not Identifier.\n", anlik\_satir + 1, token);
268. break;
269. }
271. if (isEndofline(\*token))
272. {
273. fprintf(destFile, "%s\n", "Endofline");
274. break;
275. }
276. else
277. {
278. printf("Error found at %i. . %s is not Endofline.\n", anlik\_satir + 1, token);
279. }
280. }
282. //move state kontrolu
284. if (strcmp(token, "move") == 0)
285. {
286. fprintf(destFile, "Keyword %s\n", token);
287. strcpy(lastToken, token);
288. token = strtok(NULL, ayirici);
290. if (isIntConstant(token) || isIdentifier(token, lastToken))
291. {
292. if (isIntConstant(token))
293. {
294. fprintf(destFile, "IntConstant %s\n", token);
295. strcpy(lastToken, token);
296. token = strtok(NULL, ayirici);
297. }
298. else
299. {
300. fprintf(destFile, "Identifier %s\n", token);
301. strcpy(lastToken, token);
302. token = strtok(NULL, ayirici);
303. }
304. }
305. else
306. {
307. printf("Error found at %i. . %s is not IntConstant.\n", anlik\_satir + 1, token);
308. break;
309. }
311. if (strcmp(token, "to") == 0)
312. {
313. fprintf(destFile, "Keyword %s\n", token);
314. strcpy(lastToken, token);
315. token = strtok(NULL, ayirici);
316. }
317. else
318. {
319. printf("Error found at %i. . %s is not equal to 'to' keyword.\n", anlik\_satir + 1, token);
320. break;
321. }
323. if (isIdentifier(token, lastToken))
324. {
325. fprintf(destFile, "Identifier %s\n", token);
326. strcpy(lastToken, token);
327. token = strtok(NULL, ayirici);
328. }
329. else
330. {
331. printf("Error found at %i. . %s is not Identifier.\n", anlik\_satir + 1, token);
332. break;
333. }
334. if (isEndofline(\*token))
335. {
336. fprintf(destFile, "%s\n", "Endofline");
337. break;
338. }
339. else
340. {
341. printf("Error found at %i. . %s is not Endofline.\n", anlik\_satir + 1, token);
342. }
343. }
345. //add state kontrolu
347. if (strcmp(token, "add") == 0)
348. {
349. fprintf(destFile, "Keyword %s\n", token);
350. strcpy(lastToken, token);
351. token = strtok(NULL, ayirici);
353. if (isIntConstant(token) || isIdentifier(token, lastToken))
354. {
355. if (isIntConstant(token))
356. {
357. fprintf(destFile, "IntConstant %s\n", token);
358. strcpy(lastToken, token);
359. token = strtok(NULL, ayirici);
360. }
361. else
362. {
363. fprintf(destFile, "Identifier %s\n", token);
364. strcpy(lastToken, token);
365. token = strtok(NULL, ayirici);
366. }
367. }
368. else
369. {
370. printf("Error found at %i. . %s is not IntConstant.\n", anlik\_satir + 1, token);
371. break;
372. }
374. if (strcmp(token, "to") == 0)
375. {
376. fprintf(destFile, "Keyword %s\n", token);
377. strcpy(lastToken, token);
378. token = strtok(NULL, ayirici);
379. }
380. else
381. {
382. printf("Error found at %i. . %s is not equal to 'to' keyword.\n", anlik\_satir + 1, token);
383. break;
384. }
386. if (isIdentifier(token, lastToken))
387. {
388. fprintf(destFile, "Identifier %s\n", token);
389. strcpy(lastToken, token);
390. token = strtok(NULL, ayirici);
391. }
392. else
393. {
394. printf("Error found at %i. . %s is not Identifier.\n", anlik\_satir + 1, token);
395. break;
396. }
397. if (isEndofline(\*token))
398. {
399. fprintf(destFile, "%s\n", "Endofline");
400. break;
401. }
402. else
403. {
404. printf("Error found at %i. . %s is not Endofline.\n", anlik\_satir + 1, token);
405. }
406. }
408. //sub state kontrolu
410. if (strcmp(token, "sub") == 0)
411. {
412. fprintf(destFile, "Keyword %s\n", "sub");
413. strcpy(lastToken, token);
414. token = strtok(NULL, ayirici);
416. if (isIntConstant(token) || isIdentifier(token, lastToken))
417. {
418. if (isIntConstant(token))
419. {
420. fprintf(destFile, "IntConstant %s\n", token);
421. strcpy(lastToken, token);
422. token = strtok(NULL, ayirici);
423. }
424. else
425. {
426. fprintf(destFile, "Identifier %s\n", token);
427. strcpy(lastToken, token);
428. token = strtok(NULL, ayirici);
429. }
430. }
431. else
432. {
433. printf("Error found at %i. . %s is not IntConstant.\n", anlik\_satir + 1, token);
434. break;
435. }
437. if (strcmp(token, "from") == 0)
438. {
439. fprintf(destFile, "Keyword %s\n", "from");
440. strcpy(lastToken, token);
441. token = strtok(NULL, ayirici);
442. }
443. else
444. {
445. printf("Error found at %i. . %s is not equal to 'from' keyword.\n", anlik\_satir + 1, token);
446. break;
447. }
449. if (isIdentifier(token, lastToken))
450. {
451. fprintf(destFile, "Identifier %s\n", token);
452. strcpy(lastToken, token);
453. token = strtok(NULL, ayirici);
454. }
455. else
456. {
457. printf("Error found at %i. . %s is not Identifier.\n", anlik\_satir + 1, token);
458. break;
459. }
460. if (isEndofline(\*token))
461. {
462. fprintf(destFile, "%s\n", "Endofline");
463. break;
464. }
465. else
466. {
467. printf("Error found at %i. . %s is not Endofline.\n", anlik\_satir + 1, token);
468. }
469. }
471. //out state kontrolu
473. if (strcmp(token, "out") == 0)
474. {
476. fprintf(destFile, "Keyword %s\n", "out");
477. strcpy(lastToken, token);
478. token = strtok(NULL, ayirici);
480. while (strcmp(token, ".") != 0)
481. {
483. if (isIntConstant(token) || isIdentifier(token, lastToken))
484. {
485. if (isIntConstant(token))
486. {
487. fprintf(destFile, "IntConstant %s\n", token);
488. strcpy(lastToken, token);
489. token = strtok(NULL, ayirici);
490. }
491. else
492. {
493. fprintf(destFile, "Identifier %s\n", token);
494. strcpy(lastToken, token);
495. token = strtok(NULL, ayirici);
496. }
497. }
499. else if (isStringConstant(token))
500. {
501. fprintf(destFile, "StringConstant %s\n", token);
502. strcpy(lastToken, token);
503. token = strtok(NULL, ayirici);
504. }
505. else if (strcmp(token, "newline") == 0)
506. {
507. fprintf(destFile, "Keyword %s\n", token);
508. strcpy(lastToken, token);
509. token = strtok(NULL, ayirici);
510. }
511. else if (isSeperator(token))
512. {
513. fprintf(destFile, "Seperator\n");
514. strcpy(lastToken, token);
515. token = strtok(NULL, ayirici);
516. }
518. else
519. {
520. printf("Error found at %i. . %s is against the grammer of Out State.\n", anlik\_satir + 1, token);
521. break;
522. }
523. }
525. if (isEndofline(\*token))
526. {
527. fprintf(destFile, "%s\n", "Endofline");
528. break;
529. }
530. else
531. {
532. printf("Error found at %i. . %s is not Endofline.\n", anlik\_satir + 1, token);
533. break;
534. }
535. }
537. //loop state kontrolu
539. if (strcmp(token, "loop") == 0)
540. {
542. fprintf(destFile, "Keyword %s\n", "loop");
543. strcpy(lastToken, token);
544. token = strtok(NULL, ayirici);
546. if (isIntConstant(token) || isIdentifier(token, lastToken))
547. {
548. if (isIntConstant(token))
549. {
550. fprintf(destFile, "IntConstant %s\n", token);
551. strcpy(lastToken, token);
552. token = strtok(NULL, ayirici);
553. }
554. else
555. {
556. fprintf(destFile, "Identifier %s\n", token);
557. strcpy(lastToken, token);
558. token = strtok(NULL, ayirici);
559. }
560. }
561. else
562. {
563. printf("Error found at %i. . %s is against the State of Loop.\n", anlik\_satir + 1, token);
564. break;
565. }
567. if (strcmp(token, "times") == 0 || strcmp(token, "times\n") == 0)
568. {
569. fprintf(destFile, "Keyword %s\n", "times");
570. strcpy(lastToken, token);
571. token = strtok(NULL, ayirici);
572. }
573. else
574. {
575. printf("Error found at %i. . %s is not the 'times' keyword.\n", anlik\_satir + 1, token);
576. break;
577. }
578. break;
579. }
581. //OpenBlock
582. if (strcmp(token, "[\n") == 0 || strcmp(token, "[") == 0)
583. {
584. bool isFound = false;
585. while (isFound != true)
586. {
587. for (int i = anlik\_satir; i < satir\_sayisi; i++)
588. {
589. if (strchr(satirlar[i], ']') != NULL)
590. {
591. isFound = true;
592. break;
593. }
594. }
595. }
596. if (isFound)
597. {
598. fprintf(destFile, "Open Block\n");
599. strcpy(lastToken, token);
600. token = strtok(NULL, ayirici);
601. }
602. break;
603. }
605. //CloseBlock
606. if (strcmp(token, "]") == 0 || strcmp(token, "]\n") == 0)
607. {
608. fprintf(destFile, "CloseBlock\n");
609. strcpy(lastToken, token);
610. token = strtok(NULL, ayirici);
611. }
613. else
614. {
615. //hatali
616. printf("Error found at %i. . %s is not valid keyword.\n", anlik\_satir + 1, token);
617. break;
618. }
619. }
620. anlik\_satir++;
621. }
623. fclose(sourceFile);
624. fclose(destFile);
625. return 0;
626. }

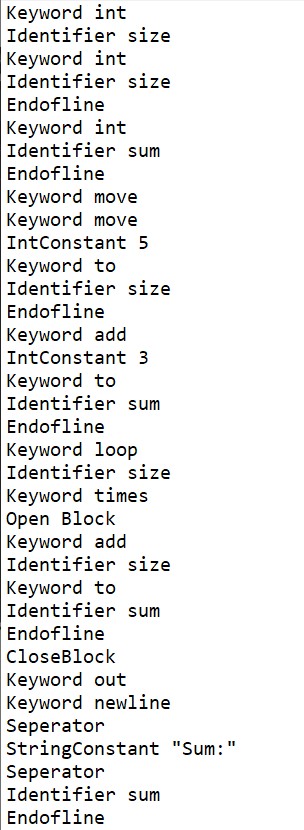
**3. TEST DATAS AND SCREENSHOTS**

**Test Datas:**

****

**Screenshots:**

****

****

**4. LIMITATIONS**

****