

编译原理实验

1. 实验要求

以下列词法表为例：

词法单元类型	词法单元	词素
关键字	IF	if
	THEN	then
	ELSE	else
	END	end
	REPEAT	repeat
	UNTIL	until
	READ	read
	WRITE	write
自定义符	ID	example_id
	NUM	123
运算符	ASSIGN	:=
	RELOP	=
		<>
		>
		<
		>=
		<=
	PLUS	+
	MINUS	-
	TIMES	*
	OVER	/
	LPAREN	(
	RPAREN)
	SEMI	;
空格	DELIMITER	space \t \n \r

对于输入：

```
1 read x; // input x
2 if 0 < x then /* compute when x>0 */
3   fact := 1;
4   repeat
5     fact := fact * x;
6     x := x-1
7   until x = 0;
8   write fact //output fact
9 end
```

它的词法分析输出应该为：

tag	attr
READ	
ID	1
SEMI	;
IF	
NUM	0.000000
RELOP	<
ID	1
THEN	
ID	2
ASSIGN	:=
NUM	1.000000
SEMI	;
REPEAT	
ID	2
ASSIGN	:=
ID	2
TIMES	*
ID	1
SEMI	:
ID	1
ASSIGN	:=
ID	1

tag	attr
MINUS	-
NUM	1.000000
UNTIL	
ID	1
RELOP	=
NUM	0.000000
SEMI	;
WRITE	
ID	2
DOLLAR	\$
Annotations :	
// input x	
/* compute when x > 0 */	
// output fact	

2. 词法分析

```
1 digraph finite_state_machine {
2     rankdir=LR;
3     size="8,5"
4     node [shape = doublecircle]; 1 2 3 4 5 6 7 8 9 10 11;
5     node [shape = circle];
6     0 → 0 [label="space, \\t, \\n, \\r"];
7     0 → 1 [label="letter"];
8     1 → 1 [label="letter | digit"];
9     0 → 2 [label="digit"];
10    2 → 2 [label="digit"];
11    2 → 3 [label="."];
12    3 → 3 [label="digit"];
13    0 → 4 [label="( | ) | ;"];
14    0 → 5 [label=":"];
15    5 → 6 [label="="];
16    0 → 7 [label]="= | + | - | * | /";
17    0 → 8 [label="<"];
18    8 → 9 [label="> | ="];
19    0 → 10 [label=">"];
20    10 → 11 [label="="];
21 }
```

2.1. 状态说明

- 状态0：初态，遇到space等DELIMITER时回到初态，遇到字母进入状态1，遇到数字进入状态2，遇到(|)|;进入状态4，遇到":"进入状态5，遇到=|+|-|*|/进入状态7，遇到<进入状态8，遇到>进入状态10。
- 状态1：终态，对应保留字或标识符。遇到字母或数字保留在状态1。
- 状态2：对应整数。遇到数字保留在状态2，遇到"."进入状态3。
- 状态3：对应小数。遇到数字保留在状态3。
- 状态4：对应分割符。
- 状态5：中间状态，遇到"="进入状态6。
- 状态6：终态，对应ASSIGN。
- 状态7：终态，对应运算符。
- 状态8：中间状态，遇到>或=进入状态9。
- 状态9：终态，对应RELOP。
- 状态10：中间状态，遇到=进入状态11。
- 状态11：终态，对应RELOP。

2.2. 词法分析器代码实现

词法分析器的代码实现分为以下几个部分：

1. **头文件 `tokens.h`**：定义了词法单元类型的枚举 `TokenType` 以及对应的字符串表示 `tokenNames`。
2. **词法分析器 `lexer.c`**：实现了词法分析的具体逻辑，包括读取输入文件、识别词法单元、输出词法单元及其属性。

2.2.1. 头文件 `tokens.h`

```
1  #ifndef TOKENS_H
2  #define TOKENS_H
3
4  typedef enum {
5      IF,
6      THEN,
7      ELSE,
8      END,
9      REPEAT,
10     UNTIL,
11     READ,
12     WRITE,
13     ID,
14     NUM,
15     ASSIGN,
16     RELOP,
17     PLUS,
18     MINUS,
19     TIMES,
20     OVER,
21     LPAREN,
```

```

22     RPAREN,
23     SEMI,
24     DELIMITER,
25     COMMENT,
26     DOLLAR
27 } TokenType;
28
29 const char *tokenNames[] = {
30     "IF", "THEN", "ELSE", "END", "REPEAT", "UNTIL", "READ", "WRITE", "ID",
31     "NUM", "ASSIGN", "RELOP", "PLUS", "MINUS", "TIMES", "OVER", "LPAREN",
32     "RPAREN", "SEMI", "DELIMITER", "COMMENT", "DOLLAR"};
31
32 #endif // TOKENS_H

```

2.2.2. 词法分析器 `lexer.c`

```

1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  typedef enum {
7      READ, ID, SEMI, IF, NUM, RELOP, THEN, ASSIGN, REPEAT, TIMES, UNTIL,
8      WRITE, MINUS, DOLLAR, END, ELSE, COMMENT, PLUS, OVER, LPAREN, RPAREN
9  } TokenType;
10
11 void printToken(TokenType token, const char *attr) {
12     const char *tokenNames[] = {
13         "READ", "ID", "SEMI", "IF", "NUM", "RELOP", "THEN", "ASSIGN",
14         "REPEAT", "TIMES", "UNTIL", "WRITE", "MINUS", "DOLLAR", "END", "ELSE",
15         "COMMENT", "PLUS", "OVER", "LPAREN", "RPAREN"};
16     printf("%s\t%s\n", tokenNames[token], attr);
17 }
18
19 typedef struct {
20     char name[256];
21     int id;
22 } Symbol;
23
24 Symbol symbolTable[256];
25 int symbolCount = 0;
26
27 int lookupSymbol(const char *name) {
28     for (int i = 0; i < symbolCount; i++) {
29         if (strcmp(symbolTable[i].name, name) == 0) {
30             return symbolTable[i].id;
31         }
32     }
33     return -1;
34 }
35
36 int addSymbol(const char *name) {
37     strcpy(symbolTable[symbolCount].name, name);
38     symbolTable[symbolCount].id = symbolCount + 1;
39     return symbolTable[symbolCount++].id;
40 }

```

上面的代码定义了词法单元的类型、打印词法单元的函数以及符号表的相关操作。

```

1 void processFile(const char *filePath) {
2     FILE *file = fopen(filePath, "r");
3     if (!file) {
4         perror("Failed to open file");
5         exit(EXIT_FAILURE);
6     }
7
8     char buffer[256];
9     int bufferIndex = 0;
10    int c;
11    char annotations[1024] = "";
12    int annotationIndex = 0;
13
14    while ((c = fgetc(file)) != EOF) {
15        if (isspace(c)) {
16            continue;
17        }
18
19        if (isalpha(c)) {
20            bufferIndex = 0;
21            while (isalpha(c) || isdigit(c)) {
22                buffer[bufferIndex++] = c;
23                c = fgetc(file);
24            }
25            buffer[bufferIndex] = '\0';
26            ungetc(c, file);
27
28            if (strcmp(buffer, "read") == 0) {
29                printToken(READ, "");
30            } else if (strcmp(buffer, "if") == 0) {
31                printToken(IF, "");
32            } else if (strcmp(buffer, "then") == 0) {
33                printToken(THEN, "");
34            } else if (strcmp(buffer, "repeat") == 0) {
35                printToken(REPEAT, "");
36            } else if (strcmp(buffer, "until") == 0) {
37                printToken(UNTIL, "");
38            } else if (strcmp(buffer, "write") == 0) {
39                printToken(WRITE, "");
40            } else if (strcmp(buffer, "end") == 0) {
41                printToken(END, "");
42            } else if (strcmp(buffer, "else") == 0) {
43                printToken(ELSE, "");
44            } else {
45                int id = lookupSymbol(buffer);
46                if (id == -1) {
47                    id = addSymbol(buffer);
48                }
49                char idStr[10];
50                sprintf(idStr, "%d", id);
51                printToken(ID, idStr);

```

```

52     }
53     } else if (isdigit(c)) {
54         bufferIndex = 0;
55         while (isdigit(c)) {
56             buffer[bufferIndex++] = c;
57             c = fgetc(file);
58         }
59         if (c == '.') {
60             buffer[bufferIndex++] = c;
61             c = fgetc(file);
62             while (isdigit(c)) {
63                 buffer[bufferIndex++] = c;
64                 c = fgetc(file);
65             }
66         }
67         buffer[bufferIndex] = '\0';
68         ungetc(c, file);
69         double num = atof(buffer);
70         sprintf(buffer, "%.6f", num);
71         printToken(NUM, buffer);
72     } else if (c == '/') {
73         c = fgetc(file);
74         if (c == '/') {
75             // single line comment
76             bufferIndex = 0;
77             buffer[bufferIndex++] = '/';
78             buffer[bufferIndex++] = '/';
79             while ((c = fgetc(file)) != '\n' && c != EOF) {
80                 buffer[bufferIndex++] = c;
81             }
82             buffer[bufferIndex] = '\0';
83             strcat(annotations, buffer);
84             strcat(annotations, "\n");
85         } else if (c == '*') {
86             // Multi-line comment
87             bufferIndex = 0;
88             buffer[bufferIndex++] = '/';
89             buffer[bufferIndex++] = '*';
90             while (1) {
91                 c = fgetc(file);
92                 if (c == EOF) {
93                     break;
94                 }
95                 buffer[bufferIndex++] = c;
96                 if (c == '*') {
97                     c = fgetc(file);
98                     if (c == '/') {
99                         buffer[bufferIndex++] = '/';
100                         break;
101                     } else {
102                         ungetc(c, file);
103                     }
104                 }
105             }
106             buffer[bufferIndex] = '\0';
107             strcat(annotations, buffer);

```

```

108         strcat(annotations, "\n");
109     } else {
110         ungetc(c, file);
111         printToken(OVER, "/");
112     }
113 } else {
114     switch (c) {
115     case ';':
116         printToken(SEMI, ";");
117         break;
118     case ':':
119         if ((c = fgetc(file)) == '=') {
120             printToken(ASSIGN, ":=");
121         } else {
122             ungetc(c, file);
123         }
124         break;
125     case '<':
126         if ((c = fgetc(file)) == '=' || c == '>') {
127             buffer[0] = '<';
128             buffer[1] = c;
129             buffer[2] = '\0';
130             printToken(RELOP, buffer);
131         } else {
132             ungetc(c, file);
133             printToken(RELOP, "<");
134         }
135         break;
136     case '>':
137         if ((c = fgetc(file)) == '=') {
138             printToken(RELOP, ">=");
139         } else {
140             ungetc(c, file);
141             printToken(RELOP, ">");
142         }
143         break;
144     case '=':
145         printToken(RELOP, "=");
146         break;
147     case '+':
148         printToken(PLUS, "+");
149         break;
150     case '-':
151         printToken(MINUS, "-");
152         break;
153     case '*':
154         printToken(TIMES, "*");
155         break;
156     case '(':
157         printToken(LPAREN, "(");
158         break;
159     case ')':
160         printToken(RPAREN, ")");
161         break;
162     default:
163         break;

```



```

164         }
165     }
166 }
167
168 fclose(file);
169
170 // Print annotations
171 printf("Annotations :\\n%s", annotations);
172 }
173
174 int main(int argc, char *argv[]) {
175     if (argc != 2) {
176         fprintf(stderr, "Usage: %s <file_path>\\n", argv[0]);
177         return EXIT_FAILURE;
178     }
179
180     processFile(argv[1]);
181
182     return EXIT_SUCCESS;
183 }

```

上面的代码实现了词法分析器的主要逻辑，包括处理关键字、标识符、数字、运算符、分隔符和注释等。

2.3. 心得体会

在实现词法分析器的过程中，我们遇到了多个挑战，特别是在处理多行注释时。以下是我们在处理多行注释时的几个版本的代码以及遇到的问题：

2.3.1. 初始版本

在初始版本中，我们简单地处理多行注释，假设注释的结束符 `*/` 总是会正确出现。然而，这种假设在实际情况下并不总是成立。

```

1  if (c == '*') {
2      while ((c = fgetc(file)) != '/' && c != EOF) {
3          buffer[bufferIndex++] = c;
4      }
5      buffer[bufferIndex++] = '/';
6  }

```

2.3.2. 改进版本

在改进版本中，我们增加了对 `EOF` 的检查，以确保在文件结束时能够正确处理未闭合的注释。然而，这个版本仍然存在一个问题：如果多行注释内出现了单个的 `*` 号，其后面不是 `/`，那么会导致这个符号后面的字符丢失。

```

1  if (c == '*') {
2      while (1) {
3          c = fgetc(file);
4          if (c == EOF) {
5              break;
6          }
7          buffer[bufferIndex++] = c;

```

```

8         if (c == '*') {
9             c = fgetc(file);
10            if (c == '/') {
11                buffer[bufferIndex++] = '/';
12                break;
13            }
14        }
15    }
16 }

```

2.3.3. 最终版本

在最终版本中，我们通过使用 `ungetc` 函数来解决上述问题。具体来说，当我们遇到 `*` 号时，会读取下一个字符，如果不是 `/`，则将其放回输入流中。这种方法确保了多行注释内的所有字符都能被正确处理。

```

1  if (c == '*') {
2      while (1) {
3          c = fgetc(file);
4          if (c == EOF) {
5              break;
6          }
7          buffer[bufferIndex++] = c;
8          if (c == '*') {
9              c = fgetc(file);
10             if (c == '/') {
11                 buffer[bufferIndex++] = '/';
12                 break;
13             } else {
14                 ungetc(c, file);
15             }
16         }
17     }
18 }

```

2.3.4. 短路机制的问题

在处理多行注释时，我们还遇到了C语言的短路机制问题。具体来说，只有当当前字符是 `*` 时，才有可能执行 `fgetc`，并且才需要调用 `ungetc`。这种短路机制导致我们在编写代码时需要特别小心，以确保逻辑的正确性。

```

1  if (c == '*') {
2      c = fgetc(file);
3      if (c == '/') {
4          buffer[bufferIndex++] = '/';
5          break;
6      } else {
7          ungetc(c, file);
8      }
9  }

```

通过这些改进和优化，我们最终实现了一个能够正确处理各种输入的词法分析器。这不仅提高了我们的编程技巧，也加深了我们对编译原理的理解。

2.3.5. 编译和运行词法分析器

```
1 gcc lexer.c -o lexer
2 ./lexer input.txt > output.txt
```

运行结果将输出到 `output.txt` 文件中。

3. 语法分析

以下是根据之前定义的词法单元构建的文法：

```
1 <program> ::= <stmt_list>
2
3 <stmt_list> ::= <stmt> ; <stmt_list> | <stmt>
4
5 <stmt> ::= <if_stmt> | <repeat_stmt> | <assign_stmt> | <read_stmt> |
  <write_stmt>
6
7 <if_stmt> ::= IF <exp> THEN <stmt_list> END | IF <exp> THEN <stmt_list> ELSE
  <stmt_list> END
8
9 <repeat_stmt> ::= REPEAT <stmt_list> UNTIL <exp>
10
11 <assign_stmt> ::= ID ASSIGN <exp>
12
13 <read_stmt> ::= READ ID
14
15 <write_stmt> ::= WRITE <exp>
16
17 <exp> ::= <simple_exp> <comparison_op> <simple_exp> | <simple_exp>
18
19 <comparison_op> ::= RELOP
20
21 <simple_exp> ::= <term> <add_op> <simple_exp> | <term>
22
23 <add_op> ::= PLUS | MINUS
24
25 <term> ::= <factor> <mul_op> <term> | <factor>
26
27 <mul_op> ::= TIMES | OVER
28
29 <factor> ::= LPAREN <exp> RPAREN | ID | NUM
```

3.1. 文法处理

在后续的图表中，我们将使用简写符号来表示非终结符和终结符。大写字母代表非终结符，小写字母代表终结符。例如：

- `<program>` 用 `P` 表示
- `<stmt_list>` 用 `SL` 表示
- `<stmt>` 用 `S` 表示
- `<if_stmt>` 用 `I` 表示

- `<repeat_stmt>` 用 `R` 表示
- `<assign_stmt>` 用 `A` 表示
- `<read_stmt>` 用 `RD` 表示
- `<write_stmt>` 用 `W` 表示
- `<exp>` 用 `E` 表示
- `<comparison_op>` 用 `RO` 表示
- `<simple_exp>` 用 `SE` 表示
- `<add_op>` 用 `AO` 表示
- `<term>` 用 `T` 表示
- `<mul_op>` 用 `MO` 表示
- `<factor>` 用 `F` 表示

LL(1) 文法需要先提取左公因子。以下是提取左公因子后的文法：

```

1  P → SL
2
3  SL → S SL'
4
5  SL' → ; SL | ε
6
7  S → I | R | A | RD | W
8
9  I → if E then SL I'
10
11 I' → end | else SL end
12
13 R → repeat SL until E
14
15 A → id := E
16
17 RD → read id
18
19 W → write E
20
21 E → SE E'
22
23 E' → RO SE | ε
24
25 RO → relop
26
27 SE → T SE'
28
29 SE' → AO T SE' | ε
30
31 AO → plus | minus
32
33 T → F T'
34
35 T' → MO F T' | ε
36

```

37

MO → times | over

38

39

F → (E) | id | num

3.2. First 集和 Follow 集

以下是每个非终结符的 First 集和 Follow 集：

非终结符	缩写	文法表达式	First 集	Follow 集
Program	P	$P \rightarrow SL$	if repeat id read write	\$
StmtList	SL	$SL \rightarrow S SL'$	if repeat id read write	\$ end else until
StmtList'	SL'	$SL' \rightarrow ; SL$;	\$ end else until
		$SL' \rightarrow \epsilon$	\$ end else until	
Stmt	S	$S \rightarrow I$	if	; \$ end else until
		$S \rightarrow R$	repeat	
		$S \rightarrow A$	id	
		$S \rightarrow RD$	read	
		$S \rightarrow W$	write	
IfStmt	I	$I \rightarrow \text{if } E \text{ then } SL \text{ } I'$	if	; \$ end else until
IfStmt'	I'	$I' \rightarrow \text{end}$	end	; \$ end else until
		$I' \rightarrow \text{else } SL \text{ end}$	else	

RepeatStmt	R	$R \rightarrow \text{repeat SL until E}$	repeat	; \$ end else until
AssignStmt	A	$A \rightarrow \text{id} := E$	id	; \$ end else until
ReadStmt	RD	$RD \rightarrow \text{read id}$	read	; \$ end else until
WriteStmt	W	$W \rightarrow \text{write E}$	write	; \$ end else until
Exp	E	$E \rightarrow SE E'$	(id num	then ; \$ end else until)
Exp'	E'	$E' \rightarrow RO SE$	relop	then ; \$ end else until)
		$E' \rightarrow \epsilon$	then ; \$ end else until)	
RelOp	RO	$RO \rightarrow \text{relop}$	relop	(id num

SimpleExp	SE	$SE \rightarrow T SE'$	(id num	then ; \$ end else until)
SimpleExp'	SE'	$SE' \rightarrow AO T SE'$	plus minus	then ; \$ end else until)
		$SE' \rightarrow \epsilon$	then ; \$ end else until)	
AddOp	AO	$AO \rightarrow \text{plus}$	plus	(id num
		$AO \rightarrow \text{minus}$	minus	
Term	T	$T \rightarrow F T'$	(id num	plus minus then ; \$ end else until)
Term'	T'	$T' \rightarrow MO F T'$	times over	plus minus then ; \$ end else until)
		$T' \rightarrow \epsilon$	plus minus then ; \$ end else until)	
MulOp	MO	$MO \rightarrow \text{times}$	times	(id num
		$MO \rightarrow \text{over}$	over	

Factor	F	$F \rightarrow (E)$	(times over plus minus then ; \$ end else until)
		$F \rightarrow id$	id	
		$F \rightarrow num$	num	

3.3. LL(1) 分析表

以下是根据之前定义的 LL(1) 文法构建的 LL(1) 分析表：

	if	repeat	id	read	write	;	end	else	until	relop	plus	minus	times	over	()	num	:=	then	\$
P	P→SL	P→SL	P→SL	P→SL	P→SL															
SL	SL → S SL'	SL → S SL'	SL → S SL'	SL → S SL'	SL → S SL'															
SL'						SL'→;SL'	SL'→ε	SL'→ε	SL'→ε											SL'→ε
S	S→I	S→R	S→A	S→RD	S→W															
I	I→if E then SL I'																			
I'							I'→end	I'→else SL end												
R		R→repeat SL until E																		
A			A→id := E																	
RD				RD→read id																
W					W→write E															
E			E → SE E'												E → SE E'		E → SE E'			
E'						E'→ε	E'→ε	E'→ε	E'→ε	E'→RO SE						E'→ε		E'→ε	E'→ε	
RO										RO→relop										
SE			SE→T SE'												SE→T SE'		SE→T SE'			
SE'						SE'→ε	SE'→ε	SE'→ε	SE'→ε		SE'→AO T SE'	SE'→AO T SE'				SE'→ε		SE'→ε	SE'→ε	
AO											AO→plus	AO→minus								
T			T→F T'												T→F T'		T→F T'			
T'						T'→ε	T'→ε	T'→ε	T'→ε		T'→ε	T'→ε	T'→MO F T'	T'→MO F T'		T'→ε		T'→ε	T'→ε	
MO													MO→times	MO→over						
F			F→id												F→(E)		F→num			

3.4. 语法分析器代码实现

在本节中，我们将介绍如何实现一个基于递归下降的语法分析器。该分析器使用 LL(1) 分析表，并且包含调试功能，可以在调试模式下输出进入和离开每个解析函数的调试信息。

3.4.1. 包含必要的头文件和定义全局变量

首先，我们包含必要的头文件，并定义全局变量。`debug` 变量用于控制调试模式，`tokens` 数组用于存储从词法分析器读取的词法单元，`tokenIndex` 和 `currentToken` 分别用于记录词法单元的总数和当前处理的词法单元索引。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "tokens.h"
5
```



```

6 #define MAX_TOKENS 1024
7
8 int debug = 0; // 调试开关, 0 表示关闭, 1 表示开启
9
10 typedef struct {
11     TokenType token;
12     char attr[256];
13 } Token;
14
15 Token tokens[MAX_TOKENS];
16 int tokenIndex = 0;
17 int currentToken = 0;

```

3.4.2. 读取词法分析器输出

`readTokens` 函数从词法分析器的输出文件中读取词法单元, 并将其存储在 `tokens` 数组中。该函数处理每一行, 判断是否包含属性, 并将 `tokenName` 转换为 `TokenType` 枚举值。

```

1 void readTokens(const char *filePath) {
2     FILE *file = fopen(filePath, "r");
3     if (!file) {
4         perror("无法打开词法分析器输出文件");
5         exit(EXIT_FAILURE);
6     }
7
8     char line[512];
9     while (fgets(line, sizeof(line), file)) {
10         line[strcspn(line, "\n")] = 0;
11
12         char tokenName[256];
13         char attr[256];
14         TokenType tokenType;
15
16         char *tabPos = strchr(line, '\t');
17         if (tabPos) {
18             *tabPos = 0;
19             strcpy(tokenName, line);
20             strcpy(attr, tabPos + 1);
21         } else {
22             strcpy(tokenName, line);
23             attr[0] = '\0';
24         }
25
26         if (strcmp(tokenName, "IF") == 0)
27             tokenType = IF;
28         else if (strcmp(tokenName, "THEN") == 0)
29             tokenType = THEN;
30         else if (strcmp(tokenName, "ELSE") == 0)
31             tokenType = ELSE;
32         else if (strcmp(tokenName, "END") == 0)
33             tokenType = END;
34         else if (strcmp(tokenName, "REPEAT") == 0)
35             tokenType = REPEAT;
36         else if (strcmp(tokenName, "UNTIL") == 0)
37             tokenType = UNTIL;

```

```

38     else if (strcmp(tokenName, "READ") == 0)
39         tokenType = READ;
40     else if (strcmp(tokenName, "WRITE") == 0)
41         tokenType = WRITE;
42     else if (strcmp(tokenName, "ID") == 0)
43         tokenType = ID;
44     else if (strcmp(tokenName, "NUM") == 0)
45         tokenType = NUM;
46     else if (strcmp(tokenName, "ASSIGN") == 0)
47         tokenType = ASSIGN;
48     else if (strcmp(tokenName, "RELOP") == 0)
49         tokenType = RELOP;
50     else if (strcmp(tokenName, "PLUS") == 0)
51         tokenType = PLUS;
52     else if (strcmp(tokenName, "MINUS") == 0)
53         tokenType = MINUS;
54     else if (strcmp(tokenName, "TIMES") == 0)
55         tokenType = TIMES;
56     else if (strcmp(tokenName, "OVER") == 0)
57         tokenType = OVER;
58     else if (strcmp(tokenName, "LPAREN") == 0)
59         tokenType = LPAREN;
60     else if (strcmp(tokenName, "RPAREN") == 0)
61         tokenType = RPAREN;
62     else if (strcmp(tokenName, "SEMI") == 0)
63         tokenType = SEMI;
64     else if (strcmp(tokenName, "DOLLAR") == 0)
65         tokenType = DOLLAR;
66     else {
67         printf("未知的词法单元: %s\n", tokenName);
68         exit(EXIT_FAILURE);
69     }
70
71     tokens[tokenIndex].token = tokenType;
72     strcpy(tokens[tokenIndex].attr, attr);
73     tokenIndex++;
74
75     if (tokenType == DOLLAR) {
76         break;
77     }
78 }
79 fclose(file);
80 }

```

3.4.3. 获取和匹配词法单元

`getToken` 函数返回当前词法单元的类型，`match` 函数用于匹配并消耗期望的词法单元。如果匹配成功，`currentToken` 递增；否则，输出语法错误信息并退出程序。

```

1 TokenType getToken() {
2     return tokens[currentToken].token;
3 }
4
5 void match(TokenType expected) {
6     if (getToken() == expected) {

```

```

7         if (debug) {
8             printf("匹配: %s\n", tokenNames[expected]);
9         }
10        currentToken++;
11    } else {
12        printf("语法错误: 期望 %d, 得到 %d\n", expected, getToken());
13        exit(EXIT_FAILURE);
14    }
15 }

```

3.4.4. 递归下降解析函数声明

我们声明了所有递归下降解析函数，这些函数将根据LL(1)分析表和文法规则实现。

```

1 void Program();
2 void StmtList();
3 void StmtListPrime();
4 void Stmt();
5 void IfStmt();
6 void IfStmtPrime();
7 void RepeatStmt();
8 void AssignStmt();
9 void ReadStmt();
10 void WriteStmt();
11 void Exp();
12 void ExpPrime();
13 void RelOp();
14 void SimpleExp();
15 void SimpleExpPrime();
16 void AddOp();
17 void Term();
18 void TermPrime();
19 void MulOp();
20 void Factor();

```

3.4.5. 解析程序和语句列表

`Program` 函数解析程序的起始符号，并调用 `StmtList` 函数解析语句列表。`StmtList` 函数解析一个语句并调用 `StmtListPrime` 解析后续语句。

```

1 void Program() {
2     if (debug)
3         printf("进入: Program\n");
4     if (getToken() == IF || getToken() == REPEAT || getToken() == ID ||
        getToken() == READ || getToken() == WRITE) {
5         StmtList();
6         if (getToken() == DOLLAR) {
7             if (debug)
8                 printf("匹配: DOLLAR\n");
9             printf("语法分析成功! \n");
10        } else {
11            printf("语法错误: 程序末尾缺少 $ 符号\n");
12            exit(EXIT_FAILURE);
13        }

```

```

14     } else {
15         printf("语法错误: 无法识别的起始符号\n");
16         exit(EXIT_FAILURE);
17     }
18     if (debug)
19         printf("离开: Program\n");
20 }
21
22 void StmtList() {
23     if (debug)
24         printf("进入: StmtList\n");
25     Stmt();
26     StmtListPrime();
27     if (debug)
28         printf("离开: StmtList\n");
29 }

```

3.4.6. 解析语句列表的后续部分

`StmtListPrime` 函数解析语句列表的后续部分。如果当前词法单元是分号，则匹配分号并解析下一个语句；否则，解析空产生式。

```

1 void StmtListPrime() {
2     if (debug)
3         printf("进入: StmtListPrime\n");
4     if (getToken() == SEMI) {
5         match(SEMI);
6         Stmt();
7         StmtListPrime();
8     } else if (getToken() == END || getToken() == ELSE || getToken() ==
UNTIL || getToken() == DOLLAR || getToken() == THEN) {
9         // SL' -> ε
10    } else {
11        printf("语法错误: 期望分号或结束符号\n");
12        exit(EXIT_FAILURE);
13    }
14    if (debug)
15        printf("离开: StmtListPrime\n");
16 }

```

3.4.7. 解析语句

`Stmt` 函数根据当前词法单元的类型调用相应的解析函数，如 `IfStmt`、`RepeatStmt`、`AssignStmt`、`ReadStmt` 和 `WriteStmt`。

```

1 void Stmt() {
2     if (debug)
3         printf("进入: Stmt\n");
4     if (getToken() == IF) {
5         IfStmt();
6     } else if (getToken() == REPEAT) {
7         RepeatStmt();
8     } else if (getToken() == ID) {
9         AssignStmt();

```

```

10     } else if (getToken() == READ) {
11         ReadStmt();
12     } else if (getToken() == WRITE) {
13         WriteStmt();
14     } else {
15         printf("语法错误: 无法识别的语句\n");
16         exit(EXIT_FAILURE);
17     }
18     if (debug)
19         printf("离开: Stmt\n");
20 }

```

3.4.8. 解析 if 语句

`IfStmt` 函数解析 if 语句，包括条件表达式、then 部分的语句列表和 else 部分（如果存在）。

```

1 void IfStmt() {
2     if (debug)
3         printf("进入: IfStmt\n");
4     match(IF);
5     Exp();
6     match(THEN);
7     StmtList();
8     IfStmtPrime();
9     if (debug)
10        printf("离开: IfStmt\n");
11 }
12
13 void IfStmtPrime() {
14     if (debug)
15         printf("进入: IfStmtPrime\n");
16     if (getToken() == END) {
17         match(END);
18     } else if (getToken() == ELSE) {
19         match(ELSE);
20         StmtList();
21         match(END);
22     } else {
23         printf("语法错误: 期望 end 或 else\n");
24         exit(EXIT_FAILURE);
25     }
26     if (debug)
27         printf("离开: IfStmtPrime\n");
28 }

```

3.4.9. 解析其他语句

类似地，我们实现了 `RepeatStmt`、`AssignStmt`、`ReadStmt` 和 `WriteStmt` 函数，分别解析 repeat 语句、赋值语句、read 语句和 write 语句。

```

1 void RepeatStmt() {
2     if (debug)
3         printf("进入: RepeatStmt\n");
4     match(REPEAT);

```

```

5     StmtList();
6     match(UNTIL);
7     Exp();
8     if (debug)
9         printf("离开: RepeatStmt\n");
10 }
11
12 void AssignStmt() {
13     if (debug)
14         printf("进入: AssignStmt\n");
15     match(ID);
16     match(ASSIGN);
17     Exp();
18     if (debug)
19         printf("离开: AssignStmt\n");
20 }
21
22 void ReadStmt() {
23     if (debug)
24         printf("进入: ReadStmt\n");
25     match(READ);
26     match(ID);
27     if (debug)
28         printf("离开: ReadStmt\n");
29 }
30
31 void WriteStmt() {
32     if (debug)
33         printf("进入: WriteStmt\n");
34     match(WRITE);
35     Exp();
36     if (debug)
37         printf("离开: WriteStmt\n");
38 }

```

3.4.10. 解析表达式

Exp 函数解析表达式，调用 **SimpleExp** 解析简单表达式，并调用 **ExpPrime** 解析后续部分。

```

1 void Exp() {
2     if (debug)
3         printf("进入: Exp\n");
4     SimpleExp();
5     ExpPrime();
6     if (debug)
7         printf("离开: Exp\n");
8 }
9
10 void ExpPrime() {
11     if (debug)
12         printf("进入: ExpPrime\n");
13     if (getToken() == RELOP) {
14         RelOp();
15         SimpleExp();

```

```

16     } else if (getToken() == THEN || getToken() == SEMI || getToken() == END
    || getToken() == ELSE || getToken() == UNTIL || getToken() == RPAREN ||
    getToken() == DOLLAR) {
17         // E' -> ε
18     } else {
19         printf("语法错误: 无法识别的表达式\n");
20         exit(EXIT_FAILURE);
21     }
22     if (debug)
23         printf("离开: ExpPrime\n");
24 }
25
26 void RelOp() {
27     if (debug)
28         printf("进入: RelOp\n");
29     match(RELOP);
30     if (debug)
31         printf("离开: RelOp\n");
32 }

```

3.4.11. 解析简单表达式和项

SimpleExp 函数解析简单表达式，调用 **Term** 解析项，并调用 **SimpleExpPrime** 解析后续部分。

Term 函数解析项，调用 **Factor** 解析因子，并调用 **TermPrime** 解析后续部分。

```

1 void SimpleExp() {
2     if (debug)
3         printf("进入: SimpleExp\n");
4     Term();
5     SimpleExpPrime();
6     if (debug)
7         printf("离开: SimpleExp\n");
8 }
9
10 void SimpleExpPrime() {
11     if (debug)
12         printf("进入: SimpleExpPrime\n");
13     if (getToken() == PLUS || getToken() == MINUS) {
14         AddOp();
15         Term();
16         SimpleExpPrime();
17     } else if (getToken() == RELOP || getToken() == THEN || getToken() ==
    SEMI || getToken() == END || getToken() == ELSE || getToken() == UNTIL ||
    getToken() == RPAREN || getToken() == DOLLAR) {
18         // SE' -> ε
19     } else {
20         printf("语法错误: 无法识别的简单表达式\n");
21         exit(EXIT_FAILURE);
22     }
23     if (debug)
24         printf("离开: SimpleExpPrime\n");
25 }
26
27 void AddOp() {
28     if (debug)

```

```

29     printf("进入: AddOp\n");
30     if (getToken() == PLUS) {
31         match(PLUS);
32     } else if (getToken() == MINUS) {
33         match(MINUS);
34     } else {
35         printf("语法错误: 期望加法操作符\n");
36         exit(EXIT_FAILURE);
37     }
38     if (debug)
39         printf("离开: AddOp\n");
40 }
41
42 void Term() {
43     if (debug)
44         printf("进入: Term\n");
45     Factor();
46     TermPrime();
47     if (debug)
48         printf("离开: Term\n");
49 }
50
51 void TermPrime() {
52     if (debug)
53         printf("进入: TermPrime\n");
54     if (getToken() == TIMES || getToken() == OVER) {
55         MulOp();
56         Factor();
57         TermPrime();
58     } else if (getToken() == PLUS || getToken() == MINUS || getToken() ==
RELOP || getToken() == THEN || getToken() == SEMI || getToken() == END ||
getToken() == ELSE || getToken() == UNTIL || getToken() == RPAREN ||
getToken() == DOLLAR) {
59         // T' -> ε
60     } else {
61         printf("语法错误: 无法识别的项\n");
62         exit(EXIT_FAILURE);
63     }
64     if (debug)
65         printf("离开: TermPrime\n");
66 }
67
68 void MulOp() {
69     if (debug)
70         printf("进入: MulOp\n");
71     if (getToken() == TIMES) {
72         match(TIMES);
73     } else if (getToken() == OVER) {
74         match(OVER);
75     } else {
76         printf("语法错误: 期望乘法操作符\n");
77         exit(EXIT_FAILURE);
78     }
79     if (debug)
80         printf("离开: MulOp\n");
81 }

```


3.4.12. 解析因子

Factor 函数解析因子，可以是括号内的表达式、标识符或数字。

```
1 void Factor() {
2     if (debug)
3         printf("进入: Factor\n");
4     if (getToken() == LPAREN) {
5         match(LPAREN);
6         Exp();
7         match(RPAREN);
8     } else if (getToken() == ID) {
9         match(ID);
10    } else if (getToken() == NUM) {
11        match(NUM);
12    } else {
13        printf("语法错误: 无法识别的因子\n");
14        exit(EXIT_FAILURE);
15    }
16    if (debug)
17        printf("离开: Factor\n");
18 }
```

3.4.13. 主函数

在 **main** 函数中，我们读取命令行参数，判断是否开启调试模式，然后读取词法分析器的输出文件并调用 **Program** 函数开始语法分析。

```
1 int main(int argc, char *argv[]) {
2     if (argc < 2 || argc > 3) {
3         printf("用法: %s <词法分析器输出文件> [-d]\n", argv[0]);
4         return 1;
5     }
6
7     if (argc == 3 && strcmp(argv[2], "-d") == 0) {
8         debug = 1;
9     }
10
11    readTokens(argv[1]);
12    Program();
13
14    return 0;
15 }
```

通过以上步骤，我们实现了一个基于递归下降的语法分析器，并且在调试模式下可以输出进入和离开每个解析函数的调试信息，方便发现和解决问题。

3.4.14. 编译和运行语法分析器

我们使用以下命令编译语法分析器：

```
1 gcc -o parser parser.c
```

然后，我们可以运行语法分析器并指定词法分析器的输出文件：

```
1 | ./parser output.txt
```

在调试模式下运行语法分析器的输出为：

```
1 | 进入: Program
2 | 进入: StmtList
3 | 进入: Stmt
4 | 进入: ReadStmt
5 | 匹配: READ
6 | 匹配: ID
7 | 离开: ReadStmt
8 | 离开: Stmt
9 | 进入: StmtListPrime
10 | 匹配: SEMI
11 | 进入: Stmt
12 | 进入: IfStmt
13 | 匹配: IF
14 | 进入: Exp
15 | 进入: SimpleExp
16 | 进入: Term
17 | 进入: Factor
18 | 匹配: NUM
19 | 离开: Factor
20 | 进入: TermPrime
21 | 离开: TermPrime
22 | 离开: Term
23 | 进入: SimpleExpPrime
24 | 离开: SimpleExpPrime
25 | 离开: SimpleExp
26 | 进入: ExpPrime
27 | 进入: RelOp
28 | 匹配: RELOP
29 | 离开: RelOp
30 | 进入: SimpleExp
31 | 进入: Term
32 | 进入: Factor
33 | 匹配: ID
34 | 离开: Factor
35 | 进入: TermPrime
36 | 离开: TermPrime
37 | 离开: Term
38 | 进入: SimpleExpPrime
39 | 离开: SimpleExpPrime
40 | 离开: SimpleExp
41 | 离开: ExpPrime
42 | 离开: Exp
43 | 匹配: THEN
44 | 进入: StmtList
45 | 进入: Stmt
46 | 进入: AssignStmt
47 | 匹配: ID
48 | 匹配: ASSIGN
49 | 进入: Exp
```

50 进入: SimpleExp
51 进入: Term
52 进入: Factor
53 匹配: NUM
54 离开: Factor
55 进入: TermPrime
56 离开: TermPrime
57 离开: Term
58 进入: SimpleExpPrime
59 离开: SimpleExpPrime
60 离开: SimpleExp
61 进入: ExpPrime
62 离开: ExpPrime
63 离开: Exp
64 离开: AssignStmt
65 离开: Stmt
66 进入: StmtListPrime
67 匹配: SEMI
68 进入: Stmt
69 进入: RepeatStmt
70 匹配: REPEAT
71 进入: StmtList
72 进入: Stmt
73 进入: AssignStmt
74 匹配: ID
75 匹配: ASSIGN
76 进入: Exp
77 进入: SimpleExp
78 进入: Term
79 进入: Factor
80 匹配: ID
81 离开: Factor
82 进入: TermPrime
83 进入: MulOp
84 匹配: TIMES
85 离开: MulOp
86 进入: Factor
87 匹配: ID
88 离开: Factor
89 进入: TermPrime
90 离开: TermPrime
91 离开: TermPrime
92 离开: Term
93 进入: SimpleExpPrime
94 离开: SimpleExpPrime
95 离开: SimpleExp
96 进入: ExpPrime
97 离开: ExpPrime
98 离开: Exp
99 离开: AssignStmt
100 离开: Stmt
101 进入: StmtListPrime
102 匹配: SEMI
103 进入: Stmt
104 进入: AssignStmt
105 匹配: ID

106	匹配: ASSIGN
107	进入: Exp
108	进入: SimpleExp
109	进入: Term
110	进入: Factor
111	匹配: ID
112	离开: Factor
113	进入: TermPrime
114	离开: TermPrime
115	离开: Term
116	进入: SimpleExpPrime
117	进入: AddOp
118	匹配: MINUS
119	离开: AddOp
120	进入: Term
121	进入: Factor
122	匹配: NUM
123	离开: Factor
124	进入: TermPrime
125	离开: TermPrime
126	离开: Term
127	进入: SimpleExpPrime
128	离开: SimpleExpPrime
129	离开: SimpleExpPrime
130	离开: SimpleExp
131	进入: ExpPrime
132	离开: ExpPrime
133	离开: Exp
134	离开: AssignStmt
135	离开: Stmt
136	进入: StmtListPrime
137	离开: StmtListPrime
138	离开: StmtListPrime
139	离开: StmtList
140	匹配: UNTIL
141	进入: Exp
142	进入: SimpleExp
143	进入: Term
144	进入: Factor
145	匹配: ID
146	离开: Factor
147	进入: TermPrime
148	离开: TermPrime
149	离开: Term
150	进入: SimpleExpPrime
151	离开: SimpleExpPrime
152	离开: SimpleExp
153	进入: ExpPrime
154	进入: RelOp
155	匹配: RELOP
156	离开: RelOp
157	进入: SimpleExp
158	进入: Term
159	进入: Factor
160	匹配: NUM
161	离开: Factor

```
162  进入: TermPrime
163  离开: TermPrime
164  离开: Term
165  进入: SimpleExpPrime
166  离开: SimpleExpPrime
167  离开: SimpleExp
168  离开: ExpPrime
169  离开: Exp
170  离开: RepeatStmt
171  离开: Stmt
172  进入: StmtListPrime
173  匹配: SEMI
174  进入: Stmt
175  进入: WriteStmt
176  匹配: WRITE
177  进入: Exp
178  进入: SimpleExp
179  进入: Term
180  进入: Factor
181  匹配: ID
182  离开: Factor
183  进入: TermPrime
184  离开: TermPrime
185  离开: Term
186  进入: SimpleExpPrime
187  离开: SimpleExpPrime
188  离开: SimpleExp
189  进入: ExpPrime
190  离开: ExpPrime
191  离开: Exp
192  离开: WriteStmt
193  离开: Stmt
194  进入: StmtListPrime
195  离开: StmtListPrime
196  离开: StmtListPrime
197  离开: StmtListPrime
198  离开: StmtList
199  进入: IfStmtPrime
200  匹配: END
201  离开: IfStmtPrime
202  离开: IfStmt
203  离开: Stmt
204  进入: StmtListPrime
205  离开: StmtListPrime
206  离开: StmtListPrime
207  离开: StmtList
208  匹配: DOLLAR
209  语法分析成功!
210  离开: Program
```

3.5. 生成中间代码

本节中，我们将在语法分析的基础上，建立抽象语法树。我们使用 Graphviz 绘制抽象语法树，并生成中间代码。

首先，定义抽象语法树的数据结构：

```
1  typedef struct ASTNode {
2      char label[256];
3      struct ASTNode *children[5];
4      int childCount;
5  } ASTNode;
6
7  ASTNode *createNode(const char *label) {
8      ASTNode *node = (ASTNode *)malloc(sizeof(ASTNode));
9      strcpy(node->label, label);
10     node->childCount = 0;
11     return node;
12 }
13
14 void addChild(ASTNode *parent, ASTNode *child) {
15     if (parent->childCount < 5) {
16         parent->children[parent->childCount++] = child;
17     } else {
18         printf("错误: 节点的子节点超过5个\n");
19         exit(EXIT_FAILURE);
20     }
21 }
```

在语法分析器中，我们对每个解析函数进行了修改，以生成抽象语法树。例如：

3.5.1. 解析 IfStmt

```
1  ASTNode *IfStmt() {
2      if (debug) printf("进入: IfStmt\n");
3      ASTNode *node = createNode("IfStmt");
4      addChild(node, match(IF));
5      addChild(node, Exp());
6      addChild(node, match(THEN));
7      addChild(node, StmtList());
8      addChild(node, IfStmtPrime());
9      if (debug) printf("离开: IfStmt\n");
10     return node;
11 }
```

3.5.2. 解析 Exp

```
1 ASTNode *Exp() {
2     if (debug) printf("进入: Exp\n");
3     ASTNode *node = createNode("Exp");
4     addChild(node, SimpleExp());
5     addChild(node, ExpPrime());
6     if (debug) printf("离开: Exp\n");
7     return node;
8 }
```

在 `match` 函数中，我们确保在匹配终结符时将其作为叶节点添加到抽象语法树中：

```
1 ASTNode *match(TokenType expected) {
2     if (getToken() == expected) {
3         if (debug) {
4             printf("匹配: %s\n", tokenNames[expected]);
5         }
6         ASTNode *node = createNode(tokenNames[expected]);
7         if (tokens[currentToken].attr[0] != '\0') {
8             char label[512]; // 增加缓冲区大小
9             snprintf(label, sizeof(label), "%s: %s", tokenNames[expected],
10 tokens[currentToken].attr);
11             strcpy(node->label, label);
12         }
13         currentToken++;
14         return node;
15     } else {
16         printf("语法错误: 期望 %d, 得到 %d\n", expected, getToken());
17         exit(EXIT_FAILURE);
18     }
19 }
```

3.5.3. 输出 Graphviz 格式的抽象语法树

我们添加了一个函数，用于输出抽象语法树为 Graphviz 格式：

```
1 void printAST(ASTNode *node, FILE *file) {
2     if (node == NULL)
3         return;
4     fprintf(file, "\"%p\" [label=\"%s\"];\n", node, node->label);
5     for (int i = 0; i < node->childCount; i++) {
6         fprintf(file, "\"%p\" -> \"%p\";\n", node, node->children[i]);
7         printAST(node->children[i], file);
8     }
9 }
```

3.5.4. 修改 `main` 函数以支持生成抽象语法树

在 `main` 函数中，我们添加了对 `-AST` 参数的处理，并调用 `printAST` 函数输出抽象语法树：

```
1 int main(int argc, char *argv[]) {
2     if (argc < 2 || argc > 4) {
```

```

3     printf("用法: %s <词法分析器输出文件> [-d] [-AST ast.dot]\n", argv[0]);
4     return 1;
5 }
6
7 char *astFilePath = NULL;
8 for (int i = 2; i < argc; i++) {
9     if (strcmp(argv[i], "-d") == 0) {
10         debug = 1;
11     } else if (strcmp(argv[i], "-AST") == 0 && i + 1 < argc) {
12         astFilePath = argv[++i];
13     }
14 }
15
16 readTokens(argv[1]);
17 ASTNode *root = Program();
18
19 if (astFilePath) {
20     FILE *astFile = fopen(astFilePath, "w");
21     if (!astFile) {
22         perror("无法打开AST输出文件");
23         return 1;
24     }
25     fprintf(astFile, "digraph AST {\n");
26     printAST(root, astFile);
27     fprintf(astFile, "}\n");
28     fclose(astFile);
29 }
30
31 return 0;
32 }

```

通过以上修改，可以生成抽象语法树并输出为 Graphviz 格式。运行语法分析器时，使用 `-AST` 参数指定输出文件：

```
1 | ./parser output.txt -AST ast.dot
```

这样，抽象语法树将输出到 `ast.dot` 文件中，可以使用 Graphviz 工具进行可视化：

```
1 | dot -Tpng ast.dot -o ast.png
```

示例程序的抽象语法树如下所示：

