LL(1)语法分析器——实验报告

作者：郭栩源

一、实验题目要求

本次实验要求编写一个LL(1)预测分析器，它通过预测分析表进行句子的语法分析。
要求所分析算数表达式由如下的文法产生：
E→E+T | E–T | T
T→T*F | T/F | F
F→(E) | num
本人使用C语言实现语法分析器，开发环境为macOS 13.0和Windows 10专业版22H2。

二、程序设计说明

前置工作

在进行程序设计之前，首先将源文法改造成LL(1)文法，即进行消除左递归和提取左公因子。得到如下文法：
E → T E_PRIME
E_PRIME → + T E_PRIME | - T E_PRIME | EPLISON
T → F T_PRIME
T_PRIME → * F T_PRIME | / F T_PRIME | EPLISON
F → (E) | num
需要注意的是，并非所有文法在进行以上两步变换后都能成为LL(1)文法，但本例中的文法被证明确实可以化为LL(1)文法。
然后，对该文法的每个非终结符求其FIRST和FOLLOW集：

非终结符	FIRST	FOLLOW
E	(,num,	),$
E_PRIME	+,-,EPLISON	),$
T	(,num	+,-,),$
T_PRIME	*,/,EPLISON	+,_,),$
F	(,num	*,/,+,-,),$

程序全局变量和函数简介

宏，用于定义一些基础常量，如产生式数量、终结符数量、非终结符数量等：

```
#define NUM_OF_PRODUCTION 10
#define NUM_OF_TERMINATE 9
#define NUM_OF_NONTERMINATE 5
#define MAX_TOKEN_SIZE 10
#define STACK_SIZE 100
#define BUFFER_SIZE 100
```

全局变量：

```
FILE *in,*out;                  // 读写文件指针
int step;                       // 全局计数器，记录当前操作步数
int rsp,stack[STACK_SIZE];      // rsp是栈顶指针，stack是解析栈
int ip,buffer[BUFFER_SIZE];     // ip是输入缓冲区的指针，buffer存储输入的令牌序列

// 所有的符号（包括终结符和非终结符）都被定义在一个`enum`结构中
// 此处将错误表项ERROR和SYNCH也加入符号中
// 且ERROR值为0（默认值），因此在预测分析表中空表项默认为ERROR。
enum symbol_type {
    ERROR,SYNCH,
    EPLISON,END,PLUS,MINUS,MULTIPLY,DIVIDE,LEFT_PAREN,RIGHT_PAREN,NUMBER,
    E,E_PRIME,T,T_PRIME,F
};

// 符号的名称数组，用于输出
char *symbol_name[2+NUM_OF_TERMINATE+NUM_OF_NONTERMINATE]= {
        "ERROR","SYNCH","EPLISON","END","+","-","*","/","(",")","NUM",
        "E","E_PRIME","T","T_PRIME","F"
};

// FOLLOW集
struct follow {
        int non_terminate;
        int follow[NUM_OF_TERMINATE+1];
} follows[NUM_OF_NONTERMINATE+1];

// 产生式被定义为一个结构，其中包含左边的非终结符、右边的符号序列和FIRST集.
// 此处的left并不等同于生成式左端，只有当left>=2+NUM_OF_TERMINATE时，
// 即对应非终结符时，其含义为生成式左端。
// 特殊地，left为0或1时，在预测分析表中代表了错误表项ERROR和SYNCH。
struct production {
        int left;                               // 左边的非终结符
        int right[MAX_TOKEN_SIZE];              // 右边的符号序列
        int first[NUM_OF_TERMINATE+1];          // FIRST集
} productions[NUM_OF_PRODUCTION+1],         //全部产生式
analyse_table[2+NUM_OF_TERMINATE+NUM_OF_NONTERMINATE][2+NUM_OF_TERMINATE+NUM_OF_NONTERMINATE];
```

函数:

```
int is_termainate(int symbol);  // 判断一个符号是否是终结符
int get_type(char* name);       // 通过名称得到符号类型
void buf_init();                // 初始化输入缓冲区，将输入文件转换为令牌序列
void stack_init();              // 初始化解析栈
void analyse_table_init();      // 初始化预测分析表
void error(int type);           // 错误处理函数，答应错误信息，并进行错误恢复
void print_info();              // 打印当前的栈、输入缓冲区
void print_production(struct production pd);    // 打印当前所使用的产生式
```

预测分析表的构造

主要步骤如下：

- 对文法的每个产生式A→alpha，对FIRST(alpha)中的每一个终结符a，将A→alpha放到表项M[A,a]中。
- 对文法的每个产生式A→alpha，若FIRST(alpha)中含有EPLISON，则对每个FOLLOW(A)中的符号b，将A→alpha放到表项M[A,b]中。
- 对于每个非终结符A，对每个FOLLOW(A)中的符号b，若M[A,b]为空，将SYNCH填入M[A,b]，用于错误处理。
- 在预测分析表的所有其他空白项中填入ERROR（本程序中空表项默认为ERROR）。

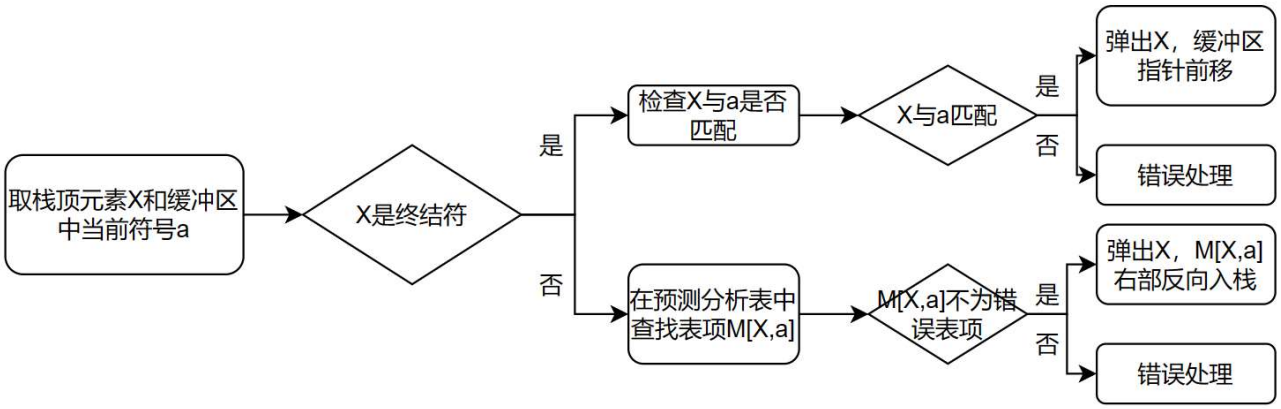
详细实现见源程序main.c中的analyse_table_init()函数。

预测分析程序

主预测分析程序位于main函数中的do-while循环中，其主要功能为：

- 每次取栈顶元素X，和输入缓冲区中当前输入元素a。
- 若X为终结符，则检查X与a是否匹配。若匹配，弹出栈顶符号X且缓冲区指针ip前移；若不匹配，进行错误处理（参数0）。
- 若X为非终结符，则在预测分析表中查找表项M[X,a]，若M[X,a]不为错误表项，则先弹出栈顶元素，然后将M[X,a]的右部反向入栈；否则进行错误处理（参数1）。

预测分析程序工作流程图如下：



详细实现见源程序main函数中的do-while循环。

错误检测与错误处理

该程序中定义了两种错误类型：

1. 分析栈栈顶为终结符，但与当前输入符号不匹配（用错误类型0表示）。
2. 分析栈栈顶为非终结符X，当前输入符号为a，但分析表中M[X,a]为错误表项。

对于这两种类型的错误，秉持最小化错误的影响，采用一种应急式的错误处理方法：

1. 对错误类型0，直接弹出栈顶终结符。
2. 对错误类型1，当前输入符号为a，若M[X,a]为ERROR，程序不断向前移动输入缓冲区指针，直到可继续处理为止。这里的可继续处理有两种情况，一是M[X,a]中包含产生式，此时继续分析即可；二是M[X,a]为SYNCH，此时从栈顶弹出符号X并继续进行分析。弹出X的原因是，a是FOLLOW(X)中的元素，弹出X有助于更好地继续分析。

三、测试报告

测试样例1

输入：

```
num + num
```

输入说明：该测试用例旨在测试语法分析程序的基础语法分析功能。

输出：

```
--------Times of operations:0-------
[stack]:
END E
[input]:
NUM + NUM * NUM END

--------Times of operations:1-------
[stack]:
END E_PRIME T
[input]:
NUM + NUM * NUM END
[output]:
E -> T E_PRIME

--------Times of operations:2-------
[stack]:
END E_PRIME T_PRIME F
[input]:
NUM + NUM * NUM END
[output]:
T -> F T_PRIME

--------Times of operations:3-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM + NUM * NUM END
[output]:
F -> NUM

--------Times of operations:4-------
[stack]:
END E_PRIME T_PRIME
[input]:
+ NUM * NUM END

--------Times of operations:5-------
[stack]:
END E_PRIME
[input]:
+ NUM * NUM END
[output]:
T_PRIME -> EPLISON

--------Times of operations:6-------
[stack]:
```

```
END E_PRIME T +
[input]:
+ NUM * NUM END
[output]:
E_PRIME -> + T E_PRIME

--------Times of operations:7-------
[stack]:
END E_PRIME T
[input]:
NUM * NUM END

--------Times of operations:8-------
[stack]:
END E_PRIME T_PRIME F
[input]:
NUM * NUM END
[output]:
T -> F T_PRIME

--------Times of operations:9-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM * NUM END
[output]:
F -> NUM

--------Times of operations:10-------
[stack]:
END E_PRIME T_PRIME
[input]:
* NUM END

--------Times of operations:11-------
[stack]:
END E_PRIME T_PRIME F *
[input]:
* NUM END
[output]:
T_PRIME -> * F T_PRIME

--------Times of operations:12-------
[stack]:
END E_PRIME T_PRIME F
[input]:
```

```
NUM END

--------Times of operations:13-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM END
[output]:
F -> NUM

--------Times of operations:14-------
[stack]:
END E_PRIME T_PRIME
[input]:
END

--------Times of operations:15-------
[stack]:
END E_PRIME
[input]:
END
[output]:
T_PRIME -> EPLISON

--------Times of operations:16-------
[stack]:
END
[input]:
END
[output]:
E_PRIME -> EPLISON

--------Times of operations:17-------
[stack]:
[input]:
END
```

测试结果：语法分析程序分析结果正确，有基础的语法分析功能。

测试样例2

输入：

```
num + num * num + + num num
```

输入说明：输入中存在语法错误，用以检测程序的错误处理能力。

输出：

```
--------Times of operations:0-------
[stack]:
END E
[input]:
NUM + NUM * NUM + + NUM NUM END

--------Times of operations:1-------
[stack]:
END E_PRIME T
[input]:
NUM + NUM * NUM + + NUM NUM END
[output]:
E -> T E_PRIME

--------Times of operations:2-------
[stack]:
END E_PRIME T_PRIME F
[input]:
NUM + NUM * NUM + + NUM NUM END
[output]:
T -> F T_PRIME

--------Times of operations:3-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM + NUM * NUM + + NUM NUM END
[output]:
F -> NUM

--------Times of operations:4-------
[stack]:
END E_PRIME T_PRIME
[input]:
+ NUM * NUM + + NUM NUM END

--------Times of operations:5-------
[stack]:
END E_PRIME
[input]:
+ NUM * NUM + + NUM NUM END
[output]:
T_PRIME -> EPLISON

--------Times of operations:6-------
[stack]:
```

```
END E_PRIME T +
[input]:
+ NUM * NUM + + NUM NUM END
[output]:
E_PRIME -> + T E_PRIME

--------Times of operations:7-------
[stack]:
END E_PRIME T
[input]:
NUM * NUM + + NUM NUM END

--------Times of operations:8-------
[stack]:
END E_PRIME T_PRIME F
[input]:
NUM * NUM + + NUM NUM END
[output]:
T -> F T_PRIME

--------Times of operations:9-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM * NUM + + NUM NUM END
[output]:
F -> NUM

--------Times of operations:10-------
[stack]:
END E_PRIME T_PRIME
[input]:
* NUM + + NUM NUM END

--------Times of operations:11-------
[stack]:
END E_PRIME T_PRIME F *
[input]:
* NUM + + NUM NUM END
[output]:
T_PRIME -> * F T_PRIME

--------Times of operations:12-------
[stack]:
END E_PRIME T_PRIME F
[input]:
```

```
NUM + + NUM NUM END

--------Times of operations:13-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM + + NUM NUM END
[output]:
F -> NUM

--------Times of operations:14-------
[stack]:
END E_PRIME T_PRIME
[input]:
+ + NUM NUM END

--------Times of operations:15-------
[stack]:
END E_PRIME
[input]:
+ + NUM NUM END
[output]:
T_PRIME -> EPLISON

--------Times of operations:16-------
[stack]:
END E_PRIME T +
[input]:
+ + NUM NUM END
[output]:
E_PRIME -> + T E_PRIME

--------Times of operations:17-------
[stack]:
END E_PRIME T
[input]:
+ NUM NUM END

ERROR: 分析栈栈顶为T，当前输入符号为+，分析表为空
SOLUTION: 弹出栈顶符号T

--------Times of operations:18-------
[stack]:
END E_PRIME T +
[input]:
+ NUM NUM END
```

```
[output]:
E_PRIME -> + T E_PRIME

--------Times of operations:19-------
[stack]:
END E_PRIME T
[input]:
NUM NUM END

--------Times of operations:20-------
[stack]:
END E_PRIME T_PRIME F
[input]:
NUM NUM END
[output]:
T -> F T_PRIME

--------Times of operations:21-------
[stack]:
END E_PRIME T_PRIME NUM
[input]:
NUM NUM END
[output]:
F -> NUM

--------Times of operations:22-------
[stack]:
END E_PRIME T_PRIME
[input]:
NUM END

ERROR: 分析栈栈顶为T_PRIME，当前输入符号为NUM，分析表为空
SOLUTION: 右移输入缓冲区指针，当前输入缓冲区为:END

--------Times of operations:23-------
[stack]:
END E_PRIME
[input]:
END
[output]:
T_PRIME -> EPLISON

--------Times of operations:24-------
[stack]:
END
[input]:
```

```
END
[output]:
E_PRIME -> EPLISON

--------Times of operations:25-------
[stack]:
[input]:
END
```

四、使用说明

在Windows系统下，可在命令行中使用如下命令编译和运行该程序：

```
gcc main.c -o main -std=c99
main in out
```

在macOS和Linux下使用如下命令：

```
gcc main.c -o main
./main in out
```

其中，输入应写在in文件中，输出被自动保存在out文件中。

五、改进与完善

该程序实现了简单的语法分析功能，但还有许多可以改进和完善的方面：

1. 扩展性：该程序的文法定义和FIRST、FOLLOW集都存储在程序内部，所以只能够对一种特定文法进行语法分析。改进方法是：让程序读入文法并自动分析构造FIRST集和FOLLOW集。
2. 错误恢复：该程序的错误处理只是一种简单的应急式的错误处理方法，无法做到错误影响最小化。改进方法是：定义更多错误类型，针对不同错误类型采取不同错误处理方式。

还有其他如增强程序健壮性等优化方式，此处不再一一赘述。