词法分析

19376391 丁磊

编码之前设计概述

词法分析器应该编写一个核心方法，每次调用该方法时返回一个解析过的单词。外层可以不断调用该方法，从而获取输入中的所有单词，并生成类别码，完成全部输入的解析。为完成该核心方法，还需要有一些辅助的方法，例如中途解析至一半失败时退回已经读入的字符。

核心设计

- 不断调用`getchar()`读取字符
- 跳过空白字符
- 可能为保留字则连续读取判断
- 数字则连续读取
- 单个运算符读取后判断是否为一元符
    - 可能为二元符则需要继续读取
- 在以上任何一个中途读取失败时，需要调用`retract()`退回已经读入的字符，以便于后续匹配其他单词。

以函数`nextsym()`为核心，产生token并输出至文件，主函数中读取输入文件所有字符，循环调用此函数读取字符生成token，直至文件末尾。

`nextsym()`内部参考课本上的设计，读取非空白符，连续读取识别标识符并判断是否是保留字，连续读取整数，读取单个字符判断是否为一元分隔符，连续读取判断是否为二元分隔符。若以上一条满足，将当前token的字符串表示及分类分别存入`token`和`symbol`并输出至文件。

位置记录

每次记录下标位置，读取时自增，未读取到换行字符时列号自增，行号不变，如果读取到换行符，则需要将行号加一，并将列号重置。

单词字典

对于保留的关键字或者特殊的符号，将其作为成员变量保留下来，通过map管理。这样在识别到类似于main、int等单词的时候通过查表可以直接获取类别码，可以忽略复杂状态机的转移设计，对于一些较短的单词可以考虑通过状态机的转移来判别。

完善实现

文件流输入

通过分析，上述过程应当在词法分析器建立时完成，于是在具体实现部分，将该工作放入词法分析器的初始化方法中，外层函数在实例化词法分析器时传入文件的路径，词法分析器在初始化时将内容读入，部分初始化代码如下。

解析单词

每次调用时，需要先清空缓存中的单词，然后跳过空白字符

```
clearToken();
read_char();
while (isspace(c) && (position < src.length())) {
    read_char();
}
```

对于足够唯一确定的单词，则通过已经存好的map查询出对应的类别码，对于部分前缀相同的字符暂时无法判断，则需要继续读入字符直到可以预读出唯一确定的单词，值得注意的是，当这类字符读取失败时，应该退回后续读入的字符，避免影响后续的解析。

```
if (c == '<') {
    read_char();
    if (c == '=') {
        token = "<=";
        symbol = "LEQ";
    } else {
        token = "<";
        symbol = "LSS";
        retract();
    }
}
```

注释文本判断

在一开始设计时，并没有考虑到文档中提及注释的情况，所以一开始的提交中出现了较大的问题。后续加入了注释的判断，并通过标识符来区分。设定标识符ANNOTATION。对于文档中提到的两种不同注释形式，分别用具有代表性的`/**/`和`//`来区分。

注释终止条件判断

- 若为`/*`，多行注释，则循环读取直到下一个`*/`出现。
- 若为`//`，单行注释，则只需要读取至换行位置。

```
if (c == '/') {
    read_char();
    if (c == '*') {
        while (true) {
            read_char();
            if (c == '*') {
                read_char();
                if (c == '/') {
                    break;
                } else {
                    retract();
                }
            }
        }
        token = "/**/";
        symbol = "ANNOTATION";
```

```
    } else if (c == '/') {
        while (true) {
            read_char();
            if (c == '\n') {
                break;
            }
        }
        token = "//";
        symbol = "ANNOTATION";
    }
```

退回一个字符

构想时如果采用一次读入文件的一个字符，在文件的操作中也能实现。但实际使用中并不方便，对于指针位置的控制并不直观，为了更加直观地控制字符的回退，利用文件流一次性将文本内容读入。

由于实现了文件流的输入，为了实现回退的功能，在读取时记录读取位置指针，当需要回退的时候对应修改指针的位置即可实现。

检查语法错误

错误处理中strcon的字符有范围限制，由于这里实现时是词法分析器一次性读取，所以考虑将该类习惯的错误处理在词法分析器中检查。在之前读取完一个单词之后，判断类型是否是strcon，如果是则需要重新检查其中的每一个字符，如果超出了合理的范围，则报错。

语法分析

概述

采用递归子程序法进行自顶而下的分析。在调用子程序前先读入一个token，然后每个子程序内通过读入token以及递归调用其他子程序来分析一种非终结符号，并将语法成分输出到文件中。

采用预读的方法，即在多个选择间存在冲突时提前读数个token直到可以进行判断出唯一选择，并回退到预读前的token，然后调用该选择的子程序。当读入的终结符号与预期不同时，产生异常，保存至错误表中。

其中关于`<Expr>`部分的语法相对比较复杂，且存在左递归，需要根据情况来改写文法以适应程序。

实现

语法分析器`Grammar`内部含有词法分析器`Lexer`成员，通过词法分析器获取单词，再根据语法来进行递归子程序的调用。

为了解决不能唯一确定语法的问题需要预读，预读后需要回退至预读前的位置，因此将待输出内容按行保存至数组中，在回退时清除暂时预读得到的结果。

使用变量来分别保存当前读到的词法分析结果、结果的词法成分、在所有词法分析tokens中的位置。

成员的方法至少有读入token、预读结束后回退、输出语法成分。

各个递归子程序作为方法保存在类中。

顶层函数调用最外层的语法解析，将自动递归地解析完输入文件。

每个递归子程序在调用前需要先使用`next_sym()`读入一个token，然后根据右部各选择的首符号进行选择（必要时采用预读），对于非终结符号调用其子程序，终结符号则判断是否与预期一致。

左递归问题

```
1  AddExp → MulExp | AddExp ('+' | '-') MulExp
```

对于可能存在左递归的情况，可以先改写文法消除程序解析上的左递归无限循环问题，然而在这里需要注意语法成分的输出。例如AddExp虽然通过文法的改写可以解析，但是如果每次都改过文法后直接调用MulExp的递归解析子程序，在输出结果MulExp时将缺少外层AddExp的非终结符信息，与作业要求的输出并不符合。

为了解决这个问题，我当时想了几种策略，例如调用MulExp前先手动添加一个非终结符的输出语法信息，但这样感觉并不利于维护，并且在很多Exp的解析中都需要改，一旦后续需要修改或者有新的输出方式，就很容易出错。第二种是在语法解析器中做好相关的记录，将表达式的结构分析清楚后再添加输出，这也是这次作业中我采用的方案。

预读和回退

为了解决不能唯一确定语法的问题需要预读，预读后需要回退至预读前的位置，因此将待输出内容按行保存至数组中，在回退时清除暂时预读得到的结果。结束时再统一将数组中需要输出的信息打印出来。

数组部分语法处理

数组部分的语法相对比较复杂，并且由于结构并不单一，所以为了判断具体的非终结符进行预读的操作。例如表达式或赋值语句如果识别到一个数组变量，都需要继续往下预读数组维数等信息，直到读取到等号或者其他运算符才能确定当前的非终结符。

部分处理细节

main函数前的声明和定义

先进行循环读取，如果读取到const，则一定是常量的声明，调用const_declare的解析程序。

```
while (sym == "CONSTTK" || sym == "INTTK" || sym == "VOIDTK") {
        if (sym == "CONSTTK") {
            const_declare();
            next_sym();
            continue;
        }
    ....
}
```

如果是void、int类型，也可以读取到括号时确定是函数的声明，这时调用函数的声明解析。如果没找到括号，则是int型变量的声明。值得注意的是，读取到int时需要判别函数名是否为main，如果是main则进入main函数的解析。

```
if (sym == "INTTK") {
    next_sym();
    if (sym == "MAINTK") {
        retract();
        main_function_define();
        return;
    } else {
        retract();
        function_define();
    }
}
```

declare

考虑到可能有逗号分隔的多个变量，利用循环读取并判断当前符号，不是逗号时退出。退出时检查当前符号是否为分号，如果不是分号则报错。

```
while (sym == "COMMA") {
        declare();
        next_sym();
}
if (sym != "SEMICN") {
// Error
}
```

define

这一部分主要工作量在于对数组的判断，读取到变量名时继续预读一个符号，如果是中括号则还需调用表达式解析完后继续预读，判断是一维数组还是二维数组。

block

block顶层负责循环调用block_item的解析

```
while (sym != "RBRACE") {
    block_item();
    next_sym();
}
```

block_item根据当前符号调用declare或statement。

```
if (sym == "INTTK") {
    variable_declare();
} else if (sym == "CONSTTK") {
    const_declare();
} else {
    statement();
}
```

statement相对来讲比较复杂，这里先根据如果能直接确定的下一步解析，则先进行，例如if、while等，直接调用相关的解析，如果是表达式等不能直接确定的情况，预读符号，直到能确定时调用相关解析，预读结束后如果不是则需要回退相关的字符。

改进

表达式类型的识别

错误处理作业中，由于设计表达式类型的求解，在最初设计中并未考虑，这样导致难以判断函数传参的正确性。于是修改语法分析器中表达式识别相关部分，使其能够递归地返回表达式的类型。

```
DataType expression();
DataType add_expression();
DataType mult_expression();
DataType unary_expression();
DataType primary_expression();
DataType left_value();
```

当前所在层次

最初的设计并未考虑当前层数位置的记录，在实现符号表时发现这是一个很重要的信息，并且在错误处理的变量重定义和未定义中起到很重要的作用，所以对相关的层次都进行记录，也包括当前位于的函数位置，以便检查函数的形式参数的问题。

```
int block_level;
int loop_level;
string function;
```

通过function记录的函数名字查询符号表，获得函数的参数。

错误处理

为了进行错误处理，需要有符号表的辅助。

符号表

符号表有一些基本的属性记录，变量名，数据类型，变量的种类，所在层次关系，预留的地址等。如果是数组，则需要进一步记录数组的维数，如果是函数，则需要记录下函数的参数信息。

应当封装好几个基本操作，加入和查询。

在加入时也需要先进行查询，以判断是否出现了重定义等情况，所以查询应该是更为基础的一个操作。

插入符号

先查询在当前层次下有没有相同的变量，如果查询到相同的变量，则出现重定义的错误，报错返回。

如果没有则创建一个新的记录，通过函数参数记录下当前变量的各种信息，下标位置增加。

查询符号

考虑到内层变量定义会覆盖外层，所以优先在当前层次寻找变量，如果找到则成功返回，找不到则向上一个层级继续寻找，以此类推。如果在最后一个层级还没有找到，则说明变量还未定义。

错误处理

创建Error对象存储错误相关信息行号、列号、错误类型等，以及为了便于调试的富文本。

```
class Error {
    int line;
    int column;
    int id;
    char code;
    String richMsg;
}
```

检查错误时先存放在数组中，解析完成后输出到error文件。

非法符号

在FormatString中检查，如果出现错误，利用已有的行号记录报错。

重定义

调用符号表封装好的查询函数，如果在定义前查找发现已经定义过了，则需要报错重定义。主要是在define部分，还有函数形式参数也不能重复定义。

未定义

在使用时标识符还没定义的错误。主要在左值和一元表达式中

```
Item item = SymTable.search(function, tk.origin, block_level);
if (!item.valid) {
    // Error
}
```

函数形参

调用函数时检查，先记录下实际参数的个数，实现时将数据类型、标识符一起存储。之后查询符号表中的形式参数，如果个数不一样则数量不匹配，匹配后继续逐一检查数据类型是否匹配，若有不匹配则报错。

函数返回值

由于文档中的要求做了简化，所以配合语法分析器记录的所在层次即可判断是否正确返回。

常量

调用左值时查询符号表，如果是const类型则不能修改。

格式和符号问题

在非终结符推出的语法中检查，调用相关解析子程序返回后继续读一个字符，判断是否为预期字符，若不是则缺少相关符号，报错。

代码生成

编码前设计

在代码生成部分，需要充分使用到之前设计的符号表以及语法分析的结果，新加入分析时得到的中间代码，然后由中间代码解释执行或者生成目标代码即可。中间代码表达形式采用四元式，记录操作数1、操作符、操作数2、结果四个部分。

编码后实现

首先第一个部分是修改符号表。由于之前在错误处理中设计的符号表过于简单，虽然具备了基本的功能，但对于变量的信息记录的太少，并不利于中间代码的生成，并且在遇到函数调用返回等需要及时维护变量的功能实现得并不好，所以重新修改了符号表的实现。

符号表中记录了变量的名称、种类、数据类型、维数、地址、变量值、所在`block`深度、以及父指针。这里地址只是记录相对符号表的基地址，在生成中间代码时需要具体考虑。

```
enum DataType {
    invalid_data_type,
    integer,
    integer_1,
    integer_2,
    const_int,
    const_int_1,
    const_int_2,
    block_ret
};
```

并定义可视化的方法，将每个符号转换成对应可读的字符串信息，在调试时可以展示出来。

```
==============================
NAME  KIND  TYPE          DIM ADDR VALUE LEV VALUE
------------------------------
---FUNC_GLOBAL
array var   int[][]       2   0          0   dummy parent symtable  0
len   var   int           0   4          0   dummy parent symtable  0
swap  func  void          0   0          0   dummy parent symtable  0
;;;parameter;;;
int          i
int          j
;;;;;;;;;;;;;;
sort  func  void          0   0          0   dummy parent symtable  0
%RET  tmp   void          0   224        0   dummy parent symtable  0
main  func  int           0   0          0   dummy parent symtable  0
------------------------------
---main
#T65  tmp   int           0   304        1   dummy parent symtable  0
i     var   int           0   308        1   dummy parent symtable  0
i     var   int           0   312        2   dummy parent symtable  0
#T66  tmp   int           0   316        1   dummy parent symtable  0
x     var   int           0   320        2   dummy parent symtable  0
```

符号表基本的插入、查询操作都先检查是否已经存在变量名字相同且所在层次也相同的符号，特别地，如果是函数和变量名相同是允许存在的。

- 如果当前是插入操作，则产生重定义的错误。
- 如果当前是查询操作，则产生未定义的错误。

未产生错误则继续当前操作，查询时需要利用层次信息逐层往上查询，插入时记录符号的有关信息，变量值初始化信息。

中间代码设计

中间代码的设计我考虑使用简洁的方式，中间代码本身不包含复杂的操作，将这些操作留给解析的程序去做，中间代码本身只保留两个操作数和一个操作符、一个返回值。

中间代码管理类有静态变量，用于赋值给临时变量名称，每一步基本的操作需要临时变量的保存，每一次运算时外部程序调用请求一个临时变量的名字，中间代码管理类需要返回一个名字，并将静态变量的记录增加一。

```
result = "#T" + str;  // from code_index
code_index++;
```

例如

```
int x = a + b + c;
```

中间代码生成为

```
#T1 <- a + b;
#T2 <- #T1 + c;
x <- #T2;
```

对于getint()操作，由于左值可能是多种情况，如果由中间代码记录将会变得复杂，所以采用将输入读取至临时变量，再将临时变量赋给左值两步操作。

```
c = getint();
d[2] = getint();
```

中间代码

```
#T1 <- getint()
c <- #T1
#T2 <- getint()
d[2] <- #T2
```

跳转操作只需要记录label，label的管理也需要通过中间代码管理类来实现，外部调用此类获取一个label，中间代码管理类的静态变量数需要自增。对于if语句生成两个标签，一个是如果条件为假，跳转至else语句块，另一个是if语句块执行完后无条件跳转至else结束后的位置。

```
result = "label_" + str;  // from label_index
label_index++;
```

```
int y = 1;
int x;
if (y == 1) {
    // true
    x = 2;
} else {
    x = 1;
}
```

中间代码

```
variable int y
y <- 1
variable int x
#T1 <- y == 1
jump to label_if_1 if #T1 false
x <- 2
jump to label_if_1_end
label_if_1
x <- 1
label_if_1_end
```

函数调用需要记录参数的传递，以及返回现场的相关操作

```
int hello(int a) {
    return a;
}

int main() {
    int b = hello(0);
    return 0;
}
```

中间代码

```
function define int hello
format para int a
return a
main function
variable int b
prepare call hello
push para 0 a
jump hello
return from hello with value 0
#RET <- 0
b <- #RET
```

具体详细操作符定义如下

```
OP_PRINT "PRINT"
OP_GETINT "GETINT"
OP_SCANF "SCANF"
OP_ASSIGN ":="
OP_ADD "+"
OP_SUB "-"
OP_MUL "*"
OP_DIV "/"
OP_MOD "%"
OP_NOT "!"
OP_OR "||"
OP_AND "&&"
OP_XOR "^"
OP_FUNC "FUNC"
OP_END_FUNC "END_FUNC"
OP_BLK "block"
```

```
17 OP_END_BLK "end block"
18 OP_ARR_LOAD "ARR_LOAD"
19 OP_ARR_SAVE "ARR_SAVE"
20 OP_ARR_COPY "ARR_COPY"
21 OP_LABEL "LABEL"
22 OP_JUMP_IF "JUMP_IF"
23 OP_JUMP_UNCOND "JUMP"
24 OP_PREPARE_CALL "PREPARE_CALL"
25 OP_CALL "CALL"
26 QUIT,  // exit program
27 PCWAIT,  // waiting user
28 PARA,  /* format para; para int num1 */
29 FUNC,  /* func def; num1 num2 (); num1: int or void, func type;
   num2: func name; */
30 CALL,  /* call func; num1: func name, res, num2: minus $sp for
   local variable and register save; */
31 PUSH,  /* push parameter; num1: func name; num2: address, base
   on $sp; */
32 RET,  /* return value; num1: value,  or null; */
33 OP_PUSH_PARA "PUSH_PARA"
34 OP_PUSH_ONEARR "PUSH_ONEARR"
35 OP_PUSH_TWOARR "PUSH_TWOARR"
36 OP_ADD_ARRAY "ADD_ARRAY"
37 OP_RETURN "RETURN"
38 OP_EMPTY "EMPTY"
39 OP_PLACEHOLDER "PLACEHOLDER"
40 OP_VAR "VAR"
41 OP_CONST "CONST"
42 ASSIGN,  /* assign value ans = num1 */
43 PCNOTASSI, NOT,  /* assign not value ans = !num1 */
44 ARYEQ,  /* assign array; ans[num1] = num2 */
45 EQARY,  /* assign array; ans = num1[num2] */
46 INT,  /* variable; int res(=num1); */
47 CONSTINT,  /* variable; const int res(=num1); */
48 INTARR,  /* variable; array int num1[num2] */
49 CONSTARR,  /* variable; const array int num1[num2] */
50 PLUS, MINUS, PCMULT, PCDIV, PCMOD, /** calculate */
51 NEG,  /* negative; unary; ans =- num1 */
52 OP_EQL "EQL"
53 OP_NEQ "NEQ"
54 OP_GEQ "GEQ"
55 OP_GRE "GRE"
56 OP_LEQ "LEQ"
```

```
OP_LSS "LSS"
OP_GOTO "GOTO"
OP_BNZ "BNZ"
OP_BZ "BZ"
```

解释执行思考

有了中间代码，即可对中间代码进行解释执行，例如

```
if (op == ASSIGN) {
    Symbol symbol = SymTable.getSymbol(num1);  // 查询左值
    symbol.value = SymTable.getSymbol(num2).value;  // 查询右边表达式值，并赋值
}
if (op == algorithm) {
    int r = op == OP_ADD ? v1 + v2 :    op == OP_SUB ? (v1 - v2) :
                                        op == OP_MUL ? (v1 * v2) :
                                        op == OP_DIV ? (v1 / v2) :
                                        op == OP_MOD ? (v1 % v2) :
                                        op == OP_OR  ? (v1 || v2) :
                                        op == OP_AND ? (v1 && v2) :
                                        op == OP_XOR ? (v1 ^ v2):
                                        op == OP_EQL ? (v1 == v2) :
                                        op == OP_NEQ ? (v1 != v2) :
                                        op == OP_GEQ ? (v1 >= v2) :
                                        op == OP_GRE ? (v1 > v2) :
                                        op == OP_LEQ ? (v1 <= v2) :
                                        op == OP_LSS ? (v1 < v2) : 19376391;
    PseudoCode.newTemp(r);
```

```
20  }
```

核心需要解决的问题主要在于函数的调用和现场的返回，可以采用在调用前保存现场，将本层使用变量都入栈保护，恢复时弹栈回复变量值，需要注意运行栈中应该能够处理递归的函数，避免修改同一处的符号表。

目标代码生成

通过上述的中间代码，实际上不难发现主要的指令实际上是由很多的基本运算式组成，对于这些指令的中间代码，可以将四元表达式直接翻译为目标代码。额外需要的工作是管理寄存器和内存空间。对于其他的跳转指令也根据中间代码参数中传入的label值直接进行目标代码的生成。在定义变量时，将字符串常量保存在`.data`段当中，调用函数和返回时注意栈指针的维护和现场恢复。

基本运算式

这一部分较为简单并且占据了主要的指令部分，根据四元式直接生成对应的指令即可，需要记录变量使用的寄存器，或者内存中的位置，取到正确的值即可生成相应的指令。

```
1  # a + b = c
2  add $t0, $t1, $t2
```

寄存器分配

这一部分的主要工作在于对变量之间关系的记录，需要记录当前寄存器和使用中的临时变量，修改时需要产生相应的记录。对当前被标记过且需要占用的寄存器需要及时写回，避免读入错误值。基本块内的临时寄存器到基本块末尾时将父块中临时变量占用的寄存器值写入内存，实现更新。

代码优化

基本块划分

根据解析出的中间代码，对于普通的运算等四元式并不关心，遇到控制语句、跳转、标签时需要标记基本块的位置。有了基本块后，即可根据基本块进行数据流的分析。对于跨基本块的数据流不关心，只进行基本块内的数据流分析。对于每个基本块记录父块parent和子块next。

函数内联

函数内联主要减少的是内存的访问和跳转指令，需要先检查当前函数是否为递归调用的函数，如果不是即可进行内联，内联时需要将被调用函数的变量增加至调用函数的符号表中，这里需要注意的是对于同名的变量名进行改名的操作，避免变量的冲突，然后将传递参数的过程变换为赋值操作。

常量传播

对于常量值可以在编译时求值，从而减少目标代码的运算次数，对于后续引用的常量值，都可以直接用常量替换变量。例如

```
a = 1;
b = a + a;
c = b;
```

通过传播可以直接得到$c = 2$而省去了中间的操作。对于不被修改的常量，能利用替换的地方都尽量使用替换。