编译原理实验报告

2022111189 李博文

1. 词法分析器的实现

本实验使用 Flex 工具生成词法分析器。词法分析程序（lex 文件）包含定义部分、规则部分和用户代码部分。定义部分主要包含所需的 C 头文件、宏定义和全局变量声明。规则部分是核心，包含若干“正则表达式 + 动作”的规则。扫描到匹配的词素时执行对应动作，创建相应的语法树终结符节点并返回相应的 token 给语法分析器。

```c
// 创建节点
Node *createNode_lex(char *name, int lineno, void *val) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (!node) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }

    node->name = strdup(name);
    node->child = NULL;
    node->brother = NULL;

    if (strcmp(name, "ID") == 0 || strcmp(name, "TYPE") == 0) {
        node->strVal = strdup((char *)val);
    } else if (strcmp(name, "INT") == 0) {
        node->intVal = *((int *)val);
    } else if (strcmp(name, "FLOAT") == 0) {
        node->floatVal = *((float *)val);
    } else {
        node->lineno = lineno;  // 非终结符或无值的词法符号使用行号
    }

    return node;
}
```

用户代码部分实现了语法树终结符节点的创建函数：新建节点时将其兄弟和子节点指针初始化为 NULL，并根据终结符类型执行不同的初始化操作以保存对应的值（例如对 ID 类型终结符保存其标识符名称，Node 结构体定义在 node.h）。此外，还编写了词法分析阶段的错误输出函数。在 lexical.l 的规则部分，每当识别到一个符合规则的词素，立即创建相应终结符节点并返回 token。

```
FLOAT   {INT}\.[0-9]+
ID      [_a-zA-Z][0-9_a-zA-Z]*
RELOP   >|<|>=|<=|==|!=
TYPE    int|float
```

```
{INT}   { int val = atoi(yytext); yylval.node = createNode_lex("INT", yylineno, &val); return INT; }
{FLOAT} { float fval = atof(yytext); yylval.node = createNode_lex("FLOAT", yylineno, &fval); return FLOAT; }
{RELOP} { yylval.node = createNode_lex("RELOP", yylineno, NULL); return RELOP; }
{TYPE}  {  yylval.node = createNode_lex("TYPE", yylineno, strdup(yytext)); return TYPE; }
```

2. 语法分析器的实现

Bison 工具用于生成语法分析器。其文件结构与 Flex 类似，包括用户定义、token 声明和规则定义部分。本实验中主要采取了如下策略：

在用户定义部分，实现了语法树非终结符节点的创建函数。由于非终结符节点不需存储具体值，仅记录节点所在行号即可。

通过 Bison 的 %token 和 %type 声明，将所有终结符（词法单元）和非终结符的语义值类型统一定义为 Node*。例如，终结符包括数据类型、标识符、运算符、关键字和符号等；非终结符包括 Program、ExtDefList、Specifier 等。这种统一的类型设置确保词法分析返回的 token 能被语法分析正确识别，且语法树构建过程中各节点信息能正确传递和挂接。

使用 %right、%left、%nonassoc 等声明定义运算符的优先级和结合性，解决表达式解析中的移进/归约冲突。

针对每条产生式，都编写了相应的语义动作来构建并连接语法树节点。首先创建左部非终结符节点，将其第一子节点设为产生式右部的第一个符号，然后通过兄弟指针依次链接其余子节点。对于可重复的列表结构，通过递归定义产生式（右部再次出现该非终结符）实现多个子节点的挂接。

为处理语法错误，定义了专门的错误产生式，在动作中调用 yyerrok 进行错误恢复，避免一次错误导致后续解析无法继续。

```
// 创建语法树节点
Node *createNode(char *name, int lineno) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (!node) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(EXIT_FAILURE);
        }
    node->name = strdup(name);
    node->lineno = lineno;
    node->child = NULL;
    node->brother = NULL;
    return node;
}
```

```
ExtDecList: VarDec{$$ = createNode("ExtDecList", @1.first_line); $$->child = $1;}
     |VarDec COMMA ExtDecList {
            $$ = createNode("ExtDecList",@1.first_line);
            $$->child = $1;
            $1 -> brother = $2;
            $2 -> brother = $3;
                              }
     |VarDec error ExtDecList {
            yyerrok;
     }
     ;
```

```
%token <node> INT
%token <node> FLOAT
%token <node> ID
%token <node> RELOP TYPE ASSIGNOP PLUS MINUS STAR DIV
%token <node> IF ELSE WHILE RETURN STRUCT
%token <node> AND OR NOT SEMI COMMA LP RP LC LB RB RC DOT
```

```
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right LOWER_THAN_NOT NOT
%right DOT LP LB RP RB
```

```
// 处理 if else 移进/规约冲突
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

3. 报错处理

采用 Bison 提供的 yyerror 函数统一报告词法和语法错误。当检测到错误时，yyerror 会输出包含错误类型和行号的信息。另外，使用 lastErrorLine 变量记录最近一次出错的行号，确保同类错误只报告一次。发生错误时，将不输出语法树结果。

```c
// 递归打印语法树
void printTree(Node *node, int indent) {
    if (!node) return;

    for (int i = 0; i < indent; ++i) printf("  ");

    if (!node->child) {
        // 是词法单元（终结符）
        if (strcmp(node->name, "ID") == 0 || strcmp(node->name, "TYPE") == 0) {
            printf("%s: %s\n", node->name, node->strVal);
        } else if (strcmp(node->name, "INT") == 0) {
            printf("%s: %d\n", node->name, node->intVal);
        } else if (strcmp(node->name, "FLOAT") == 0) {
            printf("%s: %f\n", node->name, node->floatVal);
        } else {
            printf("%s\n", node->name);  // 其他终结符，如 SEMI, LP 等
        }
    } else {
        // 是语法单元（非终结符）
        printf("%s (%d)\n", node->name, node->lineno);
    }

    printTree(node->child, indent + 1);
    printTree(node->brother, indent);
}
%}
```

4. 递归打印语法树

实现了一个递归遍历语法树的输出功能。根据节点类型选择相应的打印格式，并先序遍历整棵语法树输出各节点信息，确保输出格式符合规范要求（正确的缩进层次、八进制和十六进制数的格式转换等）。

```c
typedef struct Node {
    char *name;
    struct Node *child, *brother;
    union {
        int lineno;
        int intVal;
        float floatVal;
        char *strVal;
    };
} Node;
```

5. 语法树节点设计

语法树节点的数据结构设计是本项目的一个特点。通过在 Node 结构体中使用共用体（union）来区分不同类型的节点，并采用“左孩子-右兄弟”链表形式表示树的层次结构。这样的设计简化了节点定义和管理，使语法树的构建和遍历更加高效。

6. 项目编译

本项目使用 CMake/Clion 工具管理编译。在项目根目录下执行 CMake 配置并构建工程即可完成编译。（需确保已安装 GCC 编译器以及 Flex 和 Bison 工具，并配置相应的环境变量。）