实验六+七

编译器设计专题实验（六）：语义分析实验报告

1. 实验目的

设计和实现一个符号表。符号表用于在编译过程中存储和检索变量名、函数名及其相关属性，并在语法和语义分析阶段进行正确性检查。

2. 实验内容

1. **词法分析**：在遇到一个新声明的变量名或函数名时，将其添加到符号表。
2. **语法分析**：填充符号表的相关信息。
3. **语义分析**：依据符号表进行语义正确性检查，验证代码的正确性。

3. 实验步骤

1. **定义数据结构**：
   - `Symbol` 结构体表示符号的属性。
   - `FunctionSymbolTable` 结构体表示函数符号表，包含函数名、返回类型、层级、外部链接、帧大小以及符号列表。
   - `SymbolTableManager` 类管理多个函数符号表，提供添加符号和显示符号表的方法。
2. **解析代码**：
   - 使用 `processCode` 函数解析输入的代码字符串。
   - 使用正则表达式匹配函数声明和变量声明。
   - 对于匹配到的函数声明，创建对应的函数符号表并提取参数信息。
   - 对于匹配到的变量声明，添加到当前函数符号表中。
3. **代码逻辑**：
   - `main` 函数创建 `SymbolTableManager` 实例。
   - 读取输入的代码字符串，调用 `processCode` 解析代码，并显示符号表内容。

4. 实验输入与输出

- **输入**：一段代码。
- **输出**：符号表及其内容，包括变量、数组和函数的类型和作用域等信息。

5. 代码实现及分析

1. `Symbol` 结构体

```
struct Symbol {
    string label;
    string name;
    string kind;
    int level;
    string outer;
    string type;
    bool para;
    bool ref;
    string seg;
    string link;

    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    Symbol(string l, string n, string k, int lv, string o, string t, bool p, bool r, string s, string lk)
        : label(l), name(n), kind(k), level(lv), outer(o), type(t), para(p), ref(r), seg(s), link(lk) {}
};
```

分析：`Symbol` 结构体定义了一个符号的各个属性，例如标签、名称、种类、层级、外部链接、类型、是否为参数、是否为引用、段和链接等信息。构造函数初始化了这些属性。

2. `FunctionSymbolTable` 结构体

```
struct FunctionSymbolTable {
    string func_name;
    string func_return_type;
    int func_level;
    string func_outer;
    int size_of_frame;
    vector<Symbol> symbols;

    FunctionSymbolTable() = default;
    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    FunctionSymbolTable(string name, string ret_type, int level, string outer, int frame_size)
        : func_name(name), func_return_type(ret_type), func_level(level), func_outer(outer), size_of_frame(frame_size) {}

    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    void addSymbol(const Symbol& symbol) {
        symbols.push_back(symbol);
    }

    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    void display() const {
        cout << "func_name: " << func_name << endl;
        cout << "func_return_type: " << func_return_type << endl;
        cout << "func_level: " << func_level << endl;
        cout << "func_outer: " << func_outer << endl;
        cout << "size_of_frame: " << size_of_frame << endl;
        cout << "variables and parameters:" << endl;
        cout << "label\tname\tkind\tlevel\touter\ttype\tpara\tref\tseg\tlink" << endl;
        for (const auto& s : symbols) {
            cout << s.label << "\t" << s.name << "\t" << s.kind << "\t"
                 << s.level << "\t" << s.outer << "\t" << s.type << "\t"
                 << (s.para ? "par" : "var") << "\t"
                 << (s.ref ? "ref" : "-1") << "\t" << s.seg << "\t" << s.link << endl;
        }
        cout << endl;
    }
};
```

分析：`FunctionSymbolTable` 结构体定义了一个函数符号表的各个属性，例如函数名、返回类型、层级、外部链接、帧大小和包含的符号（变量和参数）列表。提供了添加符号和显示符号表内容的方法。

3. `SymbolTableManager` 类

```
class SymbolTableManager {
private:
    unordered_map<string, FunctionSymbolTable> tables;
    string current_function;
    int current_level = 0;

public:
    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    void createFunctionTable(string name, string ret_type, int level, string outer, int frame_size) {
        tables[name] = FunctionSymbolTable(name, ret_type, level, outer, frame_size);
        current_function = name;
        current_level = level;
    }

    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    void addSymbol(string label, string name, string kind, int level, string outer, string type, bool para, bool ref, string seg, string link) {
        if (tables.find(current_function) != tables.end()) {
            tables[current_function].addSymbol(Symbol(label, name, kind, level, outer, type, para, ref, seg, link));
        }
    }

    函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
    void display() const {
        for (const auto& entry : tables) {
            entry.second.display();
        }
    }
};
```

分析：`SymbolTableManager` 类管理多个函数符号表，提供创建函数符号表、添加符号和显示所有符号表的方法。使用 `unordered_map` 来存储函数名和对应的符号表，跟踪当前正在处理的函数和层级。

4. `isValidIdentifier` 函数

```
bool isValidIdentifier(const string& str) {
    if (str.empty() || (!isalpha(str[0]) && str[0] != '_'))
        return false;
    return all_of(str.begin() + 1, str.end(), [](char c) {
        return isalnum(c) || c == '_';
    });
}
```

分析：`isValidIdentifier` 函数检查给定字符串是否是一个有效的标识符。有效的标识符必须以字母或下划线开头，后续字符可以是字母、数字或下划线。

5. `processCode` 函数

```
void processCode(const string& code, SymbolTableManager &manager) {
    stringstream ss(code);
    string line;
    regex func_decl_regex(R"(\b(\w+)\s+(\w+)\s*\(([^)]*)\))");
    regex var_decl_regex(R"((const\s+)?(\w+)\s+(\w+)(\[\d*\])?)");
    smatch match;
    int current_level = 0;
    string current_function = "global";
    manager.createFunctionTable("global", "void", current_level, "", 0); // 假设全局作用域

    while (getline(ss, line)) {
        line = regex_replace(line, regex("//.*"), ""); // 去除单行注释
        line = regex_replace(line, regex("/\\*.*\\*/"), ""); // 去除多行注释

        if (regex_search(line, match, func_decl_regex)) {
            string ret_type = match[1];
            string func_name = match[2];
            string params = match[3];
            current_function = func_name;
            current_level++;
            manager.createFunctionTable(func_name, ret_type, current_level, "global", 0); // 假设函数在全局作用域

            stringstream param_stream(params);
            string param;
            while (getline(param_stream, param, ',')) {
                param = regex_replace(param, regex("^\\s+|\\s+$"), ""); // 去除前后空格
                if (regex_search(param, match, var_decl_regex)) {
                    string type = match[2];
                    string name = match[3];
                    manager.addSymbol(name, name, "parameter", current_level, current_function, type, true, false, "data", "");
                }
            }
        } else if (regex_search(line, match, var_decl_regex)) {
            string type = match[2];
            string name = match[3];
            manager.addSymbol(name, name, "variable", current_level, current_function, type, false, false, "data", "");
        }
    }
}
```

分析：`processCode` 函数解析输入的代码字符串，提取函数和变量的声明。使用正则表达式匹配函数声明和变量声明。对于匹配到的函数声明，创建对应的函数符号表并提取参数信息。对于匹配到的变量声明，添加到当前函数符号表中。

6. `main` 函数

```
int main() {
    SymbolTableManager manager;
    string code = R"(
        int main() {
            int a;
            float b;
        }

        void func(int x, char y) {
            int z;
        }
    )";
    processCode(code, manager);
    manager.display();
    return 0;
}
```

分析：`main` 函数创建一个 `SymbolTableManager` 实例，输入一段示例代码，调用 `processCode` 函数解析代码，并显示符号表内容。

6. 实验结果

运行上述代码后，生成的结果

```
func_name: func
func_return_type: void
func_level: 2
func_outer: global
size_of_frame: 0
variables and parameters:
label   name    kind    level   outer   type    para    ref     seg     link
x       x       parameter       2       func    int     par     -1      data
y       y       parameter       2       func    char    par     -1      data
z       z       variable        2       func    int     var     -1      data

func_name: global
func_return_type: void
func_level: 0
func_outer:
size_of_frame: 0
variables and parameters:
label   name    kind    level   outer   type    para    ref     seg     link

func_name: main
func_return_type: int
func_level: 1
func_outer: global
size_of_frame: 0
variables and parameters:
label   name    kind    level   outer   type    para    ref     seg     link
a       a       variable        1       main    int     var     -1      data
b       b       variable        1       main    float   var     -1      data
```

7. 实验总结

本实验通过设计和实现符号表，完成了对代码的语义分析。符号表的实现包括了符号的添加和查找功能，以及对函数和变量声明的解析。通过本实验，理解了编译器设计中的符号表管理和语义分析的基本原理，并掌握了基本的实现方法。

如果有进一步的需求或改进建议，可以继续完善符号表的数据结构和语义分析器的功能，以处理更复杂的编译场景。

实验报告：编译器专题实验（七）

1.实验目的

将语义分析输出的符号表映射为内存映像，并生成依赖于栈帧的目标代码，将结果输出到文件中。

2.实验内容

1. **栈帧设计**：包括设计D表。
2. **序言、尾声、调用序列、返回序列构建。**
3. **名引用的代码变换**：引用序列构建。
4. **目标语言指令模板**：使用MIPS指令集。

3.实验步骤

1. **符号表创建**：
    - 为每个函数创建一个符号表，记录函数的形式参数和局部变量（包括所有的名字）。
    - 主程序作为一个大的函数处理。

2. **输入处理**：
    - 读取一段代码，并生成四元式作为输入。
3. **输出处理**：
    - 根据输入的四元式生成相应的汇编语言。

4.代码实现与分析

这段代码实现了一个简单的四元式到汇编代码转换程序。以下是对这段代码的简要分析：

结构体定义

```
struct Quadruple {
    std::string operation;
    std::string arg1;
    std::string arg2;
    std::string result;
};
```

分析：
定义了一个 `Quadruple` 结构体来表示四元式。每个四元式包含四个部分：

- `operation`：操作符（如 `+`, `-`, `*`, `/`, `=`）。
- `arg1`：第一个操作数。
- `arg2`：第二个操作数（如果适用）。
- `result`：结果变量。

汇编代码生成函数

```
std::string generateAssembly(const Quadruple &quad) {
    if (quad.operation == "+") {
        return "MOV EAX, " + quad.arg1 + "\nADD EAX, " + quad.arg2 + "\nMOV " + quad.result + ", EAX";
    } else if (quad.operation == "-") {
        return "MOV EAX, " + quad.arg1 + "\nSUB EAX, " + quad.arg2 + "\nMOV " + quad.result + ", EAX";
    } else if (quad.operation == "*") {
        return "MOV EAX, " + quad.arg1 + "\nIMUL EAX, " + quad.arg2 + "\nMOV " + quad.result + ", EAX";
    } else if (quad.operation == "/") {
        return "MOV EAX, " + quad.arg1 + "\nCDQ\nIDIV " + quad.arg2 + "\nMOV " + quad.result + ", EAX";
    } else if (quad.operation == "=") {
        return "MOV " + quad.result + ", " + quad.arg1;
    } else {
        return "; Unsupported operation: " + quad.operation;
    }
}
```

分析：
`generateAssembly` 函数将四元式转换为对应的汇编代码。支持的操作包括 `+`, `-`, `*`, `/` 和 `=`。根据操作符类型，生成相应的汇编指令，并返回作为字符串。

打印汇编代码函数

```
void printAssembly(const std::vector<Quadruple> &quadruples) {
    for (const auto &quad : quadruples) {
        std::cout << generateAssembly(quad) << std::endl;
    }
}
```

分析：

`printAssembly` 函数遍历所有四元式，并调用 `generateAssembly` 生成对应的汇编代码，然后打印到标准输出。

解析四元式函数

```
std::vector<Quadruple> parseQuadruples(const std::string &input) {
    std::vector<Quadruple> quadruples;
    std::istringstream iss(input);
    std::string line;

    while (std::getline(iss, line)) {
        std::istringstream quadStream(line);
        Quadruple quad;
        quadStream >> quad.operation >> quad.arg1 >> quad.arg2 >> quad.result;
        quadruples.push_back(quad);
    }

    return quadruples;
}
```

分析：

`parseQuadruples` 函数解析输入字符串中的四元式。它使用字符串流逐行读取输入，并将每行解析为一个 `Quadruple` 结构体，然后添加到 `quadruples` 向量中。

主函数

```
int main() {
    std::string input;
    std::cout << "输入四元式 (operation arg1 arg2 result)，每行一个，输入END结束：" << std::endl;
    std::getline(std::cin, input, '\0');  // Read all input until EOF

    std::vector<Quadruple> quadruples = parseQuadruples(input);

    std::cout << "生成的汇编代码：" << std::endl;
    printAssembly(quadruples);

    return 0;
}
```

对于输入a+b+c+(a*a)

输出：

```
MOV EAX, a
IMUL EAX, a
MOV t1, EAX
MOV EAX, a
ADD EAX, b
MOV t2, EAX
MOV EAX, t2
ADD EAX, c
MOV t3, EAX
MOV EAX, t3
ADD EAX, t1
MOV result, EAX
```

分析：

1. 提示用户输入四元式，每行一个，直到输入 `END` 结束。
2. 使用 `getline` 函数读取所有输入（直到 EOF）。
3. 调用 `parseQuadruples` 函数解析输入字符串为四元式列表。
4. 调用 `printAssembly` 函数生成并打印汇编代码。

总结

该程序通过以下步骤实现了从四元式到汇编代码的转换：

1. 定义四元式结构体。
2. 实现生成汇编代码的函数。
3. 实现解析输入四元式的函数。
4. 主函数读取用户输入，解析为四元式，并生成和打印相应的汇编代码。

6.实验结论

通过本次实验，成功实现了从符号表到目标代码的映射，生成了符合MIPS指令集的汇编代码。代码逻辑清晰，易于扩展，可以进一步完善和优化。