

Project 2: 语义分析 (Semantic Analysis) 实验报告

学号: 2023211603

姓名: 王何佳

Docker ID:

09fd2637167d30b23842e962644ffd6b0b73c698eb8fc45bfcddef1369076084

1. 实验概述

本次实验的主要目标是在词法分析和语法分析的基础上, 实现 BPL(Basic Programming Language)的语义分析器。与之前的语法分析不同, 语义分析旨在通过对抽象语法树(AST)的遍历, 确保程序具有明确且合法的语义含义。

我的实现主要完成了以下核心任务:

- 符号表管理: 设计并维护符号表以处理变量、函数和结构体的定义与作用域。
- 类型检查: 验证表达式、赋值、返回值和函数调用中的类型一致性。
- 错误报告: 能够检测并准确报告项目文档中规定的 15 种语义错误。
- Bonus 功能: 实现了结构等价性(Structural Equivalence)检查和基础类型的隐式类型转换(Implicit Casting)。

2. 系统设计与数据结构

为了有效地进行语义分析, 我设计了两个核心模块: 类型系统(type.c)和符号表(symtab.c)。

2.1. 类型系统表示(type.h)

我使用递归结构体 Type 来表示 BPL 中的各种数据类型(基本类型、数组、结构体、函数)。这种设计允许处理复杂的数据结构, 例如“结构体的数组”或“包含数组的结构体”。

```
typedef struct Type {  
    enum { BASIC, ARRAY, STRUCTURE, FUNCTION } kind;  
    union {
```

```

        int basic; // 0 for int, 1 for float
        struct { struct Type* elem; int size; } array;
        struct { char* name; struct FieldList* member; } structure; // 结构体链表
        struct { struct Type* returnType; struct FieldList* params; int paramNum; }
function;
    } u;
} Type;

```

在此模块中，`checkType` 函数是核心，用于递归地检查两个类型是否等价。在标准实现中，对于结构体采用了名等价(Named Equivalence)策略。

2.2. 符号表设计(symtab.c)

符号表是连接声明与使用的桥梁。我采用哈希表(Hash Table)结合作用域栈(Scope Stack)的方式来实现。

哈希算法：使用高效的 `hash_pjw` 算法将标识符名称映射到索引。

作用域管理：

- 进入作用域：增加当前深度计数器。
- 退出作用域：利用 `scopeStack` 记录当前层级定义的所有符号。当退出作用域时，遍历该层级链表，将这些符号从全局哈希表中逻辑移除（恢复同名变量的外层定义），从而高效实现作用域的嵌套与变量遮蔽。

关键代码：退出作用域逻辑

```

// symtab.c
void exitScope() {
    if (currentDepth < 0) return;
    HashNode* node = scopeStack[currentDepth];
    // 遍历当前层级定义的所有符号
    while (node != NULL) {
        unsigned int index = hash_pjw(node->name);
        // 从哈希表链中移除该节点，恢复外层定义
        HashNode* head = symbolTable[index];
        if (head == node) {
            symbolTable[index] = node->next;
        } else {
            // ... (链表删除节点的逻辑)
        }
        node = node->stack_next; // 处理栈中的下一个符号
    }
}

```

```
    }  
    currentDepth--;  
}
```

3. 语义分析策略与测试用例分析

语义分析通过对 AST 进行后序遍历完成(semantic.c)。为了清晰地说明应对策略，我将 15 种错误类型按功能分组进行解析。

3.1. 作用域与定义检查 (Errors 1, 3, 4, 15)

测试目标：确保变量、函数和结构体在使用前已定义，且在同一作用域内不重复定义。

- 在进入函数体或复合语句(CompSt)时调用 enterScope()，退出时调用 exitScope()。
- 在处理声明节点(ExtDecList,VarDec)时，使用 lookupSymbol(仅检查当前层)确认是否重复定义(Error 3,15)。
- 在处理使用节点(Exp->ID)时，lookupSymbol 会自动由内向外查找。若返回 NULL，则报“变量未定义”错误(Error 1)。

关键代码：变量声明与重定义检查

```
// semantic.c -> ExtDecList  
void ExtDecList(struct TreeNode* node, Type* type) {  
    Type* finalType = NULL;  
    // 提取变量名和类型（处理数组定义）  
    char* name = check_VarDec(node->children[0], type, &finalType);  
    // 检查是否重定义（对应 Error Type 3）  
    if (strcmp(name, "unknown") != 0) {  
        if (!insertSymbol(name, finalType)) {  
            // insertSymbol 若发现当前作用域已存在同名符号则返回 0  
            semantic_error(3, node->lineno, "redefine variable: %s", name);  
        }  
    }  
    // ... 递归处理后续变量  
}
```

3.2. 类型安全与赋值检查 (Errors 5, 6, 7)

测试目标：确保赋值号左侧为左值(L-value)，且运算符两侧类型匹配。

- 左值检查(Error 6)：在 `Exp->Exp ASSIGN Exp` 产生式中，检查左侧子节点是否为 ID、数组访问(LB)或结构体访问(DOT)。其他情况（如常数、运算结果）均为右值。
- 类型匹配(Error 5)：递归计算左右两侧的 `Type`，调用 `checkType` 确认一致性。
- 操作数检查(Error 7)：对于算术运算（PLUS,MUL 等），强制要求两侧必须为 BASIC 类型（int 或 float）且类型相同。

关键代码：赋值表达式处理

```
// semantic.c -> Exp 函数片段
if (node->child_num == 3 && strcmp(node->children[1]->name, "ASSIGN") == 0) {
    struct TreeNode* lhs = node->children[0];
    // 判断左值 (Error Type 6)
    int isLVal = 0;
    // 逻辑：检查产生式是否为 ID, Exp[Exp], 或 Exp.ID
    if (lhs->child_num == 1 && strcmp(lhs->children[0]->name, "ID") == 0) isLVal = 1;
    else if (lhs->child_num == 4 && strcmp(lhs->children[1]->name, "LB") == 0)
        isLVal = 1;
    else if (lhs->child_num == 3 && strcmp(lhs->children[1]->name, "DOT") == 0)
        isLVal = 1;
    if (!isLVal) {
        semantic_error(6, node->lineno, "rvalue on the left side of assignment operator");
        return NULL;
    }
    // 类型匹配检查 (Error Type 5)
    Type* t1 = Exp(lhs);
    Type* t2 = Exp(node->children[2]);
    if (t1 && t2 && !checkType(t1, t2)) {
        semantic_error(5, node->lineno, "unmatching type on both sides of assignment");
    }
    return t1;
}
```

3.3. 函数调用检查 (Errors 2, 8, 9, 11)

测试目标：验证函数存在性、参数列表匹配度及返回值类型。

- 定义时：将函数签名（返回值类型+参数链表）存入符号表。
- 调用时(Args)：取出符号表中的参数定义，与传入的实参列表同步遍历。对每一对参数调用 `checkType`，若数量或类型不符则报 Error 9。
- 非函数调用(Error 11)：如果对普通变量使用了 `()` 操作符，检查其类型是否为 FUNCTION。

关键代码：实参与形参匹配

```
// semantic.c -> Args
void Args(struct TreeNode* node, FieldList* paramDef) {
    Type* t = Exp(node->children[0]); // 当前实参的类型
    // 检查参数定义是否已耗尽 (参数数量不匹配)
    if (!paramDef) {
        semantic_error(9, node->lineno, "arguments mismatch...");
        return;
    }
    // 检查参数类型是否一致
    if (t && !checkType(t, paramDef->type)) {
        semantic_error(9, node->lineno, "arguments mismatch...");
        return;
    }
    // 递归检查下一个参数
    if (node->child_num == 3) Args(node->children[2], paramDef->next);
}
```

3. 4. 复杂类型访问 (Errors 10, 12, 13, 14)

测试目标：验证数组索引和结构体成员访问的合法性。

- 数组：确保被索引的对象是 ARRAY 类型(Error 10)，且索引表达式 `Exp` 的类型必须是 `int`(Error 12)。
- 结构体：确保操作符左侧是 STRUCTURE 类型(Error 13)。然后遍历该结构体的 `member` 链表查找成员名，若未找到则报 Error 14。

关键代码：结构体成员访问

```
// semantic.c -> Exp (DOT case)
if (node->child_num == 3 && strcmp(node->children[1]->name, "DOT") == 0) {
    Type* base = Exp(node->children[0]);
    char* field = node->children[2]->value.str_val; // 成员名
```

```

// 检查基底是否为结构体 (Error 13)
if (!base || base->kind != STRUCTURE) {
    semantic_error(13, node->lineno, "accessing member of non-structure
variable");
    return NULL;
}
// 遍历成员链表查找 (Error 14)
FieldList* f = base->u.structure.member;
while(f) {
    if (strcmp(f->name, field) == 0) return f->type; // 找到成员
    f = f->next;
}
semantic_error(14, node->lineno, "accessing an undefined structure member");
return NULL;
}

```

4. Bonus 功能实现

我在标准要求的基础上进行了扩展，支持了更高级的语义特性。这些代码位于 `bonus_src` 目录下。

4. 1. 结构等价

设计原理：标准实验要求使用名等价，即两个结构体必须名称相同才视为同一类型。我修改了 `checkType` 函数，实现了结构等价。

实现细节：当比较两个 `STRUCTURE` 类型时，不再比较 `name`，而是同时遍历两个结构体的成员链表(`member`)。如果所有对应位置的成员类型都相同（递归定义），则认为两个结构体类型兼容。

测试样例(`test_2_bonus_struct.bpl`):

```

struct A { int x; float y; };
struct B { int a; float b; };
// A 和 B 名称不同，但内部结构一致。在 Bonus 版本中，sa = sb 合法。
sa = sb;

```

4. 2. 隐式类型转换

设计原理：标准 BPL 严格禁止 int 和 float 混用。我实现了类型提升逻辑。

实现细节：在 Exp 函数处理算术运算和赋值时：

1. 若运算符两侧分别为 INT 和 FLOAT，则结果类型“提升”为 FLOAT，不报错。
2. 允许将 INT 类型的表达式赋值给 FLOAT 类型的变量。

测试样例(test_2_bonus_cast.bpl)：

```
int i = 1;
float f = 2.5;
float res;
res = i + f; // i 被隐式转换为 float，结果为 float，赋值合法
```

4. 3. Bonus 代码使用与验证

我编写了 Makefile 和测试脚本，通过替换源文件的方式启用 Bonus 逻辑。

```
# 备份标准代码并覆盖
cp semantic.c semantic.c.bak
cp type.c type.c.bak
cp bonus_src/semantic.c .
cp bonus_src/type.c .
# 编译并运行测试
make clean && make
./bin/bplc test/test_2_bonus_struct.bpl
./bin/bplc test/test_2_bonus_cast.bpl
```

测试结果显示，这两个测试用例在 Bonus 模式下均成功编译且无报错，验证了功能的正确性。

5. 实验结果详解

为了验证语义分析器的正确性与鲁棒性，我使用实验指导书提供的标准测试集（test_2_r01 至 test_2_r15）以及我自行编写的 Bonus 测试集进行了全面测试。

5. 1. 标准测试集验证

标准测试集覆盖了项目文档要求的全部 15 种错误类型。我对每一个测试用例进

行了执行，并将实际输出与预期语义规则进行了比对。

```
root@09fd2637167d:/app# ./bin/bplc test/test_2_o01.bpl
root@09fd2637167d:/app# ./bin/bplc test/test_2_r01.bpl
Error type 1 at Line 5: undefined variable: x3
root@09fd2637167d:/app# ./bin/bplc test/test_2_r02.bpl
Error type 2 at Line 8: undefined function: add
root@09fd2637167d:/app# ./bin/bplc test/test_2_r03.bpl
Error type 3 at Line 6: redefine variable: ttt
root@09fd2637167d:/app# ./bin/bplc test/test_2_r04.bpl
Error type 4 at Line 7: redefine function: compare
root@09fd2637167d:/app# ./bin/bplc test/test_2_r05.bpl
Error type 5 at Line 6: unmatching type on both sides of assignment
root@09fd2637167d:/app# ./bin/bplc test/test_2_r06.bpl
Error type 6 at Line 6: rvalue on the left side of assignment operator
root@09fd2637167d:/app# ./bin/bplc test/test_2_r07.bpl
Error type 7 at Line 10: unmatching operands
Error type 5 at Line 10: unmatching type on both sides of assignment
root@09fd2637167d:/app# ./bin/bplc test/test_2_r08.bpl
Error type 8 at Line 21: return value type mismatches the declared type
root@09fd2637167d:/app# ./bin/bplc test/test_2_r09.bpl
Error type 9 at Line 12: arguments mismatch the declared parameters
root@09fd2637167d:/app# ./bin/bplc test/test_2_r10.bpl
Error type 10 at Line 17: applying indexing operator on non-array type variables
root@09fd2637167d:/app# ./bin/bplc test/test_2_r11.bpl
Error type 11 at Line 16: applying function invocation operator on non-function names
root@09fd2637167d:/app# ./bin/bplc test/test_2_r12.bpl
Error type 12 at Line 15: array indexing with non-integer type expression
root@09fd2637167d:/app# ./bin/bplc test/test_2_r13.bpl
Error type 13 at Line 19: accessing member of non-structure variable
Error type 13 at Line 19: accessing member of non-structure variable
root@09fd2637167d:/app# ./bin/bplc test/test_2_r14.bpl
Error type 14 at Line 10: accessing an undefined structure member
Error type 14 at Line 12: accessing an undefined structure member
root@09fd2637167d:/app# ./bin/bplc test/test_2_r15.bpl
Error type 15 at Line 6: redefine the same structure type
```

以下结果均基于 Docker 容器内的实际运行截图。

| 测试用例文件 | 测试点描述 | 预期错误 | 实际输出 | 判定 |
|----------------|---------------------|--------|---|------|
| test_2_o01.bpl | 合法代码测试 | 无 | (无输出) | Pass |
| test_2_r01.bpl | 使用未定义的变量 x3 | Type 1 | Error type 1 at Line 5: undefined variable: x3 | Pass |
| test_2_r02.bpl | 调用未定义的函数 add | Type 2 | Error type 2 at Line 8: undefined function: add | Pass |
| test_2_r03.bpl | 变量 ttt 重复定义 | Type 3 | Error type 3 at Line 6: redefine variable: ttt | Pass |
| test_2_r04.bpl | 函数 compare 重复 定义 | Type 4 | Error type 4 at Line 7: redefine function: compare | Pass |

| | | | | |
|----------------|----------------------|------------|--|------|
| test_2_r05.bpl | 赋值号两侧类型不匹配 | Type 5 | Error type 5 at Line 6: unmatching type on both sides... | Pass |
| test_2_r06.bpl | 赋值号左侧出现右值 | Type 6 | Error type 6 at Line 6: rvalue on the left side... | Pass |
| test_2_r07.bpl | 操作数类型不匹配 &赋值类型不匹配 | Type 7 & 5 | Line 10 同时报告 Type 7 和 Type 5 | Pass |
| test_2_r08.bpl | 函数返回值类型不匹配 | Type 8 | Error type 8 at Line 21: return value type mismatches... | Pass |
| test_2_r09.bpl | 函数实参形参不匹配 | Type 9 | Error type 9 at Line 12: arguments mismatch... | Pass |
| test_2_r10.bpl | 对非数组变量使用 [] 操作符 | Type 10 | Error type 10 at Line 17: applying indexing operator... | Pass |
| test_2_r11.bpl | 对非函数变量使用 () 操作符 | Type 11 | Error type 11 at Line 16: applying function invocation... | Pass |
| test_2_r12.bpl | 数组索引非整数类型 | Type 12 | Error type 12 at Line 15: array indexing with non-integer... | Pass |
| test_2_r13.bpl | 对非结构体变量使用 . 操作符 | Type 13 | Error type 13 at Line 19: accessing member of non-structure... | Pass |
| test_2_r14.bpl | 访问未定义的结构体成员 | Type 14 | Error type 14 at Line 10/12: accessing an undefined... | Pass |
| test_2_r15.bpl | 结构体名重复定义 | Type 15 | Error type 15 at Line 6: redefine the same structure type | Pass |

5.2. 重点结果分析

在测试过程中，有几个用例的表现特别值得注意，证明了语义分析器的逻辑完整性。

级联错误处理（test_2_r07.bpl）

- 在测试用例 r07 中，代码行为 `aa.weight=aa+2;`。
- 分析：这里首先发生了结构体 `aa` 与整数 `2` 相加，触发了 **Type 7** 错误（操作数不匹配）。
- 结果：我的分析器在报告 **Type 7** 后，并没有停止工作，而是继续检查赋值操作。由于 `aa+2` 的结果类型（`undefined` 或 `int`）与左值 `aa.weight` 不匹配（或者因为之前的错误导致类型推断失败），分析器正确地继续报告了 **Type 5** 错误。
- 意义：这证明了我的分析器具有良好的错误恢复能力，能够在表达式中发现多个潜在的语义问题。

嵌套结构体与作用域（test_2_r13.bpl）

- 在测试用例 r13 中，涉及到了嵌套的成员访问。
- 输出：第 19 行报告了两次 **Type 13** 错误。
- 代码：`mother.cc.age=father.cc.age;`
- 分析：`mother.cc` 是 `int` 类型，对其访问 `.age` 触发了左侧的 **Error 13**；`father.cc` 也是 `int` 类型，对其访问 `.age` 触发了右侧的 **Error 13**。
- 意义：这表明 `Exp` 函数的递归调用逻辑正确，能够分别独立地检查赋值号左右两侧的子表达式。

5.3. Bonus 功能验证

为了验证扩展功能的有效性，我进行了对比测试。以下测试均基于 `bonus_src` 编译出的分析器。

```
root@09fd2637167d:/app# echo "--- Testing Structural Equivalence ---"
./bin/bplc test/test_2_bonus_struct.bpl
--- Testing Structural Equivalence ---
root@09fd2637167d:/app# echo "--- Testing Implicit Casting ---"
./bin/bplc test/test_2_bonus_cast.bpl
--- Testing Implicit Casting ---
```

5.3.1.结构等价性测试

测试文件：test_2_bonus_struct.bpl

场景：将结构体 B 的变量赋值给结构体 A 的变量。两者名称不同，但内部字段定义完全一致（均为 int 后跟 float）。

标准版表现：报错 Error type 5: unmatching type...（因为名等价检查失败）。

Bonus 版表现：编译通过，无报错。

结论：成功实现了结构等价性检查。

5.3.2.隐式类型转换测试

测试文件：test_2_bonus_cast.bpl

场景：执行 res_f=i+f;，其中 i 是 int，f 是 float。

标准版表现：报错 Error type 7: unmatching operands（严格要求类型一致）。

Bonus 版表现：编译通过，无报错。

结论：分析器成功识别了 int 到 float 的类型提升，并允许混合运算，符合更高级语言的特性。

6. 实验总结

本次实验中，我构建了一个功能完备的语义分析器。

- 通过哈希表与栈结合的符号表设计，实现了 $O(1)$ 的高效符号查找和正确的作用域嵌套处理。
- 通过递归的类型系统，解决了复杂的结构体与数组嵌套的类型检查问题。
- Bonus 部分的实现进一步加深了我对类型等价性理论（名等价 vs 结构等价）的理解。

整个分析器具有良好的模块化特性，代码逻辑清晰，能够准确拦截语义错误，为后续的代码生成阶段打下了坚实基础。