

Project 1:Lexical & Syntax Analysis 实验报告

学号: 2023211603

姓名: 王何佳

Docker ID:

d9cc3544d8b835b3f14ff68e5769047c0cb708d4b4984c4f6cae701add1a2871

1. 实验概述

1.1 实验目标

本次实验旨在为 BUPT Programming Language(BPL)实现一个完整的编译器前端，包含词法分析器（Lexer）和语法分析器（Parser）。

主要任务包括：

- 词法分析：**使用 GNU Flex 工具识别 BPL 的各类 Token（标识符、关键字、整形/浮点型常量、运算符等），并处理词法错误（Error Type A）。
- 语法分析：**使用 GNU Bison 工具基于上下文无关文法（CFG）构建语法树，解决二义性冲突，并利用 error token 实现错误恢复和语法错误报告（Error Type B）。
- 语法树构建：**在语法分析过程中生成抽象语法树（AST），并按指定格式打印。

1.2 额外功能

为了提升编译器的实用性和鲁棒性，我在基础要求之上额外实现了以下高级功能：

- 多格式数字支持：**八进制整数(如 0123)、科学计数法浮点数(如 1.2e-3)。
- 增强的控制流：**for 循环结构、break 和 continue 语句。
- 复合运算符：**支持+=,-=,*=,/=。
- 注释处理：**支持 C 语言风格的单行(//)和多行(/...*/)注释。
- 智能错误恢复：**基于 Panic Mode 的错误恢复机制，确保持续报错能力。

2. 编译器设计

2.1 词法分析设计

词法分析器定义在 `lex.l` 中。核心逻辑是通过正则表达式匹配输入流，生成 `Token` 并传递属性值 (`yylval`) 给语法分析器。为了追踪位置信息，利用 `YY_USER_ACTION` 宏实时更新行列号。

2.2 语法分析设计

语法分析器定义在 `syntax.y` 中。采用 `LALR(1)` 分析算法。使用 `struct TreeNode` 结构体，包含行号、节点类型 (`Token`/非终结符)、名称及子节点指针数组。在 `Bison` 的产生式动作中，通过 `createNode` 和 `addChild` 函数自底向上构建语法树。

3. 核心功能实现与测试用例分析

本章节将结合官方提供的测试用例 (`test_1_r01~test_1_r12`)，详细阐述关键功能的实现代码。

3.1 基础语法与流程控制 (`test_1_r01, r04, r05, r10`)

测试目标： 验证函数定义、`if-else`、`while`、`return` 及基本表达式的解析。

关键代码实现(`syntax.y`):

对于基础语句块 `Stmt` 和表达式 `Exp` 的定义，我严格遵循了 `BPL` 文法规范：

```
/* Stmt 语句的产生式 */
Stmt : Exp SEMI {
    $$ = createNode("Stmt", "", @$first_line);
    addChild($$, $1); addChild($$, $2);
}
| CompSt { ... }
| RETURN Exp SEMI { ... }
| IF LP Exp RP Stmt %prec LOWER_THAN_ELSE {
    /* 处理 If 语句 */
    $$ = createNode("Stmt", "", @$first_line);
    addChild($$, $1); addChild($$, $2); addChild($$, $3);
    addChild($$, $4); addChild($$, $5);
}
```

```

    }
    | IF LP Exp RP Stmt ELSE Stmt {
        /* 处理 If-Else, 利用优先级解决悬空 Else 问题 */
        ...
    }
    | WHILE LP Exp RP Stmt {
        /* 处理 While 循环 */
        ...
    };

```

使用%prec LOWER_THAN_ELSE 解决 if-else 的移进-归约冲突。

3.2 结构体与作用域(test_1_r03, r09)

测试目标：验证结构体定义、变量声明及数组访问。

关键代码实现(syntax.y):

结构体定义涉及嵌套的定义列表 DefList。

```

/* 结构体定义 */
StructSpecifier : STRUCT ID LC DefList RC {
    $$ = createNode("StructSpecifier", "", @$first_line);
    addChild($$, $1); addChild($$, $2);
    addChild($$, $3); addChild($$, $4); addChild($$, $5);
};

/* 数组访问 Exp -> Exp [ Exp ] */
Exp : Exp LB Exp RB {
    $$ = createNode("Exp", "", @$first_line);
    addChild($$, $1); addChild($$, $2);
    addChild($$, $3); addChild($$, $4);
};

```

3.3 词法错误处理(test_1_r02, r06, r11)

测试目标：识别非法字符（如\$）、非法标识符（数字开头）、非法十六进制数。

关键代码实现(lex.l):

为了精准报错，我没有只使用通配符“.”，而是编写了专门的“错误陷阱”正

则。

```
/* 捕获非法十六进制数，如 0x5gg */
0[xX][0-9a-fA-F]*[g-zG-Z]+[0-9a-fA-F]* {
    printf("Error type A at Line %d: Illegal hexadecimal number '%s'\n", yylineno,
yytext);
    yylval.node = NULL;
    return ID; // 返回 ID 防止 Parser 崩溃，或者直接不返回
}

/* 捕获以数字开头的非法标识符，如 3_id */
[0-9]+[a-zA-Z_]+ {
    printf("Error type A at Line %d: Invalid identifier '%s'\n", yylineno, yytext);
    return ID;
}

/* 通用错误捕获：未定义的符号 */
.{
    printf("Error type A at Line %d: Mysterious character '%s'\n", yylineno, yytext);
}
```

3.4 语法错误恢复(test_1_r02, r04, r08)

测试目标：验证缺失分号、缺失括号等语法错误，并测试 Panic Mode 恢复能力。

关键代码实现(syntax.y):

利用 Bison 的 error 标记，在语句级别和定义级别进行同步恢复。

```
/* 语句级错误恢复：遇到错误则跳过直到分号 */
Stmt : error SEMI {
    yyerrok; // 重置错误状态
    /* 不生成语法树节点，但允许编译器继续分析后续代码 */
}

/* 缺失右括号的恢复 */
| LP Exp error Stmt {
    printf("Error type B at Line %d: Missing closing parenthesis ')\n", @2.last_line);
}
```

3.5 十六进制与字符处理

测试目标： 正确识别十六进制整数（0xFF）和转义字符。

关键代码实现(`lex.l`):

在词法阶段直接完成数值转换，满足输出十进制的要求。

```
/* 十六进制整数识别与转换 */
0[xX][0-9a-fA-F]+ {
    char *endptr;
    // 将 16 进制字符串转换为 10 进制整数值
    long val = strtol(yytext, &endptr, 16);
    yyval.node = createTokenNode("INT", val, yylineno);
    return INT;
}
```

4. 进阶功能详解

本部分是我为提升编译器能力而额外实现的功能。

4.1 八进制与科学计数法支持

官方只要求十进制和十六进制，我额外增加了八进制和科学计数法支持，使 `Lexer` 更符合 C 语言标准。

代码实现(`lex.l`):

```
/* 八进制数：以 0 开头，后跟 0-7 */
0[0-7]+ {
    long val = strtol(yytext, NULL, 8); // 基数设为 8
    yyval.node = createTokenNode("INT", val, yylineno);
    return INT;
}

/* 科学计数法浮点数：如 1.2e-3 */
[0-9]+\.[0-9]+([eE][+-]?[0-9]+)? {
    float val = strtod(yytext, NULL);
    yyval.node = createFloatNode("FLOAT", val, yylineno);
    return FLOAT;
}
```

4.2 For 循环语句

实现了完整的 `for(init;cond;post)stmt` 结构。

代码实现(syntax.y):

由于标准 `for` 循环产生式右部包含较多符号，导致节点子节点数超过了 `tree.h` 默认的 8 个。我将 `MAX_CHILDREN` 宏调整为 16 以适配此结构。

```
/* 语法规则 */
Stmt : FOR LP Exp SEMI Exp SEMI Exp RP Stmt {
    $$ = createNode("Stmt", "", @$$.first_line);
    // 依次添加 9 个子节点: FOR, LP, Exp, SEMI, Exp, SEMI, Exp, RP, Stmt
    addChild($$, $1); addChild($$, $2); addChild($$, $3);
    addChild($$, $4); addChild($$, $5); addChild($$, $6);
    addChild($$, $7); addChild($$, $8); addChild($$, $9);
};
```

4.3 复合赋值与循环控制

支持`+=`,`-=`,`break`,`continue`。

代码实现:

Lexer:增加 `ADD_ASSIGN`,`BREAK`,`CONTINUE` 等 Token 定义。

Parser:

```
/* 复合赋值运算 */
Exp : Exp ADD_ASSIGN Exp {
    $$ = createNode("Exp", "", @$$.first_line);
    addChild($$, $1); addChild($$, $2); addChild($$, $3);
}

/* 循环控制语句 */
Stmt : BREAK SEMI {
    $$ = createNode("Stmt", "", @$$.first_line);
    addChild($$, $1); addChild($$, $2);
}
| CONTINUE SEMI { ... }
```

4.4 完整的注释处理

利用 Flex 的 `Start Conditions` 优雅地剔除注释，而不产生多余的 Token。

代码实现(lex.l):

```
%x COMMENT /* 定义 COMMENT 状态 */

%%

"//"      { char c; while((c=input()) != '\n' && c != 0); } /* 单行注释: 忽略直到换行 */

"/*"      { BEGIN(COMMENT); } /* 进入多行注释状态 */
<COMMENT>"*/" { BEGIN(INITIAL); } /* 遇到结束符, 回到初始状态 */
<COMMENT>\n  { /* 记录行号, 但不输出 */ }
<COMMENT>.   { /* 忽略注释内容 */ }
```

5. 实验结果验证

5.1 官方测试集验证

通过脚本批量运行 test_1_r01 至 test_1_r12, 结果显示:

输出格式: 完全符合缩进要求, 非终结符显示行号, Token 显示属性值。

错误识别:

- test_1_r02:成功报出 Line 4(\$)和 Line 8(@)的 Type A 错误, 以及 Line 7 的 Type B 错误。
- test_1_r11:成功识别 0x77G(非法十六进制)和\x6u(非法转义)。

数值转换: test_1_r12 中 0xFFF 正确输出为 INT:4095。

5.2 额外功能验证

为了全面验证上述额外功能, 我设计了两个专门的测试用例: test_extra_features.bpl 和 test_extra_features2.bpl。

5.2.1 验证注释与基础 For 循环

测试代码片段(test_extra_features.bpl):

```
int test_extra_features() {
```

```

int a = 0;
// This is a single line comment
/* This is a multi-line comment
   that spans two lines */

// Test for loop with compound assignment
for(a = 0; a < 10; a += 1) {
    a *= 2;
}
return a;
}

```

结果分析:

```

Stmt (9)
  FOR                                <-- [验证] FOR 循环节点正确生成
  LP
  Exp (9)
    ...
    ADD_ASSIGN                       <-- [验证] += 运算符被识别
    Exp (9)
      INT: 1
  RP
  Stmt (9)
    CompSt (9)
      LC
      StmtList (10)
        Stmt (10)
          Exp (10)
            ...
            MUL_ASSIGN <-- [验证] *= 运算符被识别
            Exp (10)
              INT: 2

```

从输出可以看出，单行和多行注释被 **Lexer** 成功过滤，未干扰语法分析。同时，FOR 循环结构解析正确，复合赋值运算符+=和*=均被正确识别。

5.2.2 验证数值解析与循环控制

测试代码片段(test_extra_features2.bpl):

```

int test_advanced_2() {

```



```

// 1. Octal test (0123 -> decimal 83)
int oct = 0123;
// 2. Scientific float test (1.2e2 -> 120.0)
float sci = 1.2e2;

int i;
for (i = 0; i < 10; i += 1) {
    if (i == 5) { continue; }
    if (i > 8) { break; }
}
return oct;
}

```

结果分析：

```

Dec (4)
  VarDec (4)
    ID: oct
  ASSIGN
  Exp (4)
    INT: 83          <-- [验证] 八进制 0123 正确转换为十进制 83
...
Dec (6)
  ...
  Exp (6)
    FLOAT: 120.000000 <-- [验证] 科学计数法 1.2e2 正确转换为浮点数 120.0
...
Stmt (11)
  CONTINUE          <-- [验证] CONTINUE 语句识别成功
  SEMI
...
Stmt (14)
  BREAK             <-- [验证] BREAK 语句识别成功
  SEMI

```

该测试证明了 **Lexer** 对扩展数值格式的支持完全符合预期，同时 **Parser** 能够正确处理循环体内部的控制流跳转语句。

6. 实验总结

本次实验通过 **Flex** 和 **Bison** 构建了一个功能完备的 **BPL** 编译器前端。

- 完成度：不仅通过了所有官方测试用例，还通过修改 `tree.h` 和扩展文法，实现了 `For` 循环、多格式数字支持等高级特性。
- 鲁棒性：实现了基于 **Panic Mode** 的错误恢复，保证了编译器在遇到错误时不会直接崩溃，能提供更 valuable 的调试信息。
- 收获：深入理解了正则匹配的优先级问题（如关键词与标识符的冲突）、**LALR(1)** 文法的二义性消除方法（优先级与结合性设置），以及自动机生成工具的底层原理。