

Name: 王何佳

Student ID: 2023211603

Class Number: 2023211804

Container ID: da9483a303782804c29135085670671bbd450780095c07fa61f3a70d0197f588

---

## Lab 3

### 一、实验目的

1. **深入理解语法分析:** 掌握语法分析在编译器中的核心地位与作用, 理解上下文无关文法 (Context-Free Grammar) 在描述程序设计语言结构时的重要性。
2. **学习并掌握 GNU Bison:** 熟悉语法分析器生成工具 Bison 的使用方法, 理解其 .y 文件的基本结构, 包括声明区、语法规则区和 C 代码区。
3. **实践 Flex 与 Bison 的协同工作:** 学习如何将词法分析器 Flex 与语法分析器 Bison 相结合, 构建一个可以协同工作的完整分析器前端, 理解它们之间通过 token 传递信息的机制。
4. **解决语法二义性问题:** 通过解决实际问题, 学习识别并解决语法规则中的二义性, 掌握改写文法以消除二义性的基本技巧。
5. **锻炼综合实践与调试能力:** 通过完成计算器和括号匹配两个任务, 锻炼分析、定位和解决编译、链接以及运行时错误的能力。

### 二、实验原理

本次实验的核心是语法分析 (Syntax Analysis), 它是编译流程中继词法分析之后的第二个阶段。

1. **上下文无关文法 (CFG):** 语法分析的基础是上下文无关文法。一个 CFG 由四个部分组成:
  - 终结符号 (Terminals): 构成字符串的基本符号, 即词法分析器提供的 Token。
  - 非终结符号 (Non-terminals): 代表语法结构的变量, 如 expression、statement 等。

- 产生式（Productions）：定义了非终结符号如何由终结符号和其他非终结符号构成，形式为  $head \rightarrow body$ 。
  - 开始符号（Start Symbol）：一个特殊的非终结符号，代表整个文法所要定义的语言结构。
2. GNU Bison: Bison 是一个语法分析器生成器，其前身是 Yacc。它接收一个以“.y”为后缀的语法规范文件，该文件定义了一个上下文无关文法，并能据此生成一个 C 语言的语法分析器函数“yyparse()”。此分析器通过不断调用词法分析器“yylex()”获取 Token，并根据产生式规则对 Token 序列进行匹配，最终构建出输入的语法结构。
3. Flex 与 Bison 的交互: Flex 生成的“yylex()”函数负责从输入流中识别出词法单元，并将其返回给 Bison 生成的“yyparse()”函数。两者通过共享一组预定义的 Token 编码来通信，这些编码通常在一个由 Bison 生成的头文件中定义，Flex 的“.l”文件需要包含该头文件。当“yylex()”读到文件末尾时，会返回 0，通知“yyparse()”输入已结束。

### 三、实验环境

容器化工具: Docker

基础镜像: ubuntu:22.04

云环境: 由于本地 Docker 网络问题, 本次实验借助 GitHub Codespaces 完成 Docker 镜像的构建与打包。

核心工具:

build-essential: 提供 gcc 等核心编译工具。

flex: 词法分析器生成器。

bison: 语法分析器生成器。

测试语言: python3

### 四、实验内容

本次实验包含两个核心部分, 旨在实践 Bison 的使用以及其与 Flex 的结合:

**实现简单计算器(calc):** 编译并运行一个基于 Flex 和 Bison 的简单整数计算器。通

过向其传递一个四则运算表达式，验证其能否正确执行并输出计算结果，熟悉 Flex 与 Bison 的基本工作流程。

**实现有效的括号匹配(parentheses):** 根据 LeetCode 第 20 题的要求，编写一个能够判断包含 “()”，“[]”，“{}” 的字符串是否有效的程序。此任务的核心是设计一个无二义性的上下文无关文法，并解决在编译和链接动态库过程中可能遇到的各种问题。

## 五、实验过程与结果分析

### 1. 环境配置

使用 GitHub 的 Codespaces 辅助配置 docker 环境。

将实验项目文件上传至 GitHub 仓库。

```
create mode 100644 README.pdf
create mode 100644 calc/Makefile
create mode 100644 calc/calc_test.py
create mode 100644 calc/lex.l
create mode 100644 calc/syntax.y
create mode 100644 parentheses/Makefile
create mode 100644 parentheses/data.pickle
create mode 100644 parentheses/lex.l
create mode 100644 parentheses/paren_test.py
create mode 100644 parentheses/syntax.tab.h
create mode 100644 parentheses/syntax.y

62477@Bastandern MINGW64 /d/HW/BUPT-Compiler-Lab3-2025Fall (main)
$ git push -u origin main
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 16 threads
Compressing objects: 100% (16/16), done.
Writing objects: 100% (16/16), 35.62 KiB | 2.37 MiB/s, done.
Total 16 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To github.com:Bastandern/bupt-compiler-lab3.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

62477@Bastandern MINGW64 /d/HW/BUPT-Compiler-Lab3-2025Fall (main)
$
```

利用 GitHub Codespaces 功能，在云端环境中成功构建了 Docker 镜像 bupt-compiler:latest。将云端构建好的镜像打包为 “bupt-compiler-image.tar” 文件并下载至本地。

```
docker build -t bupt-compiler:latest . # 构建镜像
docker save -o bupt-compiler-image.tar bupt-compiler:latest # 打包镜像
```

da9483a303782804c29135085670671bbd450780095c07fa61f3a70d0197f588

```

@Bastardern →/workspaces/bupt-compiler-lab3 (main) $ docker build -t bupt-compiler:latest .
[+] Building 46.2s (8/8) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile              0.0s
=> => transferring dockerfile: 233B                               0.0s
=> [internal] load metadata for docker.io/library/ubuntu:22.04  2.6s
=> [auth] library/ubuntu:pull token for registry-1.docker.io    0.0s
=> [internal] load .dockerignore                                 0.0s
=> => transferring context: 2B                                     0.0s
=> [1/3] FROM docker.io/library/ubuntu:22.04@sha256:09506232a8004baa32c47d68f1e5c307d648fdd59f5e7eaa42aaf87914100db3  2.2s
=> => resolve docker.io/library/ubuntu:22.04@sha256:09506232a8004baa32c47d68f1e5c307d648fdd59f5e7eaa42aaf87914100db3  0.0s
=> => sha256:09506232a8004baa32c47d68f1e5c307d648fdd59f5e7eaa42aaf87914100db3 6.69kB / 6.69kB  0.0s
=> => sha256:4cb780d50443fc4463f1f9360c03ca46512e4fdd8fd97c5ce7e69c8758924575 424B / 424B  0.0s
=> => sha256:392fa14ddd09da29a5c3d26948ff81c494424035b755d01b84ab12d92127433 2.30kB / 2.30kB  0.0s
=> => sha256:af6eca94c8104c8e90d3f9efe59c2b3a02b20aad3d985e31c7cd009ea104c447 29.54MB / 29.54MB  0.9s
=> => extracting sha256:af6eca94c8104c8e90d3f9efe59c2b3a02b20aad3d985e31c7cd009ea104c447 1.1s
=> [2/3] WORKDIR /workspace                                     1.3s
=> [3/3] RUN apt-get update && apt-get install -y build-essential flex bison python3 && rm -rf /v 36.9s
=> exporting to image                                           3.1s
=> => exporting layers                                           2.3s
=> => writing image sha256:f4df2c84e146463e669348dcd3bfcd980553c6fb7b398815d16f55f8c90bee4b 0.0s
=> => naming to docker.io/library/bupt-compiler:latest          0.8s

```



在本地加载镜像。

```
docker load -i bupt-compiler-image.tar
```

最后，使用 `docker run` 命令创建并启动容器，同时将本地实验目录挂载到容器的“/workspace”目录下，成功搭建了实验环境。

```
docker run -it -v "$(pwd):/workspace" --name compiler-lab bupt-compiler:latest /bin/bash
```

```
(base) bastandern@Bastandern:~/projects/Lab3/BUPT-Compiler-Lab3-2025Fall$ docker load -i bupt-compiler-image.tar
767e56ba346a: Loading layer [=====>] 80.42MB/80.42MB
8426229a4f55: Loading layer [=====>] 1.536kB/1.536kB
e3f7336a0fb9: Loading layer [=====>] 308.6MB/308.6MB
Loaded image: bupt-compiler:latest
(base) bastandern@Bastandern:~/projects/Lab3/BUPT-Compiler-Lab3-2025Fall$ docker run -it -v "$(pwd):/workspace" --name compiler-lab bupt-compiler:latest /bin/bash
root@da9483a30378:/workspace#
```

再次进入容器只需运行如下命令行：

```
docker start compiler-lab          # 启动容器
docker exec -it compiler-lab /bin/bash # 进入容器
```

## 2. 计算器示例(calc)

```
cd calc      #进入目录
make         #编译
```

```
root@da9483a30378:/workspace/calc# make
flex lex.l
bison -t -d syntax.y
gcc syntax.tab.c -lfl -D CALC_MAIN -o calc.out
```

测试 calc 程序，用“echo”和管道“|”把算式传给它：

```
echo "2*3+5=" | ./calc.out      # 预期输出 11
echo "10-2*3=" | ./calc.out     # 预期输出 4
```

```
root@da9483a30378:/workspace/calc# echo "2*3+5=" | ./calc.out
= 11
root@da9483a30378:/workspace/calc# echo "10-2*3=" | ./calc.out
= 4
```

测试结果表明计算器能够正确处理运算符的优先级，符合预期。

## 3. 有效的括号(parentheses)

脚本解释如下：

syntax.py

```
%{
```

```
#include <stdio.h>
#include <string.h>

void yyerror(const char *s);
extern int yylex();
extern void yy_scan_string(const char *str);

int result;
%}

%token LP RP LB RB LC RC

%%

program:
    sequence
    ;

sequence:
    %empty
    | sequence term
    ;

term:
    LP sequence RP
    | LB sequence RB
    | LC sequence RC
    ;

%%

void yyerror(const char *s) {
    result = 0;
}

int validParentheses(char *expr) {
    result = 1;
    yy_scan_string(expr);
    yyparse();
    return result;
}
```

## C 代码与声明部分

- `%token LP RP LB RB LC RC`: 这部分向 **Bison** 声明了所有合法的词法单元，这些 Token 由 “lex.l” 文件在识别出 “(”, “)”, “[” 等字符时生成。

## 语法规则部分

- `program: sequence;`: 定义了整个程序的顶层结构是一个序列。
- `sequence: %empty | sequence term;`: 它使用左递归来定义一个序列：可以是 `empty`，也可以是在一个已有的序列后面，再追加一个合法的项。这种定义方式描述了并列结构。
- `term: LP sequence RP | ...`: 定义了一个合法的“项”必须是一对正确闭合的括号，并且其内部包裹的也必须是一个合法的序列。处理了嵌套结构。

## 附加 C 函数部分

- `void yyerror(const char *s)`: 错误处理函数。当 **Bison** 在解析过程中发现不符合语法规则的情况时，就会调用这个函数
- `int validParentheses(char *expr)`: 接口函数。工作流程：假设输入是有效的。调用 “`yy_scan_string(expr)`”，让 **Flex** 准备好要分析的字符串。调用 “`yyparse()`”，启动语法分析。如果中途出错，“`yyerror`” 会将 `result` 修改为 0。返回最终的 `result` 值。

## lex.l

```
%{
#include "syntax.tab.h"
%}

%option noyywrap

%%

"("      { return LP; }
")"      { return RP; }
"["      { return LB; }
"]"      { return RB; }
"{"      { return LC; }
"}"      { return RC; }

[ \t\n]+ { /* Do nothing, just ignore it. */ }
```

```
.          { /* Do nothing, just ignore it. */ }

<<EOF>>   { return 0; }

%%
```

"(" { return LP; }：当匹配到字符(时，执行动作“{return LP;}”，即向 Bison 返回一个 LP 类型的 Token。其他五种括号的规则与此类似。

[\\t\\n]+：这个模式匹配一个或多个连续的空白字符。后面的动作{}是空的，表示匹配到这些字符后什么也不做，即直接忽略它们。

.：这个模式可以匹配除换行符以外的任何单个字符。它的动作也是空的，这意味着如果输入中出现了任何非括号、非空白的字符，它们也会被直接忽略掉。

<<EOF>> { return 0; }：这是一个至关重要的规则。“<<EOF>>”是 Flex 中一个特殊的模式，用于匹配输入的结尾。当 Flex 读到字符串末尾时，它会执行动作“{return 0;}”。向 Bison 返回 0 是一个约定好的信号，表示“所有输入都已处理完毕”，从而让语法分析器可以正确地结束解析过程。

准备好脚本之后，编译并运行测试。

```
make
python3 paren_test.py
```

```
root@da9483a30378:/workspace/parentheses# make
flex lex.l
bison -t -d syntax.y
gcc lex.yy.c syntax.tab.c -lfl -fPIC --shared -o libparen.so
root@da9483a30378:/workspace/parentheses# python3 paren_test.py
All tests passed!
```

显示“All tests passed!”表示通过所有测试样例。

## 六、问题与解决

### 1. docker 网络问题

在本地直接构建 Docker 镜像时遇到网络超时错误。通过 GitHub Codespaces 拉取镜像再下载到本地创建 docker 容器即可。



```
(base) bastandern@Bastandern:~/projects/lab3$ docker build -t bupt-compiler:latest .
[+] Building 56.5s (2/2) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 244B
=> ERROR [internal] load metadata for docker.io/library/ubuntu:22.04
-----
> [internal] load metadata for docker.io/library/ubuntu:22.04:
-----
Dockerfile:1
-----
1 | >>> FROM ubuntu:22.04
2 |
3 |     WORKDIR /workspace
-----
ERROR: failed to build: failed to solve: DeadlineExceeded: failed to fetch anonymous token: Get "https://auth.docker.io/token?scope=repository%3Alibrary%2Fubuntu%3Apull&service=registry.docker.io": dial tcp 199.96.63.53:443: i/o timeout
```

## 2. 编译过程中出现链接错误

编译过程中出现链接错误 “cannot find -ly”。

```
root@da9483a30378:/workspace/calculator# make
flex lex.l
bison -t -d syntax.y
gcc syntax.tab.c -lfl -ly -D CALC_MAIN -o calc.out
/usr/bin/ld: cannot find -ly: No such file or directory
collect2: error: ld returned 1 exit status
make: *** [Makefile:11: calc] Error 1
```

原因分析：Makefile 文件中的链接选项 “-ly” 是 Yacc 时代的产物，用于链接 Yacc 库。而现代的 Bison 已经不再需要这个库了，因此该选项是多余的。

解决方案：编辑 “calc/Makefile” 文件，将 gcc 命令中的 “-ly” 选项删除。

```
# 使用 sed 命令快速修改
sed -i 's/-ly//g' Makefile
```

## 3. Bison 语法二义性警告

Bison 输出了大量的 “shift/reduce conflicts” 和 “reduce/reduce conflicts” 警告。

```
root@da9483a30378:/workspace/parentheses# make
flex lex.l
bison -t -d syntax.y
syntax.y: warning: 30 shift/reduce conflicts [-Wconflicts-sr]
syntax.y: warning: 8 reduce/reduce conflicts [-Wconflicts-rr]
syntax.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
syntax.y:20.7-31: warning: rule useless in parser due to conflicts [-Wother]
    20 |         | valid_string valid_string
        |         ^ ~~~~~
gcc lex.yy.c syntax.tab.c -lfl -fPIC --shared -o libparen.so
```

原因分析：警告源于 “syntax.y” 文件中的产生式 “valid\_string: valid\_string valid\_string”。这条规则存在二义性。例如，对于输入 “()[]”，分析器无法确定

是应该先匹配 “()” 形成 “valid\_string”，再与 “[]” 组合；还是先匹配 “[]” 形成 “valid\_string”，再与 “()” 组合。这种不确定性导致了冲突。

解决方案：重写语法规则，消除二义性。采用左递归的方式，将语法结构定义为“一个序列可以是一个空序列，或者是一个序列后面跟一个合法的项”。

```
/* 修改后的核心规则 */
sequence:
    %empty
    | sequence term
;

term:
    LP sequence RP
    | LB sequence RB
    | LC sequence RC
;
```

这样，对于 “()[]”，分析器会先处理 “()”，将其规约为 term，并追加到 sequence 后面；接着处理 “[]”，同样规约为 term 再追加。整个过程是线性的、无冲突的。

#### 4. Python 测试脚本运行时错误

运行 paren\_test.py 时，出现 “OSError: /workspace/parentheses/libparen.so: undefined symbol: yy\_scan\_string”。

```
root@da9483a30378:/workspace/parentheses# python3 paren_test.py
Traceback (most recent call last):
  File "/workspace/parentheses/paren_test.py", line 20, in <module>
    lib = ctypes.cdll.LoadLibrary(lib_path)
  File "/usr/lib/python3.10/ctypes/__init__.py", line 452, in LoadLibrary
    return self._dlltype(name)
  File "/usr/lib/python3.10/ctypes/__init__.py", line 374, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /workspace/parentheses/libparen.so: undefined symbol: yy_scan_string
```

原因分析：该错误表明动态链接库 “libparen.so” 在加载时找不到 “yy\_scan\_string” 函数的定义。这个函数是由 Flex 在 “lex.yy.c” 文件中生成的。查看 Makefile 发现，在生成 “.so” 文件时，gcc 命令只包含了 Bison 生成的 “syntax.tab.c”，而没有包含 Flex 生成的 “lex.yy.c”。

解决方案：修改 “parentheses/Makefile”，在 gcc 命令中同时加入 “lex.yy.c” 和 “syntax.tab.c” 作为输入文件。

```
# 修改前
# $(CC) syntax.tab.c -lfl -fPIC --shared -o libparen.so
# 修改后
$(CC) lex.yy.c syntax.tab.c -lfl -fPIC --shared -o libparen.so
```

## 七、总结与心得

通过本次实验，我收获良多。

首先，我对编译器前端的工作流程有了更具体、更深入的认识。实验让我亲手将词法分析与语法分析两个阶段连接起来，直观地看到了 **Flex** 如何将字符流切分为 **Token**，而 **Bison** 又如何消费这些 **Token** 来验证语法结构，这种理论与实践的结合加深了我的理解。

其次，我熟练掌握了 **Bison** 工具的基本使用。从编写 **.y** 文件定义文法，到使用 **make** 工具链生成代码并编译，再到最终的调试，整个流程走下来，让我对语法分析器的构建有了全局性的把握。特别是在解决括号匹配问题时，处理语法二义性的经历让我深刻体会到，设计一套精确、无歧义的文法是语法分析阶段的重中之重。

最后，本次实验极大地锻炼了我的问题解决能力。无论是处理过时的 **Makefile** 链接选项，还是分析并解决动态库的“符号未定义”错误，这些调试过程让我学会了如何根据错误信息追根溯源，并最终定位问题。这不仅是编译技术的学习，更是工程实践能力的宝贵积累。