



语言解析器 实验讲解

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年03月20日



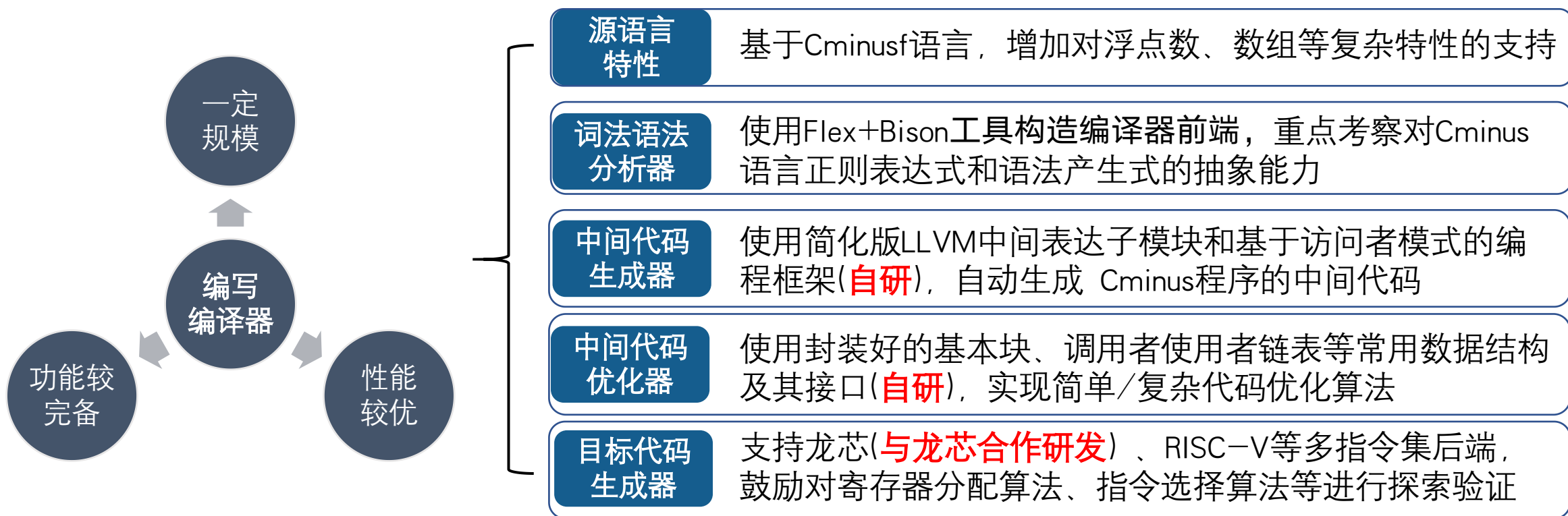
- 实验概况
- 语言解析器实验



实验体系总览



- 目标：面向**国家战略需求**，以**系统和创新能力**的培养为导向，根据计算机学科发展的趋势和新时代大学生的成长需要构建**现代编译实验体系**，体现**先进性、挑战度**，增强学生**获得感**

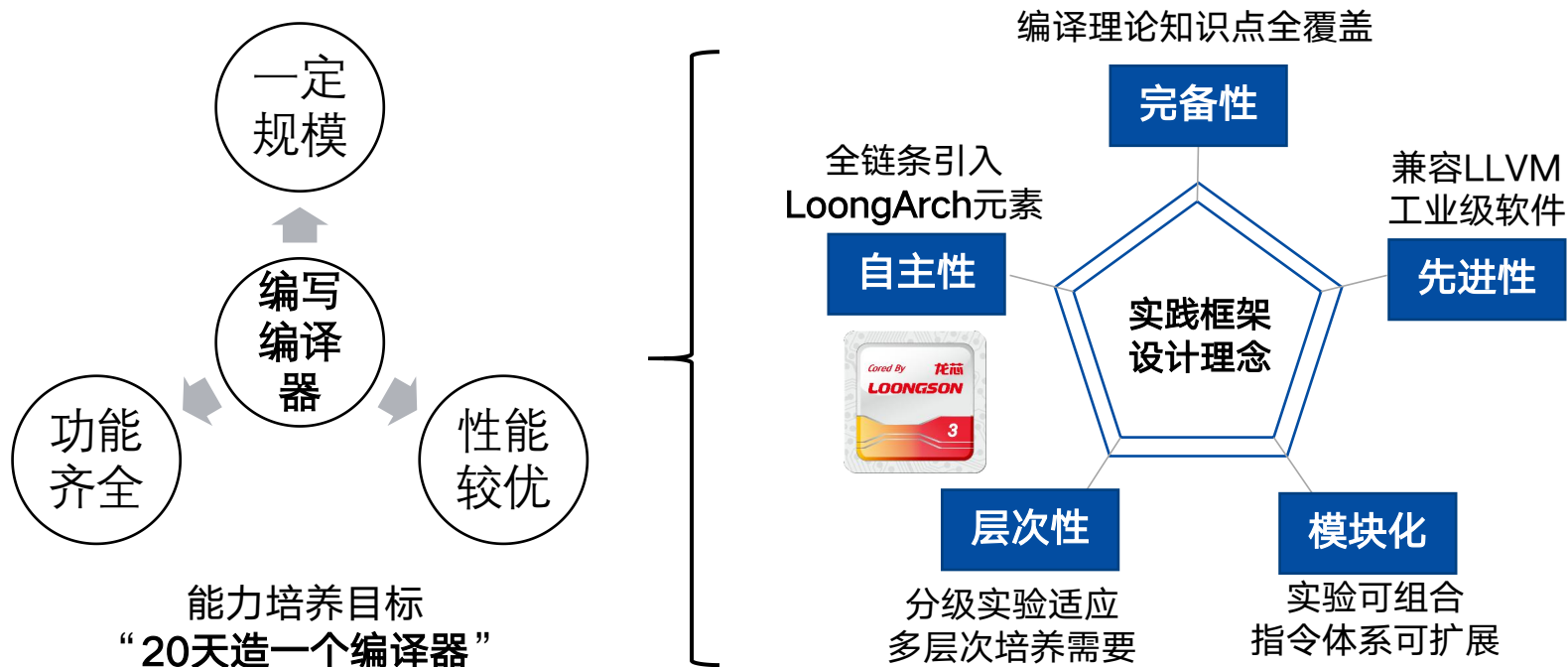




实验体系总览



自研首个支持自主指令集的编译实践体系



大规模：熟练使用10种开发软件，完成5个大实验，实现1500+行核心代码

功能全：基础功能实验（能用）+ 高阶挑战实验（好用）+ 创新开放实验（用好）

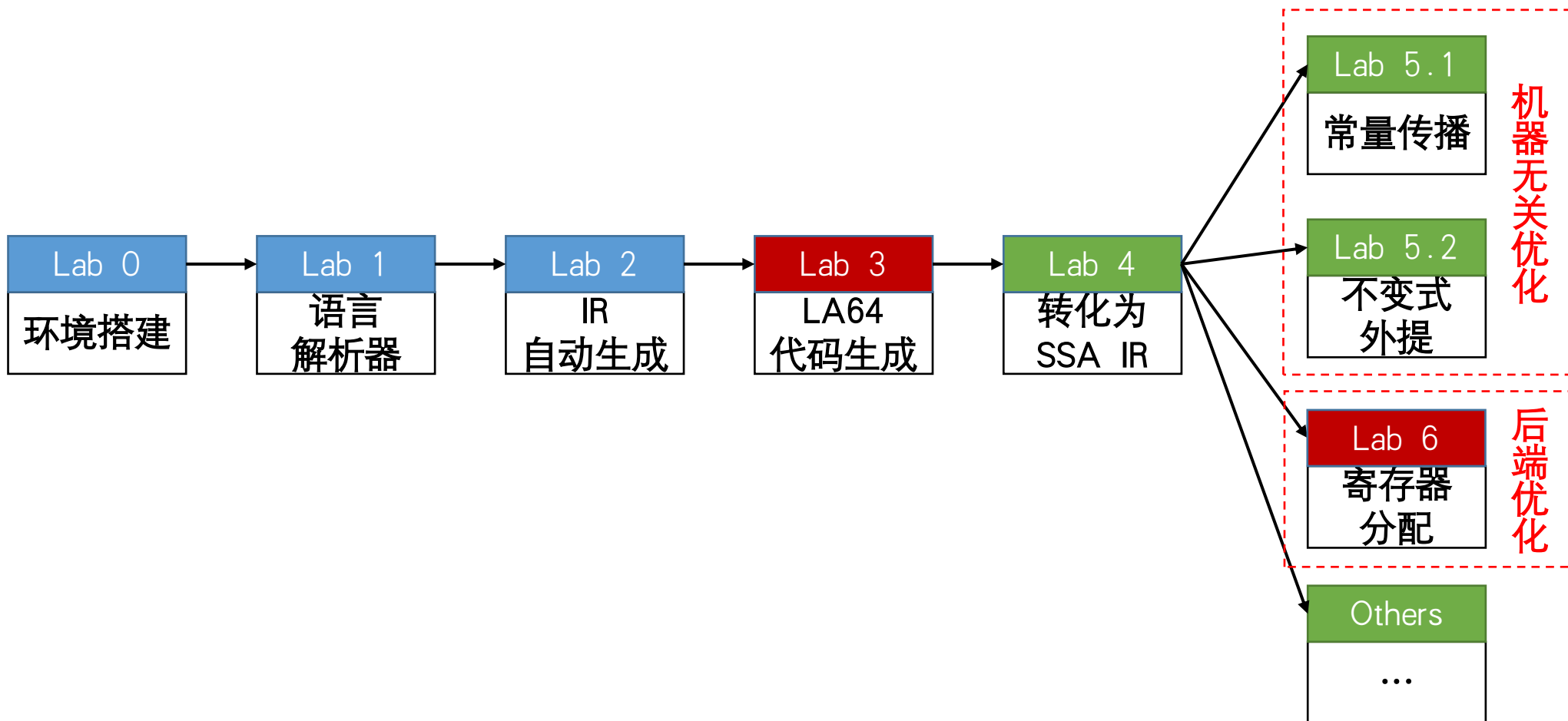
性能优：在卷积计算等4个智能计算任务上，性能比g++ -O3快16 - 68%



实验框架图



- 基础实验+龙芯后端实验+高阶创新实验有机结合





Lab 0 环境搭建



- **目的：**

- 熟悉虚拟机、VS Code、git、gdb等工具的使用，完成后端环境安装

- **要求：**

- 成功安装虚拟机，并在虚拟机安装龙芯模拟后端环境，熟练使用git命令

- **考核标准：**

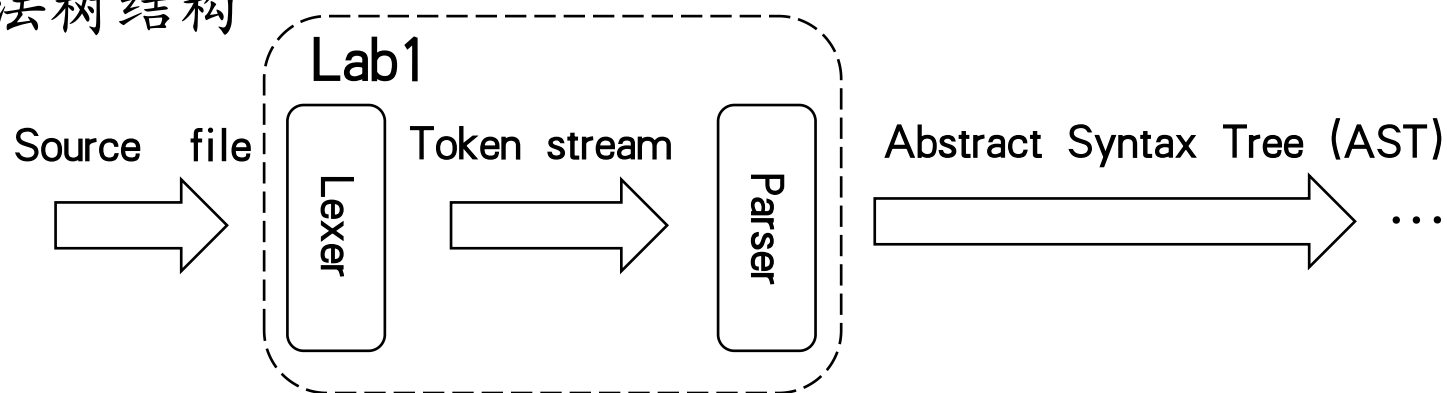
- 使用clang编译预留c文件，生成的中间代码能输出学生学号
- 成功解决仓库冲突



Lab 1 语言解析器



- **目的：从无到有完成Cminus-f解析器**
 - 掌握并运用词法分析、语法分析基础知识
 - 掌握flex和bison的原理和使用
- **要求：**
 - 基于flex生成词法分析器、将字符流转为token流
 - 基于bison生成语法分析器、根据token流建立语法树
- **考核标准：**
 - 对测试样例输出正确的语法树结构





Lab 1 语言解析器



• 实验结果:

• 词法分析器

```
root@00b10620b7eb:/labs/2022fall-compiler_cminus# ./build/lexer
Token      Text      Line      Column (Start,End)
260        int       1         (0,3)
259        main     1         (4,8)
282        (       1         (8,9)
262        void    1         (9,13)
283        )       1         (13,14)
284        {       1         (15,16)
260        int     2         (1,4)
259        i       2         (5,6)
286        ;       2         (6,7)
261        float   2         (8,13)
259        j       2         (14,15)
286        ;       2         (15,16)
```

正确识别token、及对应Text与行列数

• 语法分析器

```
root@00b10620b7eb:/labs/2022fall-compiler_cminus# ./build/parser
>--+ program
| >--+ declaration-list
| | >--+ declaration
| | | >--+ fun-declaration
| | | | >--+ type-specifier
| | | | | >--+ int
| | | | | >--+ main
| | | | | >--+ (
| | | | | >--+ params
| | | | | | >--+ void
| | | | | >--+ )
| | | | >--+ compound-stmt
| | | | | >--+ {
| | | | | >--+ local-declarations
```

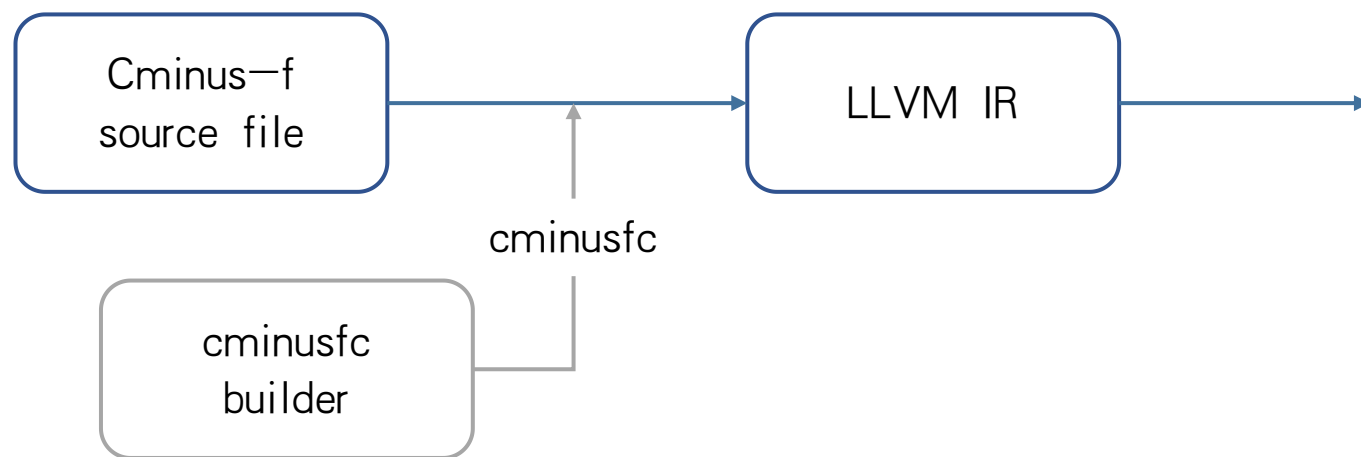
从Cminus-f代码分析得到正确语法树



Lab 2 IR自动生成



- 目的：掌握源代码到IR的生成过程
- 要求：使用LightIR框架，实现符合Cminus-f规则的IR产生算法
- 考核标准：提供多个不同复杂度的样例，比较由学生编写的cminusfc生成的文件与参考程序生成的文件差异，给出分数





Lab 2 IR自动生成



• 运行输出的结果

```
=====lv3 START=====
complex1:      Success
complex2:      Success
complex3:      Success
complex4:      Success
points of lv3 is: 11
=====lv3 END=====

total points: 100
```

80/80个通过测试用例 状态: **Accept**

• 实验平台评测结果

lv0_1 / decl_float.cminus	Accept	4
lv0_1 / decl_float_array.cminus	Accept	4
lv0_1 / decl_int.cminus	Accept	4
lv0_1 / decl_int_array.cminus	Accept	4
lv0_1 / input.cminus	Accept	4
lv0_1 / output_float.cminus	Accept	4
lv0_1 / output_int.cminus	Accept	4
lv0_1 / return.cminus	Accept	6
lv0_2 / num_add_float.cminus	Accept	1



Lab 3 LA64代码生成



• 目的:

- 熟悉了解自主指令集汇编语言的语法、将IR转换为汇编代码

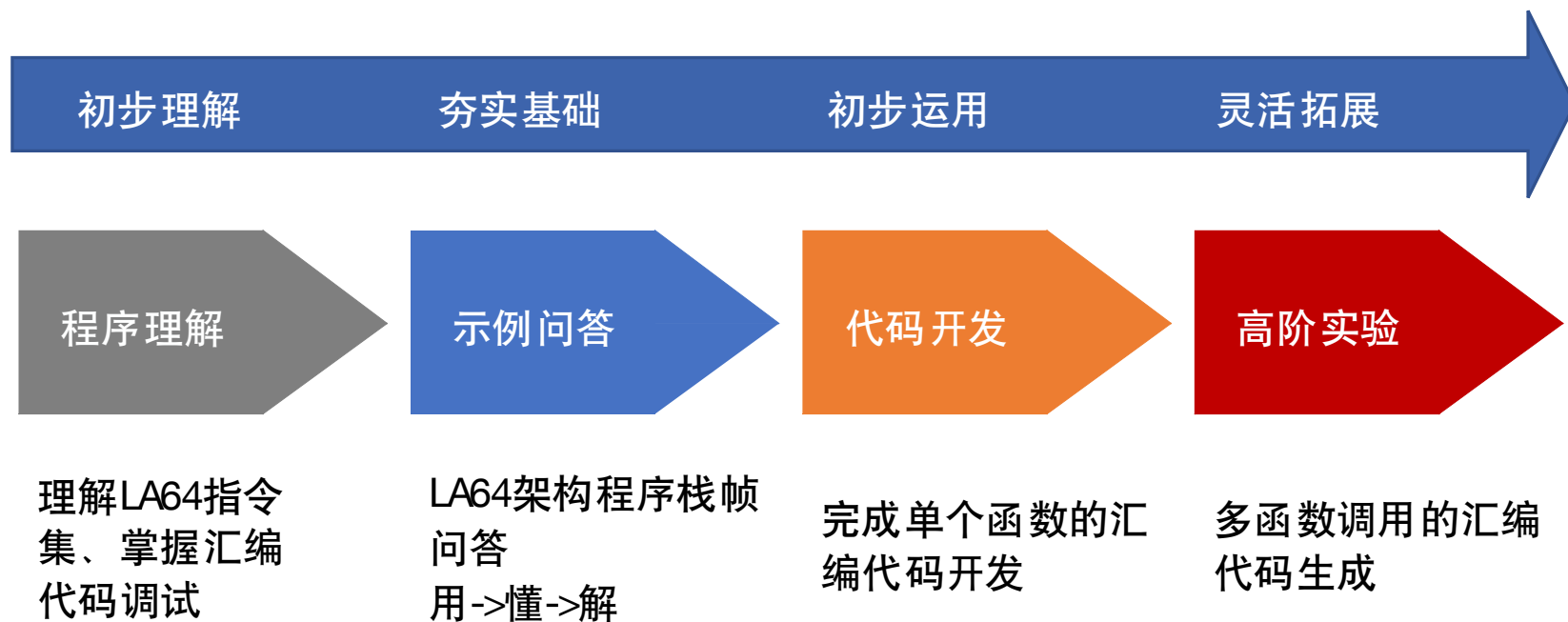
• 要求:

- 学生可以自主阅读、编写、调试LA64汇编代码
- 完成寄存器在栈上分配的汇编代码生成

实验内容: 基于自主指令集的编译后端实验

• 考核标准:

- 完成示例代码问答
- 完成要求汇编程序段编写
- 完成寄存器在栈上分配





• LA64架构ABI介绍

• 基础指令介绍

- ADD、SUB、OR、B、BL、JIBL、LD、ST...
- 示例使用程序

• 寄存器使用约定

- 整形/浮点型寄存器调用约定

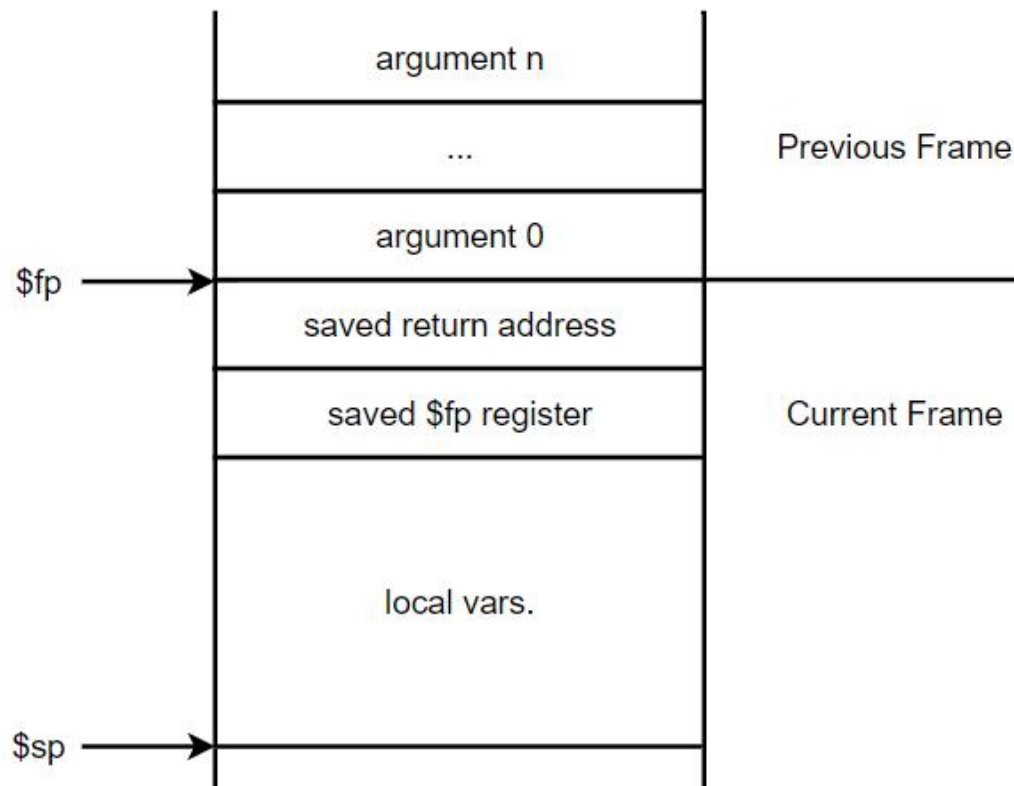
• 函数调用约定

- 栈帧布局、函数调用过程、参数传递、函数值的返回

• LA64汇编概述与示例

- 对单函数程序、带有逻辑操作、函数调用、浮点数使用等源程序生成的汇编代码解释
- LA64下的编译与调试

```
gcc src/io/i.c test.s -g -o test
```





Lab 3 LA64代码生成



源程序1

```
int main() {  
    return 0;  
}
```

LA64下汇编

```
.text                # 标记代码段  
.globl main          # 标记 main 全局可见 (必需)  
.type main, @function # 标记 main 是一个函数  
main:               # 标记程序入口  
    addi.d $sp, $sp, -16 # 分配栈空间 (必需), -16为16字节  
    st.d   $ra, $sp, 8   # 保存返回地址 (必需)  
    addi.d $fp, $sp, 16  # 设置帧指针  
  
    addi.w $a0, $zero, 0 # 设置返回值为 0  
    ld.d   $ra, $sp, 8   # 恢复返回地址 (必需)  
    addi.d $sp, $sp, 16  # 释放栈空间 (必需)  
    jr     $ra           # 返回调用main函数的父函数 (必需)
```



Lab 3 LA64代码生成



源程序2

```
int a;  
int main() {  
    a = 4;  
    if (a > 0)  
        return 1;  
    return 0;  
}
```

源程序3

```
int add(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int a = 1;  
    int b = 2;  
    int c = add(a, b);  
    output(c);  
    return 0;  
}
```

源程序4

```
int main () {  
    float b = 8;  
    float c = 3.5;  
    if (b < c)  
    {  
        return 1;  
    }  
    return 0;  
}
```



Lab 4 转换为SSA IR



- **目的:**

- 熟悉了解mem2reg优化


- **要求:**

- 学生完成mem2reg代码补全
- 掌握mem2reg算法

- **考核标准:**

- 回答示例问题
- 完成代码补全
- 测试样例通过

1 y :=
2 y :=
 x :=
y y
 y1 := 1
 y2 := 2
 x1 := y2

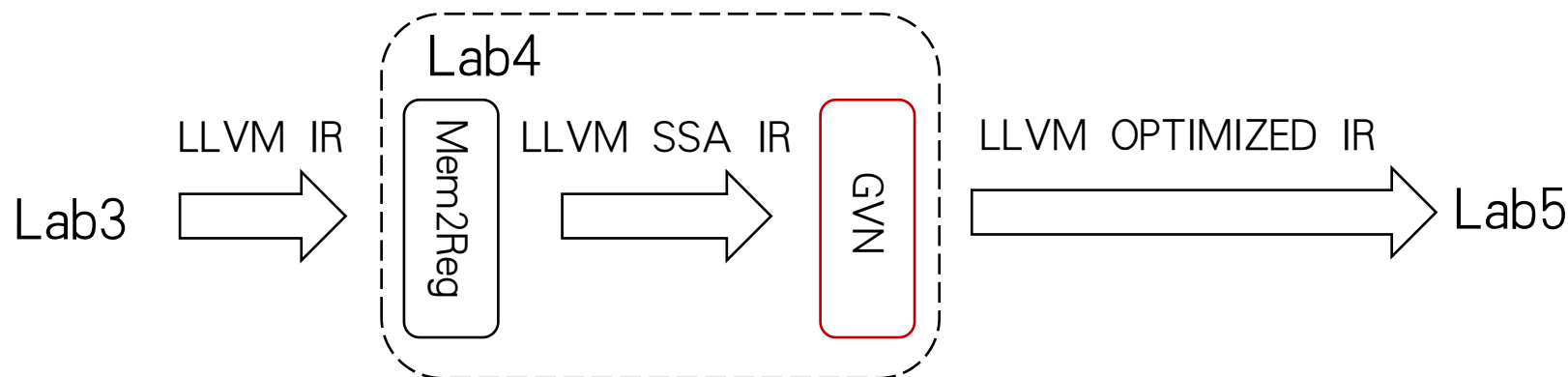




Lab 5 机器无关优化



- 目的：在中间代码上实现机器无关优化、掌握数据流分析过程
- 要求：在阅读 Mem2Reg 优化基础上，理解 LLVM SSA IR 格式，并实现全局值编号 (GVN) 算法



• 考核标准：

- 提供 8 个基础测试用例，通过比对分析结果判断 GVN 算法正确性
- 提供 2 个进阶测试用例，通过测试优化前后运行时间评价优化效果



Lab 5 机器无关优化



• 8个基础测例：

```
bin.cminus: 1.0
recursive_vpf.cminus: 0.9090909090909091
pure_func.cminus: 1.0
loop3.cminus: 1.0
single_bb1.cminus: 1.0
complex.cminus: 1.0
loop2d1.cminus: 1.0
bin2.cminus: 1.0
```

• 2个进阶测例：

testcase	before optimization	after optimization	baseline
const-prop.cminus	0.68	0.23	0.23
transpose.cminus	3.86	3.15	3.15



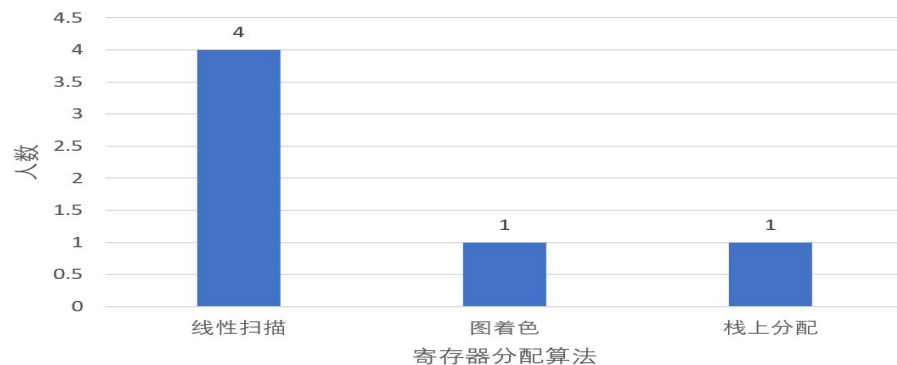
Lab 6 寄存器分配与整合

• 实验时间安排

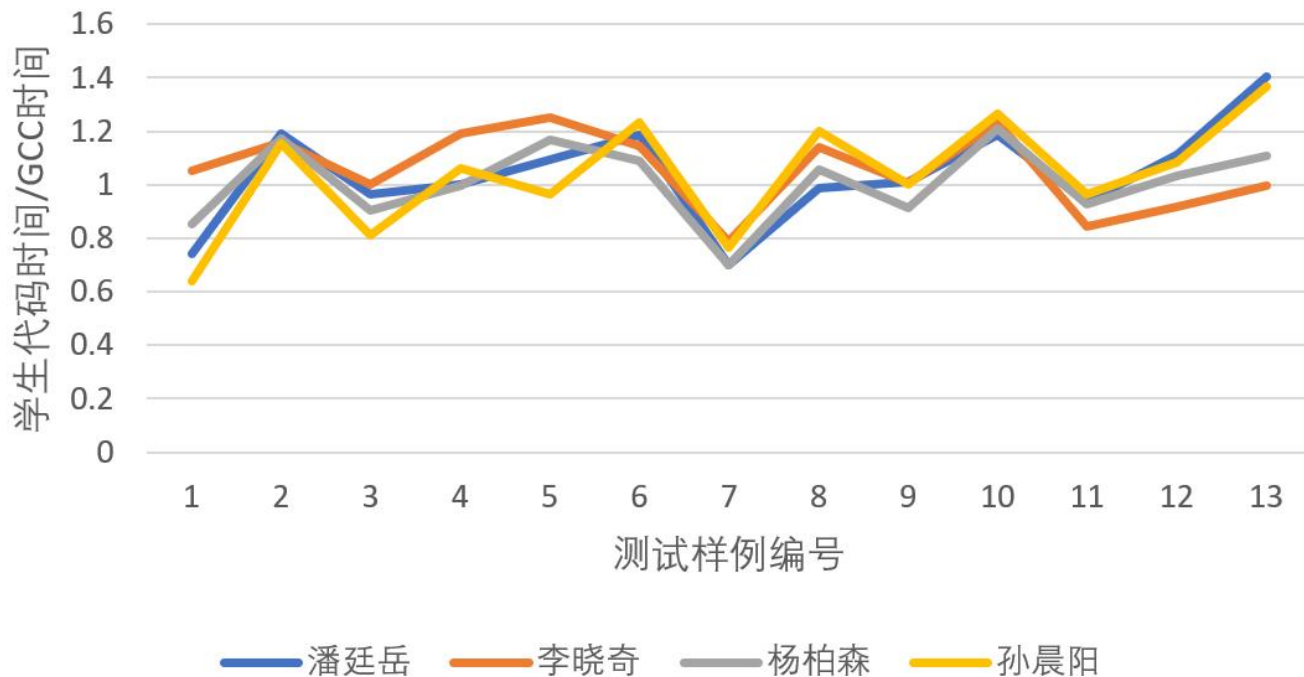
- 2023.1.20 进行实验介绍
- 2023.1.30~2023.3.14 学生实验，每周进行实验答疑
- 2023.3.15 结题答辩
- 配置4名助教进行实验开发与辅导

• 实验结果

- 6组7名同学完成实验
- 4组完成所有功能测试，并进行优化



学生代码运行时间与GCC比值





Lab 6 寄存器分配与整合



- 实验样例

- 1-return
- 2-calculate
- 3-output
- 4-if
- 5-while
- 6-array
- 7-function
- 8-store
- 9-fibonacci
- 10-float
- 11-floatcall
- 12-global
- 13-complex

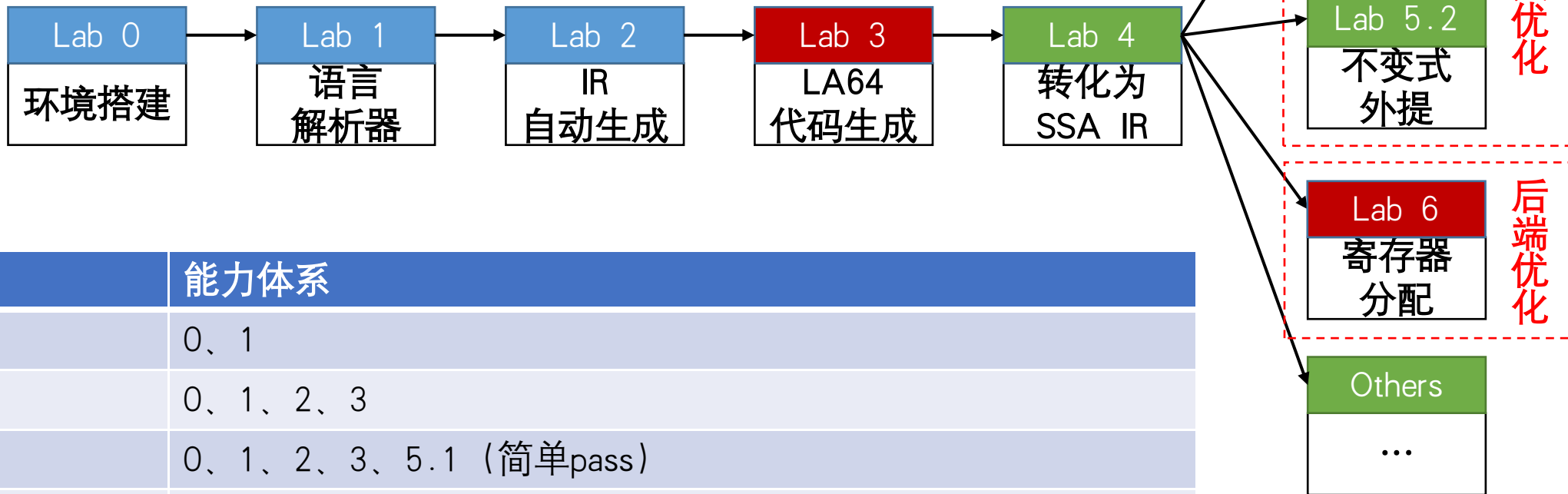
0/1 背包问题



模块化的现代编译实践架构与自组织形态



- 基础实验+龙芯后端实验+高阶创新实验有机结合



难度	能力体系
简单	0、1
适中	0、1、2、3
微难	0、1、2、3、5.1（简单pass）
中难	0、1、2、3、5（复杂pass，既有分析又有优化）
中难	0、1、2、3、4、5
中难	0、1、2、3、6
难	0、1、2、3、4、5、6



- 实验概况
- 语言解析器实验



- 正则表达式
- Flex简介
- Bison简介
- Flex和Bison联动



- 正则表达式是一种用于描述文本模式的强大工具，特别是在处理文本搜索、替换和验证等任务时非常有用。

```
import re
```

```
def find_email(text):
```

```
    email_pattern = r'\b[A-Za-z0-9._%+-]+\@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
```

```
    matches = re.findall(email_pattern, text)
```

```
    return matches
```

正则表达式

```
text = "我的电子邮件是example@gmail.com，你可以联系我。"
```

匹配文本

```
emails = find_email(text)
```

```
for email in emails:
```

```
    print(email)
```

输出: example@gmail.com



正则表达式规则（仅列举部分）：

- **[0-9]**：匹配数字0-9中的一个字符；
- **.**：匹配任何单个字符；
- *****：匹配前面的元素零次或多次，比如 **0*** 匹配由0构成的字符串；
- **+**：匹配前面的元素一次或多次；
- **?**：匹配前面的元素零次或一次；
- **\d**：匹配一个数字字符，等价于 **[0-9]**；
-



实验使用的正则表达式规则可以参考Flex文档

安装完毕Flex后，可通过info指令打开，选择Patterns章节

```
→ ~ sudo apt install flex
[sudo] password for gpzlx1:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
flex is already the newest version
0 upgraded, 0 newly installed, 0 to
→ ~ info flex
```

```
* Menu:

* Copyright::
* Reporting Bugs::
* Introduction::
* Simple Examples::
* Format::
* Patterns::
* Matching::
* Actions::
* Generated Scanner::
* Start Conditions::
* Multiple Input Buffers::
```



正则表达式核心规则是通用的，可以在线正则表达式平台学习：

<https://c.runoob.com/front-end/854/>

⚙️ 正则表达式在线测试

生成代码 匹配数字 匹配字母 匹配中文 测试实例 可视化图

/ runo{2,5}b

runob
runoob
runoobob
runoobobob

runob
runoob
runoobob
runoobobob

共找到 3 处匹配:
runoob
runoobob
runoobobob



正则表达式在线测试



- 正则表达式
- Flex简介
- Bison简介
- Flex和Bison联动

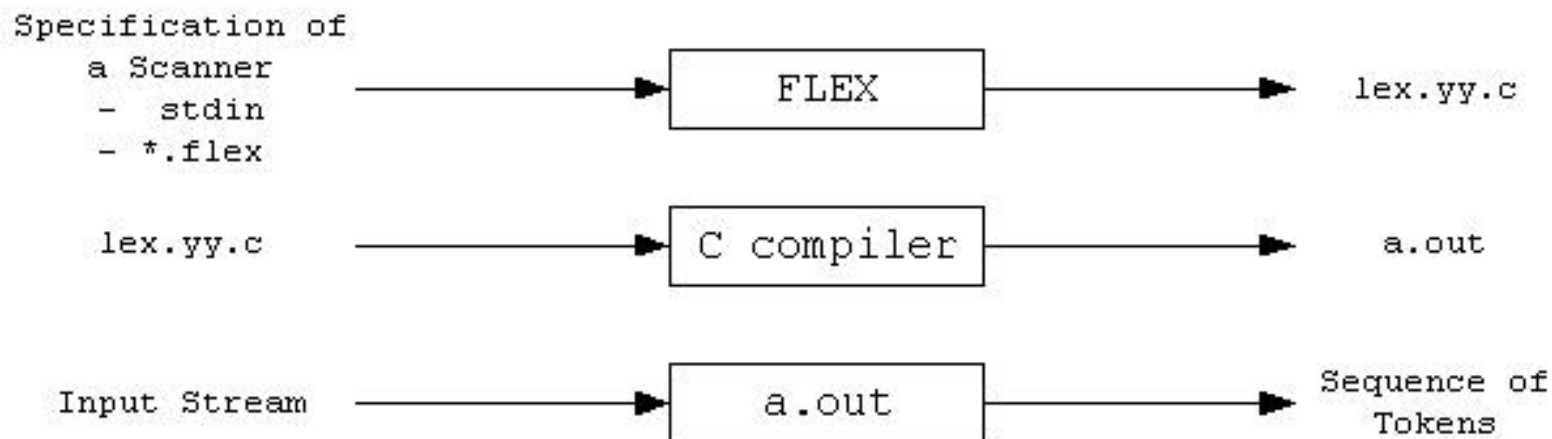
- 最早的词法分析器生成程序Lex由Mike Lesk和Eric Schmidt于1975年在AT&T共同开发完成
- Lex可以和Yacc协同使用，为编译器开发带来了显著便利
- 二十世纪八十年代，Vern Paxson用C语言重新实现了 Lex，并将其命名为Flex

Flex（Fast Lexical Analyzer Generator）是一种用于生成词法分析器的工具

Flex可以根据用户提供的正则表达式规则，将输入文本分割成一个个的词法单元（token），用于后续的语法分析或语义分析

Flex工作流程

1. 以 .l 为后缀的源程序中的规则被转换成状态转换图，生成对应的代码，包括核心的 `yylex()` 函数，保存在 `lex.yy.c` 文件中。
2. 生成的 `lex.yy.c` 文件可以通过 C 编译为可执行文件。
3. 最终，可执行文件将输入流解析成一系列的标记（tokens）。





Flex demo



```
/* file name demo.l */
%option noyywrap
%{
#include <string.h>
int chars = 0;
int words = 0;
}%

%%
[a-zA-Z]+ { chars += strlen(yytext); words++; }
. {}

%%

int main()
{
    yylex();
    printf("look, I find %d words of %d chars\n",
           words, chars);
    return 0;
}
```

```
→ test git:(master) X ls
demo.l
→ test git:(master) X flex demo.l
→ test git:(master) X ls
demo.l lex.yy.c
→ test git:(master) X gcc lex.yy.c
→ test git:(master) X ls
a.out demo.l lex.yy.c
→ test git:(master) X ./a.out
Hello World

look, I find 2 words of 10 chars
```

编译和运行该demo.l文件，其实现了一个单词和字母的统计功能



Flex demo



```
/* file name demo.l */  
%option noyywrap  
%{  
#include <string.h>  
int chars = 0;  
int words = 0;  
%}
```

声明部分

```
%%  
[a-zA-Z]+ { chars += strlen(yytext); words++; }  
. {}  
%%
```

规则部分

```
int main()  
{  
    yylex();  
    printf("look, I find %d words of %d chars\n",  
           words, chars);  
    return 0;  
}
```

C代码部分



声明部分包含名称声明和选项设置

%{ 和 %} 之间的内容会被原样复制到生成的 C 文件头部，可用于编写 C 代码，如头文件声明和变量定义等。

```
/* file name demo.l */  
%option noyywrap  
%{  
#include <string.h>  
int chars = 0;  
int words = 0;  
%}
```

声明部分

原样复制



```
441 char *yytext;  
442 #line 1 "demo.l"  
443 /* file name demo.l */  
444 #line 4 "demo.l"  
445 #include <string.h>  
446 int chars = 0;  
447 int words = 0;  
448 #line 449 "lex.yy.c"  
449 #line 450 "lex.yy.c"
```

生成的lex.yy.c中的内容



规则部分位于两个 %% 之间

每个规则由正则表达式定义的模式和与之匹配的 C 代码动作组成
当词法分析程序识别出某模式时，执行相应的 C 代码

```
%%  
[a-zA-Z]+ { chars += strlen(yytext); words++; }  
.  
%%
```

共有两条规则



规则部分位于两个 %% 之间

每个规则由正则表达式定义的模式和与之匹配的 C 代码动作组成
当词法分析程序识别出某模式时，执行相应的 C 代码。

```
%%  
[a-zA-Z]+ { chars += strlen(yytext); words++; }  
.  
%%
```

共有两条规则

正则表达式，该正则表达式匹配所有单词

当匹配到该正则表达式时，执行的对应动作。
其中 yytext 为匹配到的字符串。



规则部分位于两个 %% 之间

每个规则由正则表达式定义的模式和与之匹配的 C 代码动作组成
当词法分析程序识别出某模式时，执行相应的 C 代码。

```
%%  
[a-zA-Z]+ { chars += strlen(yytext); words++; }  
.  
%%
```

匹配任意内容

不执行任何动作

共有两条规则



Flex在进行词法分析时，可能会遇到二义性的情况

二义性指的是输入文本可以被多个正则表达式规则匹配的情况，导致分析器无法确定选择哪个规则

当存在二义性时，Flex采用以下策略解决：

- **最长匹配原则（Longest Match Rule）：**Flex默认采用最长匹配原则。当输入文本可以匹配多个规则时，选择匹配长度最长的规则。
- **规则顺序优先级：**Flex中规则的顺序决定了它们的优先级，按规则顺序进行匹配（靠前的规则，优先级高）。



```
%%  
\+ { return ADD; }  
= { return ASSIGN; }  
\+= { return ASSIGNADD; }  
%%
```

对于以上规则，对于字符串 “+=”，第三条规则 “\+= { return ASSIGNADD; }” 被触发，遵循最长匹配原则，而不是分别触发一次第一条和第二条规则

```
%%  
ABC { return 1; }  
[a-zA-Z]+ {return 2; }  
%%
```

对于以上规则，对于字符串 “ABC”，第一条规则 “ABC { return 1; }” 被触发，遵循规则顺序优先级。尽管字符串 “ABC” 可以同时触发正则表达式 “ABC” 和 “[a-zA-Z]+”，但是 “ABC” 对应的规则优先被定义。



C 代码部分可包括 `main()` 函数，用于调用 `yylex()` 执行词法分析。
`yylex()` 是由 Flex 生成的词法分析函数，默认从 `stdin` 读取输入文本。

```
int main()
{
    yylex();
    printf("look, I find %d words of %d chars\n",
           words, chars);
    return 0;
}
```

原样复制 ↓

```
1745  #line 12 "demo.l"
1746
1747
1748  ✓ int main()
1749  {
1750      yylex();
1751      printf("look, I find %d words of %d chars\n",
1752             words, chars);
1753      return 0;
1754  }
```

lex.yy.c文件中的内容



C 代码部分可包括 `main()` 函数，用于调用 `yylex()` 执行词法分析。
`yylex()` 是由 Flex 生成的词法分析函数，默认从 `stdin` 读取输入文本。

```
int main()
{
    yylex();
    printf("look, I find %d words of %d chars\n",
           words, chars);
    return 0;
}
```

原样复制 ↓

```
1745  #line 12 "demo.1"
1746
1747
1748  ✓ int main()
1749  {
1750      yylex();
1751      printf("look, I find %d words of %d chars\n",
1752             words, chars);
1753      return 0;
1754  }
```

lex.yy.c文件中的内容

`yylex()` 是由 Flex 根据规则自动生成的，用于开始执行词法分析。其从 `stdin` 读取输入开始匹配。



在Flex中，以"yy"开头的变量和函数是Flex生成的词法分析器中的一些特定名称，用于处理词法分析过程。

- yyin、yyout、**yytext**、yyleng、**yylex**、yywrap
- **yytext**: 这是一个字符串变量，用于存储当前匹配的词法单元的文本。当Flex匹配成功时，yytext将包含匹配的字符串。
- **yylex()**: 这是Flex生成的词法分析器的主函数。它用于从输入流中读取字符并进行词法分析，返回下一个词法单元的标识符。



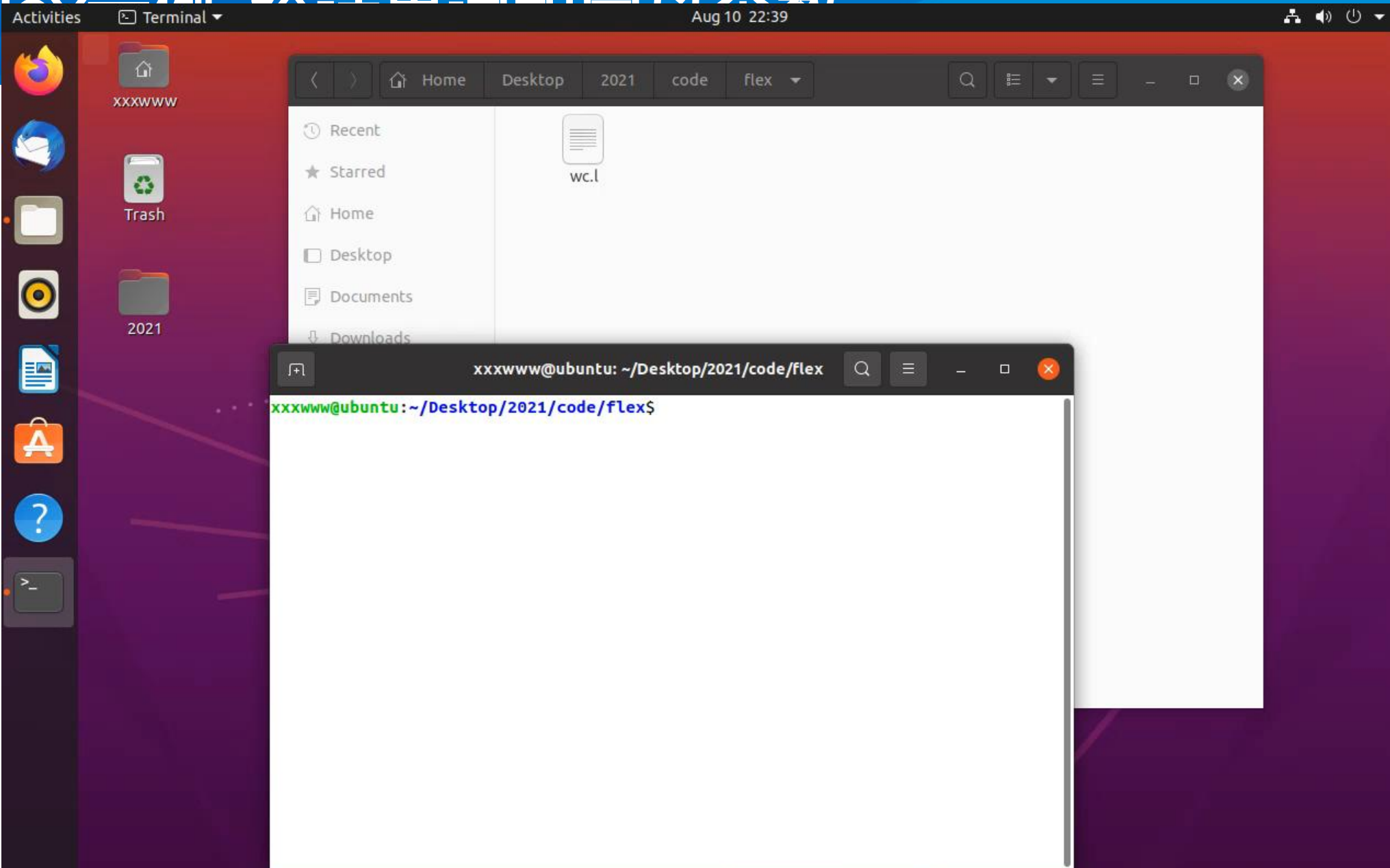
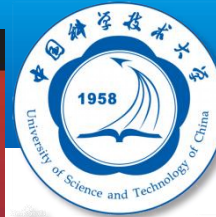
- Flex文件编译

flex flex文件

gcc xx.yy.c -lfl

使用fl库进行
编译

```
1 [TA@TA example]$ flex wc.l
2 [TA@TA example]$ gcc lex.yy.c -lfl
3 [TA@TA example]$ ./a.out
4 hello world
5 ^D
6 look, I find 2 words of 10 chars
7 [TA@TA example]$
```



- 定义Simple语言，该语言仅包含整数变量声明和四则运算
- Simple语言关键字

```
int  
+  
-  
*  
/  
;  
(  
)  
=
```

- 定义Simple语言，该语言仅包含整数变量声明和四则运算
- Simple语言标识符ID和整数NUM

```
letter = [a-zA-Z]  
digit = [0-9]  
ID = letter+  
INTEGER = digit+
```

- 定义Simple语言，该语言仅包含整数变量声明和四则运算
- Simple语言词法分析器文件
 - lexer_main.c
 - simple_analyzer.l



Flex示例-稍复杂的词法解析示例



**simple_
analyzer.**

|

```
%option noyywrap
%{
#include <stdio.h>
#include <stdlib.h>
int lines;
int pos_start;
int pos_end;
#define ADD 1
#define MUL 2
#define DIV 3
#define SUB 4
#define ASSIN 5
#define SEMICOLON 6
#define COMMA 7
#define LPARENTHESIS 8
#define RPARENTHESIS 9
#define INT 10
#define IDENTIFIER 11
#define INTEGER 12
#define ERROR 0
%}
```



Flex示例-稍复杂的词法解析示例



**simple_
analyzer.**

|

%%

```
\+      {pos_start = pos_end; pos_end += 1; return ADD;}
\−      {pos_start = pos_end; pos_end += 1; return SUB;}
\*      {pos_start = pos_end; pos_end += 1; return MUL;}
\\/      {pos_start = pos_end; pos_end += 1; return DIV;}
=        {pos_start = pos_end; pos_end += 1; return ASSIN;}
;         {pos_start = pos_end; pos_end += 1; return SEMICOLON;}
,         {pos_start = pos_end; pos_end += 1; return COMMA;}
\(\      {pos_start = pos_end; pos_end += 1; return LPARENTHESIS;}
\)       {pos_start = pos_end; pos_end += 1; return RPARENTHESIS;}
int       {pos_start = pos_end; pos_end += 3; return INT;}
[a-zA-Z]+ {pos_start = pos_end; pos_end += strlen(yytext); return IDENTIFIER;}
[0-9]+    {pos_start = pos_end; pos_end += strlen(yytext); return INTEGER;}
\n        {lines++; pos_start = 1; pos_end = 1;}
[ \t]     {pos_start = pos_end; pos_end += 1;}
```

/* 其他所有字符 */

```
. { pos_start = pos_end; pos_end++; return ERROR; }
```

%%

lexer_main.c

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

```
extern int lines;
extern int pos_start;
extern int pos_end;
```

```
extern FILE *yyin;
extern char *yytext;
extern int yylex();
```



Flex示例-稍复杂的词法解析示例



lexer_main.c

```
int main(int argc, const char **argv) {

    const char *input_file = argv[1];
    yyin = fopen(input_file, "r");
    if (!yyin) {
        fprintf(stderr, "cannot open file: %s\n", input_file);
        return 1;
    }

    int token;
    printf("%5s\t%10s\t%s\t%s\n", "Token", "Text", "Line", "Column
(Start,End)");
    while ((token = yylex())) {
        printf("%-5d\t%10s\t%d\t(%d,%d)\n",
            token, yytext,
            lines, pos_start, pos_end);
    }
    return 0;
}
```

- **Simple语言词法分析器编译命令**

```
flex simple_analyzer.l  
gcc lexer_main.c lex.yy.c -o simpleFlex
```

- Simple语言词法分析器测试文件
- 正确文本编译

```
int a;  
int B;  
a= 1+(2-3)*4/2;
```

- 编译运行结果

```
$ flex simple_analyzer.l
$ gcc lexer_main.c lex.yy.c -o SimpleFlex
$ ./SimpleFlex test.sim
```

Token (Start,End)	Text	Line	Column
10	int	0	(0,3)
11	a	0	(4,5)
6	;	0	(5,6)
10	int	1	(1,4)
11	B	1	(5,6)
6	;	1	(6,7)
11	a	2	(1,2)
5	=	2	(3,4)
12	1	2	(5,6)
1	+	2	(6,7)
8	(2	(7,8)
12	2	2	(8,9)
4	-	2	(9,10)
12	3	2	(10,11)

- Simple语言词法分析器测试文件
- 错误文本编译

```
int a;  
int B;  
B = 5 % 2;  
a = 1 + ( 2 - 3 ) * 4 / 2;
```

• 编译运行结果

```
$ ./SimpleFlex test_wrong.sim
```

Token	Text	Line	Column (Start,End)
10	int	0	(0,3)
11	a	0	(4,5)
6	;	0	(5,6)
10	int	1	(1,4)
11	B	1	(5,6)
6	;	1	(6,7)
11	B	2	(1,2)
5	=	2	(2,3)
12	5	2	(3,4)



- 正则表达式
- Flex简介
- Bison简介
- Flex和Bison联动



- 1975 ~ 1978年贝尔实验室的S. C. Johnson基于Knuth的LR语法分析理论实现了Yacc (Yet Another Compiler Compiler)
- 二十世纪八十年代，UC Berkeley的研究生Bob Corbett使用改进的内部算法实现了伯克利Yacc
- 来自FSF的Richard Stallman改写了伯克利Yacc并将其用于GNU项目，添加了很多特性，形成了今天的GNU Bison



Bison用于根据给定的语法规则生成语法分析器。Bison通过读取上下文无关文法规则来生成语法分析器

Bison采用上下文无关文法，采用LALR（Look-Ahead Left-to-Right Rightmost derivation）方法进行语法分析

Bison被广泛应用于编译器设计、解析器生成和其他需要进行语法分析的领域

Bison源程序通常以 .y 为后缀



- **Flex和Bison是Linux下用来生成词法分析器和语法分析器的两个常用工具，一般结合使用来处理复杂的文件解析工作**



- 以计算器程序为例，界面输入 $2+2*2$

- Flex将输入识别为token流

将2识别为number,

+识别为ADD,

*识别为MUL

- Bison将输入的token流按照语法规则组织为AST树

输入

$2 + 2 * 2$

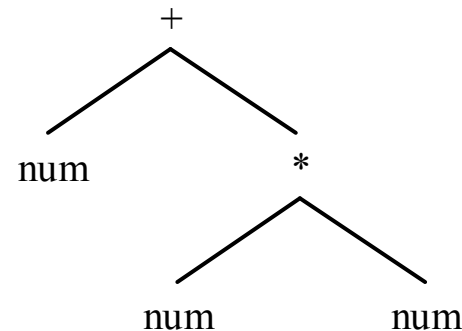
Flex分析器

token流

num ADD num MUL num

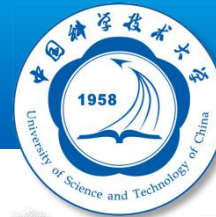
Bison分析器

语法树





Bison demo



```
/* file name : bison_demo.y */
/* Part 1 */
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
%}

%start reimu
%token REIMU

%%

reimu : REIMU { puts("\nFind\n"); }
%%

int yylex(void){
    int c = getchar();
    switch (c) {
        case EOF: return YYEOF;
        case 'H': return REIMU;
        default: return YYUNDEF;
    }
}
```

```
/* file name : bison_demo.y */
/* Part 2 */
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void){
    yyparse(); // 启动解析
    return 0;
}
```

```
→ bison_demo ls
bison_demo.y
→ bison_demo bison bison_demo.y
→ bison_demo ls
bison_demo.tab.c bison_demo.y
→ bison_demo gcc bison_demo.tab.c
→ bison_demo ./a.out
H
Find
```

syntax error



Bison demo



```

/* file name : bison_demo.y */
/* Part 1 */
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
}%

%start reimu
%token REIMU

%%

reimu : REIMU { puts("\nFind\n"); }
%%

int yylex(void){
    int c = getchar();
    switch (c) {
        case EOF: return YYEOF;
        case 'H': return REIMU;
        default: return YYUNDEF;
    }
}

```

```

/* file name : bison_demo.y */
/* Part 2 */
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void){
    yyparse(); // 启动解析
    return 0;
}

```

→ **bison_demo** ls
 bison_demo.y
 → **bison_demo** bison bison_demo.y → **Bison对语法规则进行解析得到C程序**
 → **bison_demo** ls
 bison_demo.tab.c bison_demo.y
 → **bison_demo** gcc bison_demo.tab.c → **编译C程序得到可执行文件**
 → **bison_demo** ./a.out → **运行C程序**
 H → **输入H后, 按 Ctrl + D, 此时执行规约**
 Find **reimu <- REIMU, 并执行动作 puts("\nFind\n")**

syntax error

编译和运行Bison源程序



Bison demo



```
/* file name : bison_demo.y */
```

```
%{  
#include <stdio.h>  
int yylex(void);  
void yyerror(const char *s);  
%}
```

```
%start reimu  
%token REIMU
```

```
%%  
reimu : REIMU { puts("\nFind\n"); }  
%%
```

```
int yylex(void){  
    int c = getchar();  
    switch (c) {  
        case EOF: return YYEOF;  
        case 'H': return REIMU;  
        default: return YYUNDEF;  
    }  
}
```

```
void yyerror(const char *s) {  
    fprintf(stderr, "%s\n", s);  
}
```

```
int main(void){  
    yyparse(); // 启动解析  
    return 0;  
}
```

声明部分

定义部分

规则部分

C代码部分



Bison demo



```
/* file name : bison_demo.y */  
%{  
#include <stdio.h>  
int yylex(void);  
void yyerror(const char *s);  
%}
```

```
%start reimu  
%token REIMU
```

```
%%  
reimu : REIMU { puts("\nFind\n"); }  
%%
```

```
int yylex(void){  
    int c = getchar();  
    switch (c) {  
        case EOF: return YYEOF;  
        case 'H': return REIMU;  
        default: return YYUNDEF;  
    }  
}
```

```
void yyerror(const char *s) {  
    fprintf(stderr, "%s\n", s);  
}
```

```
int main(void){  
    yyparse(); // 启动解析  
    return 0;  
}
```

声明部分

定义部分

规则部分

C代码部分

声明部分包括 C 语言代码、头文件引用、宏定义、全局变量定义和函数声明等内容，位于 %{ 和 %} 之间。



Bison demo



```
/* file name : bison_demo.y */
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
%}

%start reimu
%token REIMU

%%
reimu : REIMU { puts("\nFind\n"); }
%%

int yylex(void){
    int c = getchar();
    switch (c) {
        case EOF: return YYEOF;
        case 'H': return REIMU;
        default: return YYUNDEF;
    }
}

void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void){
    yyparse(); // 启动解析
    return 0;
}
```

声明部分

定义部分

规则部分

C代码部分

定义部分进行终结符和非终结符定义和声明，常见包括

%token、%union、%start、%type、%left、%right等。

- %token: 声明终结符的类型
- %union: 定义联合类型，用于在语法分析过程中传递数据类型信息
- %start: 指定语法分析器的起始符号
- %type: 指定非终结符的数据类型
- %left: 指定左结合的运算符
- %right: 指定右结合的运算符



- **Bison定义部分**

- 运算符的优先级是通过 `%left`、`%right` 和 `%nonassoc` 等指令来定义的

- Bison 中的优先级声明

- `%left`: 声明运算符为左结合 (left-associative)

加法和减法通常是左结合的, $a + b + c$ 等价于 $(a + b) + c$

- `%right`: 声明运算符为右结合 (right-associative)

赋值运算符通常是右结合的, $a = b = c$ 等价于 $a = (b = c)$

- `%nonassoc`: 声明运算符为非结合 (non-associative)

`%nonassoc`运算符不能连续使用

例: 比较运算符 (如 `==` 和 `!=`) 通常是非结合的, $a == b == c$ 是非法的



- **Bison定义部分**

- Bison 中的优先级顺序是从低到高。越早声明的运算符优先级越低

```
%left '+' '-'  
%left '*' '/'
```

* 和 / 的优先级高于 + 和 -



Bison demo



```
/* file name : bison_demo.y */  
%{  
#include <stdio.h>  
int yylex(void);  
void yyerror(const char *s);  
%}
```

声明部分

```
%start reimu  
%token REIMU
```

定义部分

```
%%  
reimu : REIMU { puts("\nFind\n"); }  
%%
```

规则部分

```
int yylex(void){  
    int c = getchar();  
    switch (c) {  
        case EOF: return YYEOF;  
        case 'H': return REIMU;  
        default: return YYUNDEF;  
    }  
}
```

C代码部分

```
void yyerror(const char *s) {  
    fprintf(stderr, "%s\n", s);  
}
```

```
int main(void){  
    yyparse(); // 启动解析  
    return 0;  
}
```

规则部分由归约规则和动作组成。规则基本按照巴科斯范式（BNF）描述。规则中目标或非终端符放在左边，后跟一个冒号：然后是产生式的右边，之后是对应的动作（用 {} 包含）



Bison demo



```
/* file name : bison_demo.y */  
%{  
#include <stdio.h>  
int yylex(void);  
void yyerror(const char *s);  
%}
```

声明部分

```
%start reimu  
%token REIMU
```

定义部分

```
%%  
reimu : REIMU { puts("\nFind\n"); }  
%%
```

规则部分

```
int yylex(void){  
    int c = getchar();  
    switch (c) {  
        case EOF: return YYEOF;  
        case 'H': return REIMU;  
        default: return YYUNDEF;  
    }  
}
```

C代码部分

```
void yyerror(const char *s) {  
    fprintf(stderr, "%s\n", s);  
}
```

```
int main(void){  
    yyparse(); // 启动解析  
    return 0;  
}
```

C代码部分为C代码，会被原样复制到 Bison 生成的C文件中，这里一般自定义一些函数。主要包括调用 Bison 的语法分析程序 yyparse()。其中 yyparse 函数由 Bison 根据语法规则自动生成，用于语法分析。



Bison demo



```
/* file name : bison_demo.y */
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
}%

%start reimu
%token REIMU

%%
reimu : REIMU { puts("\nFind\n"); }
%%
```

```
int yylex(void){
    int c = getchar();
    switch (c) {
        case EOF: return YYEOF;
        case 'H': return REIMU;
        default: return YYUNDEF;
    }
}
```

```
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
```

```
int main(void){
    yyparse(); // 启动解析
    return 0;
}
```

yylex和yyparse的关系： yyparse 函数在需要词法单元时调用 yylex 函数，并从 yylex 返回的词法单元中获取相关信息进行语法分析。这样，yyparse 可以根据词法分析器提供的词法单元逐步解析输入。

```
int yyparse (void)
{
    .....

    if (yychar == YYEMPTY)
    {
        YYDPRINTF ((stderr, "Reading a token\n"));
        yychar = yylex ();
    }

    .....
}
```

生成的yyparse函数中，调用yylex函数得到相关信息



• Bison指针模型

- 为了准确描述Bison语法可能会出现的冲突，引入指针模型
- 模型的内容如下：每次读到一个记号（token），有一个指针在Bison程序中移动
- 开始时，这个指针（用“↑”表示）在起始规则的第一项前

```
%token A B C D E F  
%%  
start: ↑ A B C ;
```



• Bison指针模型

- 为了准确描述Bison语法可能会出现的冲突，引入指针模型
- 模型的内容如下：每次读到一个记号（token），有一个指针在Bison程序中移动
- 开始时，这个指针（用“↑”表示）在起始规则的第一项前
- 当Bison语法分析器输入记号时，指针会相应移动，有时指针不止一个

```
%token A B C D E F
%%
start:    x
        |  y ;
x: A B ↑ C D;
y: A B ↑ E F;
```




- **Bison冲突类型模型**

- 归约/归约冲突

- 基于一个指针归约时，如果另一个指针也在归约，这个情况就是归约/归约冲突

```
start:    x  
        |    y ;
```

```
x: A ↑;
```

```
y: A ↑;
```



- **Bison冲突类型模型**

- 移进/归约冲突

- 基于一个指针归约，如果另一个指针正在移进，这个情况就是移进/归约冲突

```
start:    x  
        |  y R;
```

```
x: A ↑ R;
```

```
y: A ↑;
```



• Bison分析器状态

- 通过指定-v选项（即verbose模式），将生成详细的解析器信息以辅助调试
- 创建一个*.output文件，其中包含以下关键信息：语法分析表、状态信息以及潜在的移进/规约冲突

```
$ bison -v calc.y  
$ ls *.output # 查看导出文件  
calc.output
```



- **Bison语法分析表**

- 二义性例子

```
start:  A B x Z;  
      |  y Z;  
x: C;  
y: A B C;
```



```
state 7  
  x: C ↑                      //rule 3  
  y: A B C ↑                  //rule 4  
Z reduce using rule 3(x)  
Z [reduce using rule 4(y)]  
$default reduce using rule 3(x)
```



• Bison语法分析表

• 二义性例子1

```
start:  A B x Z;  
      |  y Z;  
x: C;  
y: A B C;
```



```
state 7  
  x: C ↑                               //rule 3  
  y: A B C ↑                           //rule 4  
Z reduce using rule 3(x)  
Z [reduce using rule 4(y)]  
$default reduce using rule 3(x)
```

Bison采用更早出现的规则优先级更高的策略来解决归约/归约冲突

若发生解析冲突，未被选中的规则将显示在方括号内以供参考。



• Bison语法分析表

• 二义性例子2

```
start:  x ;  
      |  y R;  
x:  A R;  
y:  A;
```



```
state 1  
  x: A ↑ R      //rule3  
  y: A ↑        //rule4  
R shift, and go to state 5  
      //using rule 3  
R [reduce using rule 4(y)]
```



• Bison语法分析表

• 二义性例子2

```
start:  x ;  
      |  y R;  
x:  A R;  
y:  A;
```



```
state 1  
  x: A ↑ R      //rule3  
  y: A ↑        //rule4  
R shift, and go to state 5  
           //using rule 3  
R [reduce using rule 4(y)]
```

更早出现的规则
优先级更高



- Cminusf语法中常见的冲突及解决办法

$\text{expr} - \text{expr} - \text{expr}$



$(\text{expr} - \text{expr}) - \text{expr}$
 $\text{expr} - (\text{expr} - \text{expr})$

产生移进/归约冲突
采用更早出现的规则



• Cminusf语法中常见的冲突及解决办法

• 负号

```
%left '+' '-'  
%left '*' '/'  
%left UMINUS  
%%  
expr: expr '+' expr  
      | expr '-' expr  
      | expr '*' expr  
      | expr '/' expr  
      | UMINUS expr  
      | INT
```

UMINUS表示负号



- Cminuf语法中常见的冲突及解决办法

- if else文法

```
statement: IF '(' expression ')' statement  
          | IF '(' expression ')' statement ELSE statement  
          | other_statement
```



```
statement: IF '(' expression ')' statement ↑ ;  
statement: IF '(' expression ')' statement ↑ ELSE statement ;
```

产生移进
/归约冲
突



- **Cminuf**语法中常见的冲突及解决办法

- if else文法

IF '(' expression ')' IF '(' expression ')' statement ELSE statement



IF '(' expression ')'{ IF '(' expression ')' statement} ELSE statement
IF '(' expression ')' {IF '(' expression ')' statement ELSE statement }

两种解释



- **Cminuf**语法中常见的冲突及解决办法

- if else文法 显示指定优先级

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%
statement: IF '(' expression ')' statement %perc LOWER_THAN_ELSE
//%perc说明%perc之前的IF '(' expression ')' statement
//和之后的LOWER_THAN_ELSE优先级相同。
        | IF '(' expression ')' statement ELSE statement
        | other_statement
```



- **Cminuf**语法中常见的冲突及解决办法

- if else 文法

IF '(' expression ')' IF '(' expression ')' statement ELSE statement



~~IF '(' expression ')'~~ { IF '(' expression ')' statement } ELSE statement
IF '(' expression ')' { IF '(' expression ')' statement ELSE statement }



• 二义性处理机制小结

• 移进-规约冲突

- 默认会选择移进操作 (shift)，而不是规约。如果定义了运算符的优先级和结合性，Bison 会根据这些信息决定是移入还是规约

• 规约-规约冲突

- 会选择最早定义的规则进行规约

• 优先级和结合性

- 通过 `%left`、`%right` 和 `%nonassoc` 定义运算符优先级和结合性

• 自定义优先级

- 通过 `%left`、`%right` 等语法定义的优先级，写在越后面，优先级越高



- 正则表达式
- Flex简介
- Bison简介
- Flex和Bison联动



- **Bison是解析器生成器**
- **将LALR文法转换成可编译的C代码**
- **Bison与Flex关系**
 - Bison-解析器生成器，主导地位
 - Flex-辅助工具，用来生成yylex函数
 - yylex函数在yyparse函数执行过程中被调用



- 利用Flex和Bison完成一个四则运算的计算器

- 目录结构

- calc.l flex文件
- calc.y bison文件
- driver.c 驱动文件



Flex & Bison demo



calc.y bison文件 **声明部分**

```
%{  
#include <stdio.h>  
    int yylex(void);  
    void yyerror(const char *s);  
%}
```

定义部分

```
%token RET  
%token <num> NUMBER  
%token <op> ADDOP MULOP LPAREN  
RPAREN  
%type <num> top line expr term factor  
  
%start top  
  
%union {  
    char  op;  
    double num;  
}
```



Flex & Bison demo



calc.y bison文件

符号定义

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
%}

%token RET
%token <num> NUMBER
%token <op> ADDOP MULOP LPAREN RPAREN
%type <num> top line expr term factor

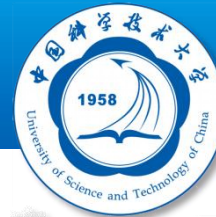
%start top

%union {
    char op;
    double num;
}
```

token 定义
start 开始符号
大写字母 终结符(Flex文件解析)
小写字母 非终结符



Flex & Bison demo



calc.y bison文件

开始符号

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(const char *s);
}%

%token RET
%token <num> NUMBER
%token <op> ADDOP MULOP LPAREN RPAREN
%type <num> top line expr term factor

%start top

%union {
    char op;
    double num;
}
```

token 定义

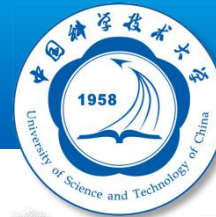
start 开始符号

大写字母 终结符(Flex文件解析)

小写字母 非终结符



Flex & Bison demo



calc.y bison文件

```
%{  
    #include <stdio.h>  
    int yylex(void);  
    void yyerror(const char *s);  
}%  
  
%token RET  
%token <num> NUMBER  
%token <op> ADDOP MULOP LPAREN RPAREN  
%type <num> top line expr term factor  
  
%start top  
  
%union {  
    char op;  
    double num;  
}
```

语义值集合

union定义了语法符号的语义值类型的集合



Flex & Bison demo



calc.y bison文件 **规则部分**

```
%%  
top : top line {  
| {  
  
line : expr RET  
{  
    printf(" = %f\n", $1);  
}  
  
expr : term  
{  
    $$ = $1;  
}  
| expr ADDOP term  
{  
    switch ($2) {  
    case '+': $$ = $1 + $3; break;  
    case '-': $$ = $1 - $3; break;  
    }  
}  
}
```

\$\$ 当前节点

\$1 已解析的第一个节点

\$2 已解析的第二个节点

...



Flex & Bison demo



calc.y bison文件 **规则部分**

```
%%  
top : top line {  
| {  
  
line : expr RET  
{  
    printf(" = %f\n", $1);  
}  
  
expr : term  
{  
    $$ = $1;  
}  
| expr ADDOP term  
{  
    switch ($2) {  
    case '+': $$ = $1 + $3; break;  
    case '-': $$ = $1 - $3; break;  
    }  
}  
}
```

\$\$ 当前节点

\$1 已解析的第一个节点

\$2 已解析的第二个节点

...



Flex & Bison demo



calc.y bison文件 **规则部分**

```
%%  
top : top line {  
| {  
  
line : expr RET  
{  
    printf(" = %f\n", $1);  
}  
  
expr : term  
{  
    $$ = $1;  
}  
| expr ADDOP term  
{  
    switch ($2) {  
    case '+': $$ = $1 + $3; break;  
    case '-': $$ = $1 - $3; break;  
    }  
}
```

\$\$ 当前节点

\$1 已解析的第一个节点

\$2 已解析的第二个节点

...



Flex & Bison demo



calc.y bison文件 **规则部分**

```
%%
```

```
top : top line {}  
| {}
```

```
line : expr RET  
{  
    printf(" = %f\n", $1);  
}
```

```
expr : term  
{  
    $$ = $1;  
}  
| expr ADDOP term  
{  
    switch ($2) {  
        case '+': $$ = $1 + $3; break;  
        case '-': $$ = $1 - $3; break;  
    }  
}
```

\$\$ 当前节点

\$1 已解析的第一个节点

\$2 已解析的第二个节点

...



Flex & Bison demo



calc.y bison文件 **规则部分**

```
%%  
top : top line {  
| {  
  
line : expr RET  
{  
    printf(" = %f\n", $1);  
}  
  
expr : term  
{  
    $$ = $1;  
}  
| expr ADDOP term  
{  
    switch ($2) {  
    case '+': $$ = $1 + $3; break;  
    case '-': $$ = $1 - $3; break;  
    }  
}  
}
```

\$\$ 当前节点

\$1 已解析的第一个节点

\$2 已解析的第二个节点

...



Flex & Bison demo



calc.y bison文件

```
term : factor {  
    $$ = $1;  
}  
| term MULOP factor {  
    switch ($2) {  
        case '*': $$ = $1 * $3; break;  
        case '/': $$ = $1 / $3; break; // 这里会出什么问题?  
    }  
}  
  
factor : LPAREN expr RPAREN  
{  
    $$ = $2;  
}  
| NUMBER  
{  
    $$ = $1;  
}  
%%
```



Flex & Bison demo

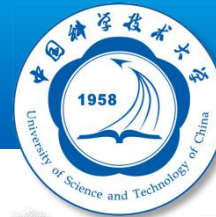


calc.y bison文件C代码部分

```
void yyerror(const char *s)
{
    fprintf(stderr, "%s\n", s);
}
```



Flex & Bison demo



calc.l flex文件

```
/* calc.l */  
%option noyywrap
```

声明部分

%%

规则部分

%%

C代码部分

```
%{  
/* 引入 calc.y 定义的 token */  
#include "calc.tab.h"  
%}  
  
%%  
  
\( { return LPAREN; }  
\) { return RPAREN; }  
"+"|"-" { yylval.op = yytext[0]; return ADDOP; }  
"*"|"/" { yylval.op = yytext[0]; return MULOP; }  
[0-9]+|[0-9]+\.[0-9]*|[0-9]*\.[0-9]+ { yylval.num = atof(yytext); return  
NUMBER; }  
" |\t { }  
\r\n|\n|\r { return RET; }  
%%
```



Flex & Bison demo



calc.l flex文件

与Bison文件
交互

声明部分

%%

规则部分

%%

C代码部分

```
/* calc.l */
%option noyywrap

%{
/* 引入 calc.y 定义的 token */
#include "calc.tab.h"
}%

%%
\ ( { return LPAREN; }
\) { return RPAREN; }
"+"|"-" { yylval.op = yytext[0]; return ADDOP; }
"*"|"/" { yylval.op = yytext[0]; return MULOP; }
[0-9]+|[0-9]+\.[0-9]*|[0-9]*\.[0-9]+ { yylval.num = atof(yytext);
                                         return NUMBER; }

" "\t { }
\r\n|\n|\r { return RET; }

%%
```



Flex & Bison demo



diver.c

驱动文件

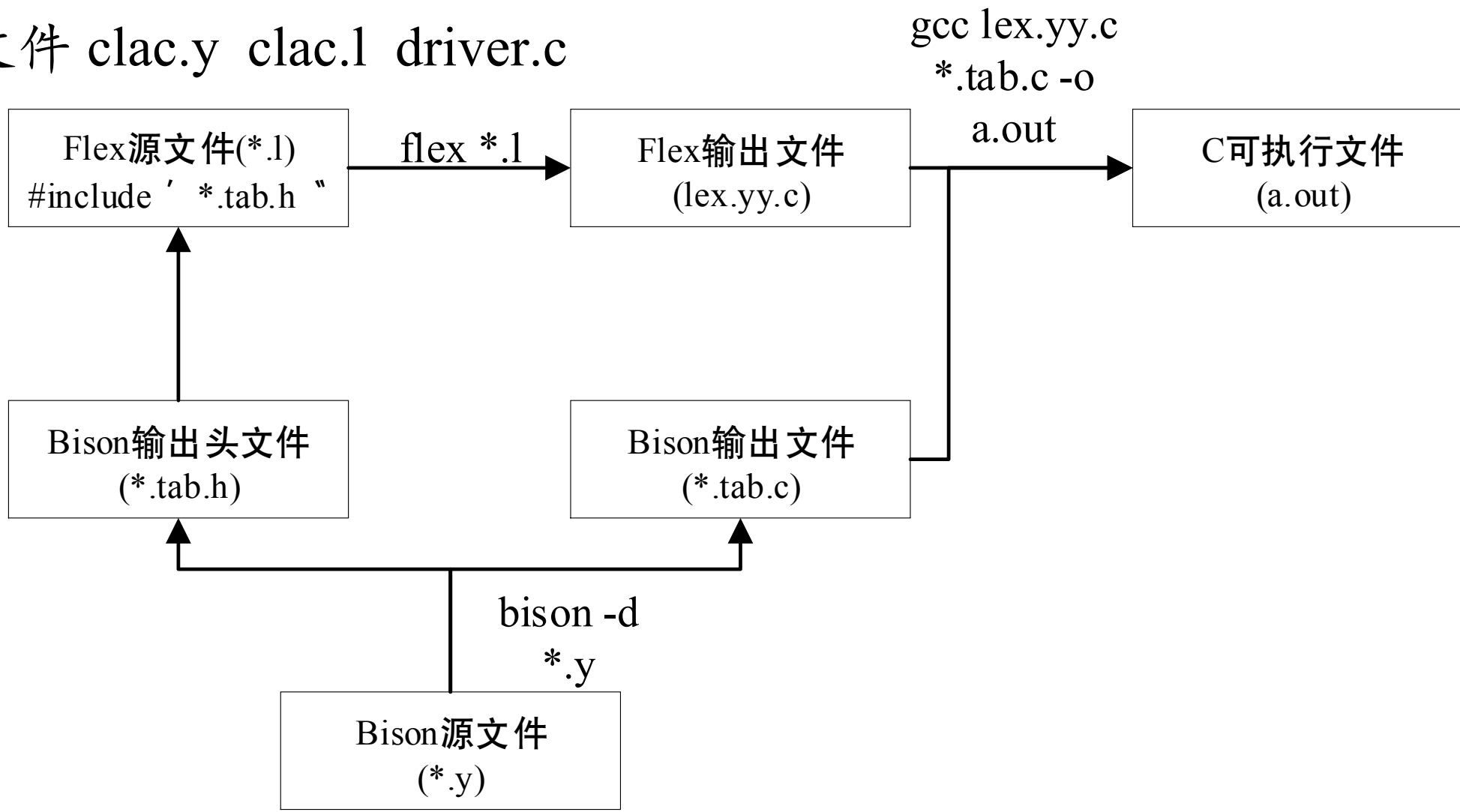
```
int yyparse();
```

```
int main()
{
    yyparse();
    return 0;
}
```



• Flex与Bison协同工作完成四则运算 编译

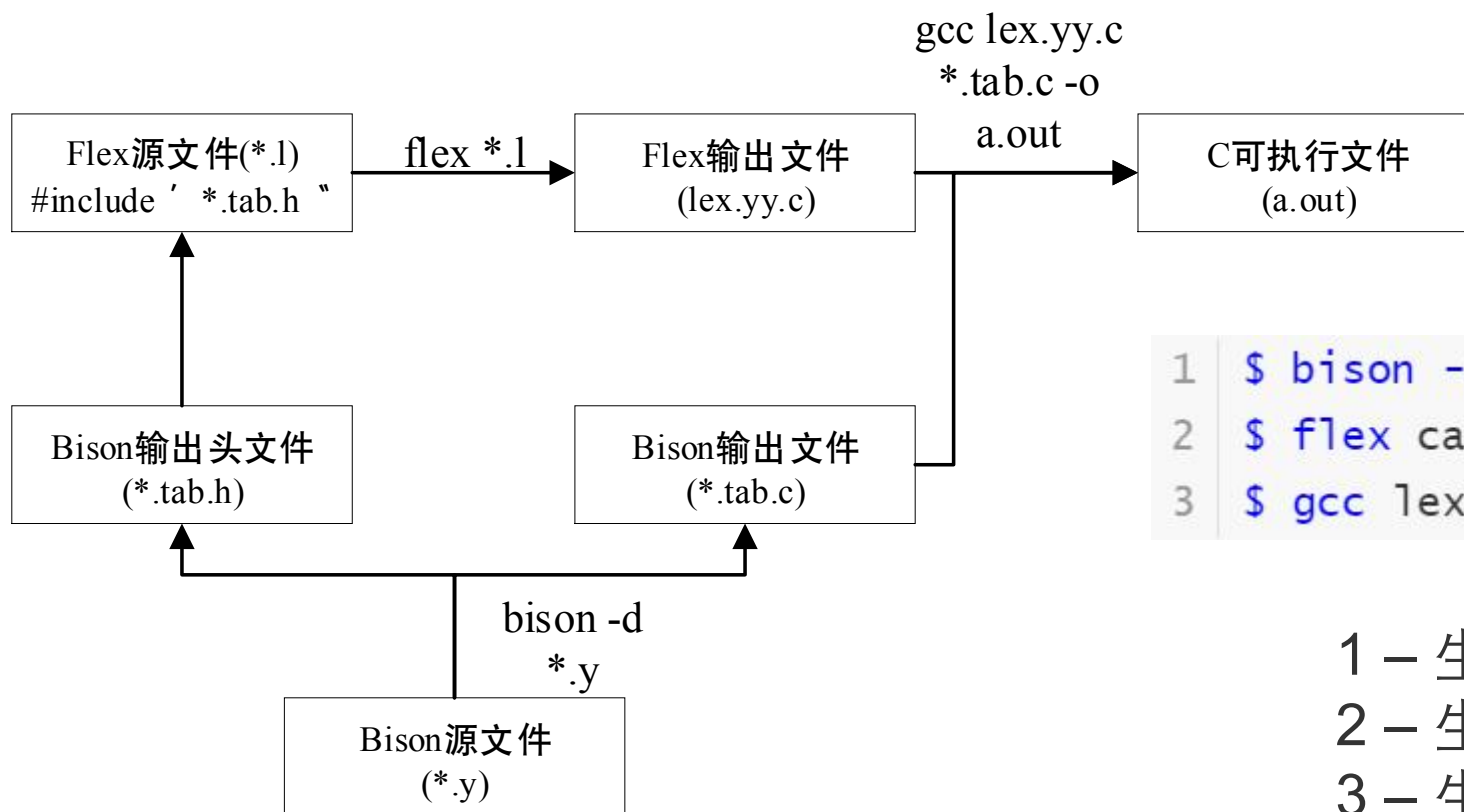
- 文件 clac.y clac.l driver.c





• Flex与Bison协同工作完成四则运算 编译

- 文件 `calc.y` `calc.l` `driver.c`

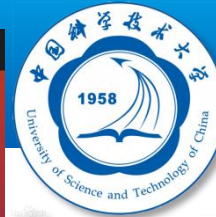


```
1 $ bison -d calc.y
2 $ flex calc.l
3 $ gcc lex.yy.c calc.tab.c driver.c -o calc
```

- 1 – 生成`calc.tab.c`和`calc.tab.h`
- 2 – 生成`lex.yy.c`
- 3 – 生成`calc`可执行文件



File & Directory



Activities Terminal Aug 10 22:46

xxxwww

Trash

2021

Recent

Starred

Home

Desktop

Documents

Downloads

calc.l calc.y driver.c

xxxwww@ubuntu: ~/Desktop/2021/code/bison

```
xxxwww@ubuntu:~/Desktop/2021/code/bison$
```



Flex & Bison demo



尝试使用Flex和Bison实现以下一个极简语法，以识别一个最简单的程序并输出其语法树。

语法：

- `program -> int main() { statements }`
- `statements -> statement`
- `statement -> ; | return ;`

程序：

```
int main() {  
    return;  
}
```

```
int main() {  
    ;  
}
```



• 极简语言解析器文件

- `syntax_tree.h` 语法树头文件
- `syntax_tree.c` 语法树源文件
- `flex.l` 词法解析文件
- `bison.y` 语法解析文件
- `main.c` 主文件



- 语法树头文件 `syntax_tree.h`

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
```

```
struct TreeNode
{
    char *type;
    struct TreeNode *left;
    struct TreeNode *right;
};
```

```
struct TreeNode *createTreeNode(char *type);
```

```
void printTree(struct TreeNode *node, int level);
```

```
struct TreeNode *parse();
```



- 语法树源文件 `syntax_tree.c`

```
#include "syntax_tree.h"

struct TreeNode *createTreeNode(char *type)
{
    struct TreeNode *node = (struct TreeNode
*)malloc(sizeof(struct TreeNode));
    node->type = type;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```



- 语法树源文件 `syntax_tree.c`

```
void printTree(struct TreeNode *node, int level)
{
    if (node == NULL)
    {
        return;
    }

    for (int i = 0; i < level; i++)
    {
        printf(" ");
    }

    printf("%s\n", node->type);
    printTree(node->left, level + 1);
    printTree(node->right, level + 1);
}
```



Flex & Bison demo



```
program -> int main() { statements }  
statements -> statement  
statement -> ; | return ;
```

Flex程序识别各种token

- 关键字：int、main、return
- 符号：{、}、（、） 、 ；
- 换行符等：\n、.....



Flex程序 flex.l

```
%option noyywrap
%{
#include "bison.tab.h" // 引入Bison生成的头文件
%}

%%
"int"      { return INT; }
"main"     { return MAIN; }
"("        { return LPAREN; }
")"        { return RPAREN; }
"{"        { return LBRACE; }
"}"        { return RBRACE; }
";"        { return SEMICOLON; }
"return"   { return RETURN; }
\n         { /* 忽略 */ }
.          { /* 忽略 */ }
%%
```

在Bison程序中定义，
由bison.tab.h传递给Flex程序



Bison程序 bison.y

1. 实现语法

```
program -> int main() { statements }  
statements -> statement  
statement -> ; | return ;
```

```
program : INT MAIN LPAREN RPAREN LBRACE statements RBRACE;  
statements : statement;  
statement : SEMICOLON | RETURN SEMICOLON;
```



Bison程序 bison.y

2. 定义token

```
struct TreeNode {  
    char* type;  
    struct TreeNode* left;  
    struct TreeNode* right;  
};
```

.....

```
%union {  
    struct TreeNode* node;  
    char op;  
}
```

```
%token INT MAIN LPAREN RPAREN LBRACE RBRACE SEMICOLON NEWLINE RETURN  
%type <node> program statement statements  
%start program
```



Bison程序 bison.y

3. 结合语法定义动作，构建语法树

```
program : INT MAIN LPAREN RPAREN LBRACE statements RBRACE
        { gt = createTreeNode("Program"); // gt 是一个全局变量
          $$ = gt;
          $$->left = $6; };

statements : statement { $$ = createTreeNode("statements");
                       $$->left = $1; };

statement : SEMICOLON
          { $$ = createTreeNode("EmptyStatement"); }
          | RETURN SEMICOLON
          { $$ = createTreeNode("ReturnStatement"); };
```



Bison程序 bison.y

```
/* file name bison.y part1 声明与定义部分 */
%{
#include <stdio.h>
#include "syntax_tree.h"
int yylex(void);
void yyerror(const char *s);
int yyparse();
struct TreeNode *gt;          // Global syntax tree
%}
```

```
%union { struct TreeNode* node; char op; }
```

```
%token INT MAIN LPAREN RPAREN LBRACE RBRACE SEMICOLON NEWLINE RETURN
%type <node> program statement statements
%start program
```



Bison程序 bison.y

```
/* file name bison.y part2 规则部分 */
```

```
%%
```

```
program : INT MAIN LPAREN RPAREN LBRACE statements RBRACE
```

```
    { gt = createTreeNode("Program"); // gt 是一个全局变量
```

```
      $$ = gt;
```

```
      $$->left = $6; };
```

```
statements : statement { $$ = createTreeNode("statements");
```

```
                $$->left = $1; };
```

```
statement : SEMICOLON
```

```
    { $$ = createTreeNode("EmptyStatement"); }
```

```
    | RETURN SEMICOLON
```

```
    { $$ = createTreeNode("ReturnStatement"); };
```

```
%%
```



Bison程序 bison.y

```
/* file name bison.y part3 C代码部分 */  
struct TreeNode *parse()  
{  
    yyparse();  
    return gt;  
}  
  
void yyerror(const char *s)  
{  
    fprintf(stderr, "%s\n", s);  
}
```



Flex & Bison demo



主程序 main.c

```
#include "syntax_tree.h"

int main(void)
{
    struct TreeNode *tree = parse();
    printTree(tree, 0);
}
```




编译与运行

编译

#!/bin/bash

bison -d bison.y

flex flex.l

gcc lex.yy.c bison.tab.c main.c syntax_tree.c # 生成可执行文件 a.out

运行

→ cat input.txt

```
int main(){ return; }
```

→ ./a.out < input.txt

Program

Statements

ReturnStatement

1 – 生成bison.tab.c和bison.tab.h

2 – 生成lex.yy.c

3 – 生成calc可执行文件



Flex & Bison demo



更多关于Flex的内容:

→ info flex (在命令行中执行)

更多关于Bison的内容:

- <https://www.gnu.org/software/bison/manual/bison.html>



- 有且仅有一个名为main的主函数定义，可以包含若干全局变量声明、常量声明和其他函数定义
- Cminusf语言支持int/float类型和int/float类型的一维数组类型，int型整数为32位有符号数，float为32位单精度浮点数



语言解析器实验介绍



- **Cminusf语言是实验要实现的编程语言，是C语言的一个子集**
- **每个Cminusf程序的源码存储在一个扩展名为cminus的文件**



• 函数

- 参数: 带参数或不带参数, 参数的类型是int/float或者数组类型
- 返回值: 返回int/float类型的值, 或者不返回值

变量定义

基本类型

函数定义

函数形参

函数形参表

形参说明

var-declaration \rightarrow type-specifier ID; | type-specifier ID[INTEGER];

type-specifier \rightarrow int | float | void

fun-declaration \rightarrow type-specifier ID(params) compound-stmt

params \rightarrow param-list | void

param-list \rightarrow param-list, param | param

param \rightarrow type-specifier ID | type-specifier ID[]



• 变量声明

- 可在一个变量声明语句中声明多个变量或常量
- 声明时可以带初始化表达式
- 所有变量要求先定义再使用
- 在函数外声明的为全局变量，在函数内声明的为局部变量/常量

声明 declaration \rightarrow var-declaration | fun-declaration

变量定义 var-declaration \rightarrow type-specifier ID; | type-specifier ID[INTEGER];



• 语句

- 赋值语句、表达式语句（表达式可以为空）、语句块、if语句、while语句、return语句
- 语句块中可以包含若干变量声明和语句

语句块 $\text{statement-list statement} \mid \varepsilon$

语句 $\text{statement} \rightarrow$
 $\quad \mid \text{compound-stmt}$
 $\quad \mid \text{selection-stmt}$
 $\quad \mid \text{iteration-stmt}$
 $\quad \mid \text{return-stmt}$

表达式语句块 $\text{expression-stmt} \rightarrow \text{expression}; \mid ;$

判断语句块 $\text{selection-stmt} \rightarrow$
 $\quad \text{if}(\text{expression}) \text{ statement}$
 $\quad \mid \text{if}(\text{expression}) \text{ statement else statement}$

循环语句块 $\text{iteration-stmt} \rightarrow \text{while}(\text{expression}) \text{ statement};$

返回语句 $\text{return-stmt} \rightarrow \rightarrow \rightarrow \text{return}; \mid \text{return expression};$



• 表达式

- 基本的算术运算 (+、-、*、/、%) 、 关系运算 (==、!=、<、>、<=、>=)
- 非0表示真、0表示假， 关系或逻辑运算的结果用1表示真、0表示假

关系运算符

relop \rightarrow $<= \mid < \mid > \mid >= \mid == \mid !=$

加减运算式

additive-expression \rightarrow additive-expression addop term \mid term

加减运算符

addop \rightarrow $+ \mid -$

乘除运算式

term \rightarrow term mulop factor \mid factor

乘除运算符

mulop \rightarrow $* \mid /$



- 标识符和整数NUM

- 通过正则表达式定义

letter = a|...|z|A|...|Z

digit = 0|...|9

ID = letter+

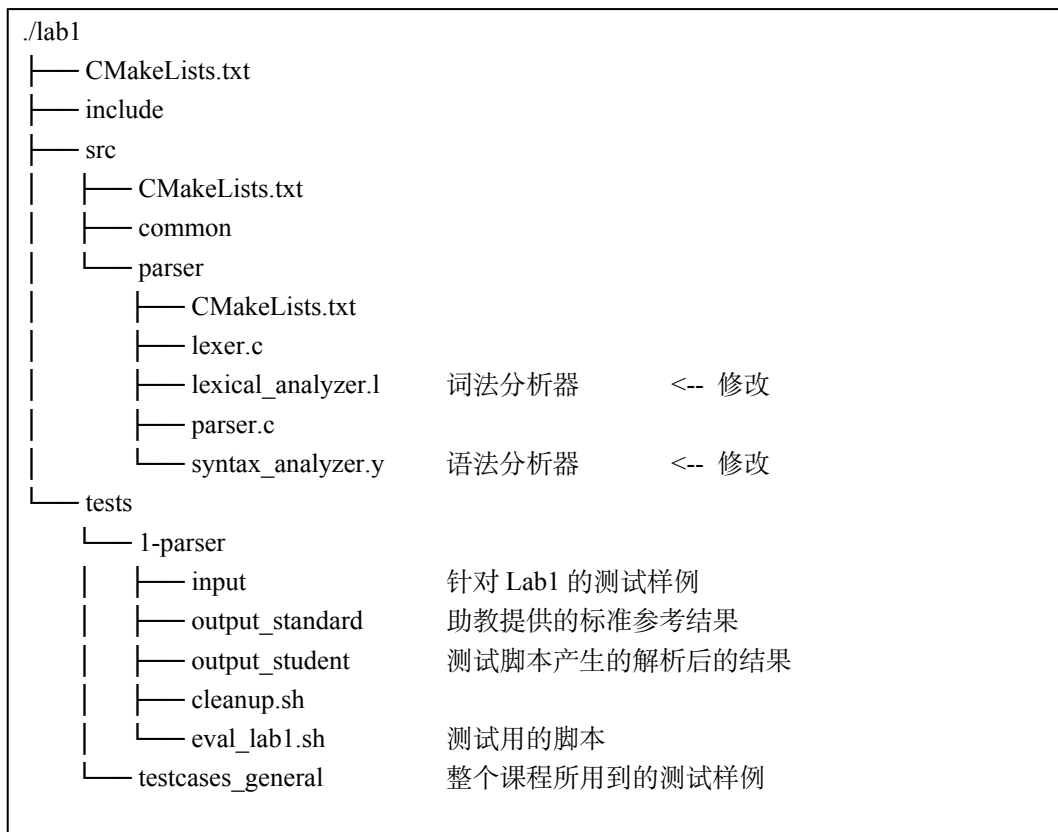
INTEGER = digit+

FLOAT = (digit+. | digit*.digit+)



• 实验要求

- 完成词法分析器： 补全 `src/parser/lexical_analyzer.l`
- 完成语法分析器： 补全 `src/parser/syntax_analyzer.y`



Lab1 相关目录结构



• 实验测试

• 手动测试

- 编译成功后，会在 build 文件夹下找到 lexer 和 parser 可执行文件，用于对 Cminusf 文件进行词法和语法分析；使用方法：lexer/parser <input_file>
- 以 1-return.cminus 为例，运行 lexer 和 parser 的结果如下图所示

```
void main(void) { return; }
```

1-return.cminus

• → testcases_general git:(jianmu202408) lexer 1-return.cminus

Token	Text	Line	Column (Start,End)
282	void	1	(1,5)
284	main	1	(6,10)
272	(1	(10,11)
282	void	1	(11,15)
273)	1	(15,16)
276	{	1	(17,18)
281	return	1	(19,25)
270	;	1	(25,26)
277	}	1	(27,28)

○ → testcases_general git:(jianmu202408) □

lexer 运行示例

lexer 输出词法分析结果

• → testcases_general git:(jianmu202408) parser 1-return.cminus

```
>--+ program
| >--+ declaration-list
| | >--+ declaration
| | | >--+ fun-declaration
| | | | >--+ type-specifier
| | | | | >--+ void
| | | | >--+ main
| | | | >--+ (
| | | | >--+ params
| | | | | >--+ void
| | | | >--+ )
| | | >--+ compound-stmt
| | | | >--+ {
| | | | >--+ local-declarations
| | | | | >--+ epsilon
| | | | >--+ statement-list
| | | | | >--+ statement-list
| | | | | | >--+ epsilon
| | | | | >--+ statement
| | | | | >--+ return-stmt
| | | | | | >--+ return
| | | | | | >--+ ;
| | | | >--+ }
| >--+ }
```

○ → testcases_general git:(jianmu202408) □

parser 运行示例, parser 输出分析树



• 实验测试

• 自动测试

- 实验框架提供了 `tests/1-parser/eval_lab1.sh` 自动化测试脚本，批量执行测试样例，并与助教提供的标准参考结果进行比较

- 脚本的第一个参数代表测试样例，可以是 `easy`、`normal`、`hard` 以及 `testcases_general`
- 脚本的第二个参数是可选的，为 `no`（默认）或者 `yes`，指示是否与标准结果进行比较

• 测试样例

- 通用样例（`tests/testcases_general`）
- Lab1 根据测试难度在目录 `tests/1-parser/input` 下准备了针对性样例：
 - `easy`、`normal`、`hard`



• 实验测试

• 自动测试

```
● → 1-parser git:(master) X ./eval_lab1.sh easy yes
[info] Analyzing FAIL_comment.cminus
error at line 1 column 4: syntax error
[info] Analyzing FAIL_comment2.cminus
error at line 1 column 1: syntax error
[info] Analyzing FAIL_function.cminus
error at line 3 column 1: syntax error
[info] Analyzing FAIL_id.cminus
error at line 1 column 6: syntax error
[info] Analyzing expr.cminus
[info] Analyzing id.cminus
[info] Comparing...
[info] No difference! Congratulations!
○ → 1-parser git:(master) X
```

eval_lab1.sh 使用示例

1

easy 中的全部样例通过

```
● → 1-parser git:(jianmu202408) X ./eval_lab1.sh hard yes
[info] Analyzing You_Should_Pass.cminus
[info] Analyzing assoc.cminus
[info] Analyzing gcd.cminus
error at line 11 column 13: syntax error
[info] Analyzing hanoi.cminus
[info] Analyzing if.cminus
[info] Analyzing selectionsort.cminus
[info] Comparing...
Files /workspace/2023ustc-jianmu-compiler-ta/tests/1-parser/output
t_student/hard/gcd.syntax_tree and /workspace/2023ustc-jianmu-com
piler-ta/tests/1-parser/output_standard/hard/gcd.syntax_tree diff
er
○ → 1-parser git:(jianmu202408) X
```

eval_lab1.sh 使用示例 2

hard 中的样例 gcd 未通过

一起努力 打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年03月20日