

本科实验报告

课程名称： 编译原理

实验名称： 编译器设计实现

姓名/学号： 殷叶航 3150104531   
 　 周　洋 3150104024

所在学院： 计算机科学与技术学院

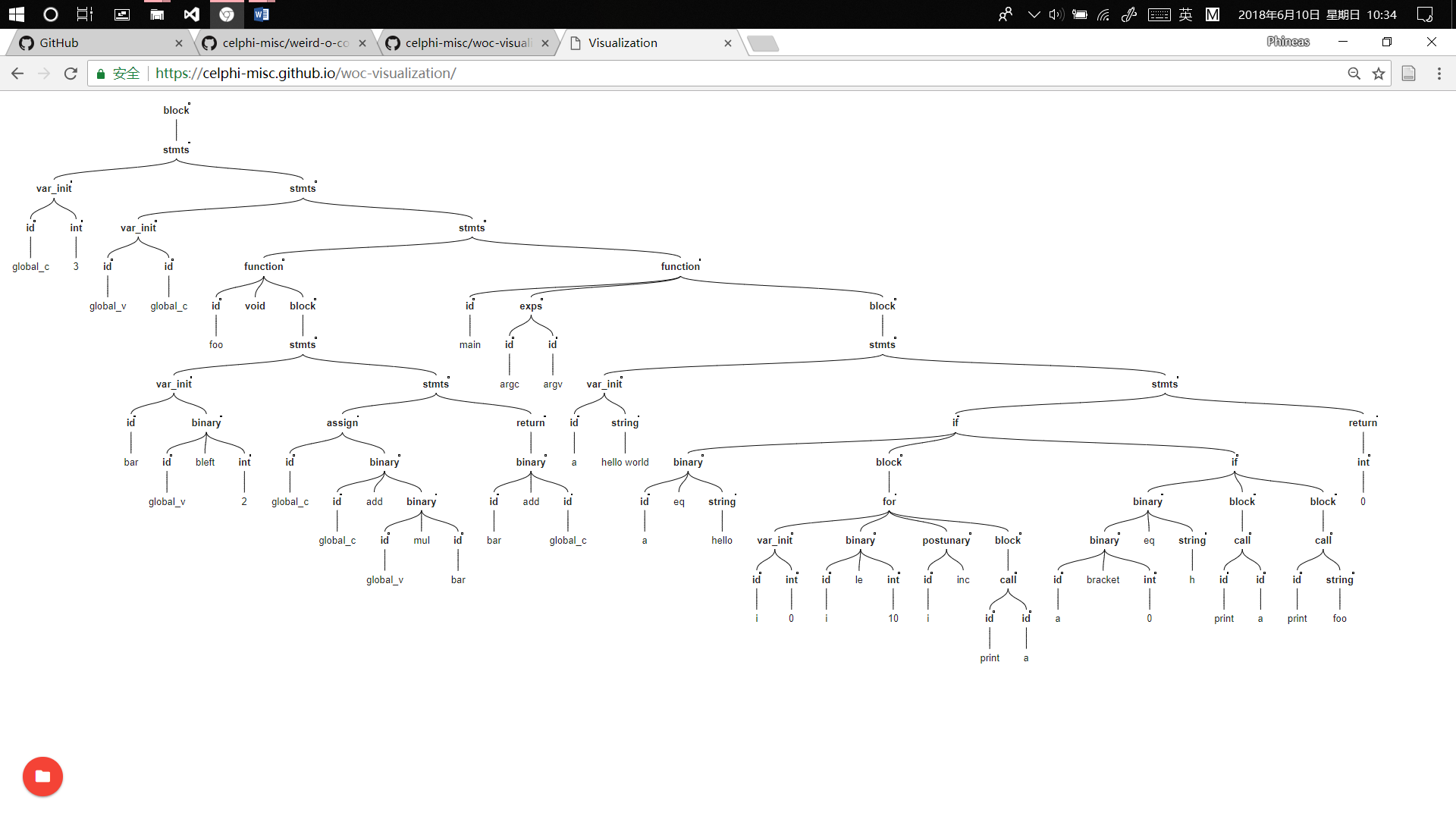
专业类别： 计算机科学与技术

指导教师： 李莹

2018/6/10

Weird-o-Compiler设计实现

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名： | 殷叶航、周洋 | 学号： | 3150104531  3150104024 | 专业： | | 计算机科学与技术 | |
| 课程名称： | 编译原理 | | 同组学生姓名： | |  | |
| 实验时间： | 2018-6-10 | 实验地点： |  | 指导老师： | | 李莹 |



# 成果

我们实现了一个类C语言的编译器，可以用语言源文件构造抽象语法树，并生成中间表示树（以JSON表示）。同时实现了一个基于Web技术的可视化工具，可以将结果绘制为SVG显示在页面上。目前阶段已经实现实验的基本要求，即完成词法分析和语法分析、生成抽象语法树和中间表示。我们也将结果很好地展示了出来。同时，实现还包括了基于Yacc自身错误恢复特性的简单错误恢复。

## 语言定义

语言的定义基本上是C的子集，但是去掉了类型和指针。为了保证语言的完备性，我们借鉴JavaScript加入了部分特性，且允许函数定义在任意位置发生。在后期，我们会引入类似于JavaScript中对象的数据结构抽象。

原本的类型被替换成了变量类型var和函数类型function，在语法层面上，这与引入了诸如int, char, double的类型没有显著差别。后面我们会考虑实现一个解释器作为后端，使其作为一个动态语言。

## 运行环境

Linux或Mac OS X，没有在Windows上测试过。

## 项目概述

项目仓库地址：

* 编译器设计，使用了Lex和Yacc，以及其他一些C语言代码，可以将源代码编译为JSON格式的抽象语法树和IR树。

<https://github.com/celphi-misc/weird-o-compiler>

* 抽象语法树与IR树的可视化：

<https://github.com/celphi-misc/woc-visualization>

* 显示抽象语法树的一个GitHub Page：

<https://celphi-misc.github.io/woc-visualization>

WoC会多遍过源文件，首先会经Lex做词法解析，随后经Yacc做语法解析，并构造抽象语法树。经过翻译，抽象语法树得到了一个中间表示。



## 构建与使用

获得一份源代码仓库的拷贝，在其根目录下执行make，可执行程序为woc。运行woc程序，可以查看到帮助：

celphi-mac:weird-o-compiler celestialphineas$ ./woc

woc [file] [options]

Options:

-a Generating AST JSON

-i Generating IR tree JSON

命令行参数为，先文件名，后选项。选项分别为生成抽象语法树和生成中间表示树。例如：

celphi-mac:weird-o-compiler celestialphineas$ ./woc test.woc -a

可以生成抽象语法树，若在执行过程中没有任何输出，则说明工作一切正常。执行结束之后，将生成文件test.ast.json，即以JSON字符串表示的抽象语法树。该字符串可以通过我们实现的可视化工具打开，查看树的结构。

相类似地，我们也可以执行：

celphi-mac:weird-o-compiler celestialphineas$ ./woc test.woc –i

若源代码没有出现问题，将生成test.ir.json，是以JSON形式表示的中间表示。同样可以通过我们实现的可视化工具打开，查看树的结构。

# 实现

## 项目结构

### 编译器部分

ast\_print.c ast\_print.h 将抽象语法树生成为一个JSON字符串

ast.c ast.h 抽象语法树的结构和通过语义动作生成的函数

ir\_tree.c ir\_tree.h 将抽象语法树生成IR树

parser.y 语法分析Yacc（包含主程序）

scanner.l 词法分析Lex

warn.h 用于报错的宏定义

### 可视化部分

index.html 主页

js/index.js 主程序

css/index.css 样式表

依赖：

* jquery
* materialize.css
* Treant.css

## 构建脚本

该项目使用GNU Make构建，以下是构建脚本的关键部分，从中可以理清项目中各文件的依赖关系。

OBJS=$(PARSER).o $(SCANNER).o $(AST).o ast\_print.o irtree.o

**all**: $(TARGET)

$(TARGET): $(OBJS)

    $(CC) $(OBJS) -o $(TARGET) $(CCLIBS)

**%.o**: %.c

    $(CC) $(CCWARN) $\*.c -c $(CCLIBS)

$(PARSER).c: $(PARSER).y

    $(YACC) $(YACC\_FLAGS) -o $(PARSER).c $(PARSER).y

$(SCANNER).c: $(SCANNER).l

    $(LEX) $(LEX\_FLAGS) -o $(SCANNER).c $(SCANNER).l

$(PARSER).h: $(PARSER).y

$(SCANNER).h: $(SCANNER).l

$(PARSER).o: $(SCANNER).h

$(SCANNER).o: $(PARSER).h

## 词法分析

在我们的项目中，词法分析会将源代码中的所有字符均分析为标记，而不将任何字符直接暴露给词法分析过程。在词法分析过程中，行号和字符位置将被记录。如下所示的YY\_UPDATE宏定义，会在每处标记化被调用，其作用是更新当前列号。

#define YY\_UPDATE yylloc->first\_line = yylloc->last\_line = yylineno;\

yylloc->first\_column = yycolumn; yylloc->last\_column = yycolumn + yyleng - 1;\

yycolumn += yyleng;

%}

如果遇到无法解析的字符，则会警告该错误，跳过字符，继续词法分析过程。

. { **WARN\_INFO**(UNKNOWN\_TOKEN); }

词法分析中的大多数标记都是没有值的，而部分有值的标记，比如数字、字符串等的字面量，会在词法分析的同时转换为数值或字符串。在语法分析的文件中，定义了%union，允许了词法分析在一些地方返回值：

%union

{

char\* stringVal;

long long intVal;

double floatVal;

pNode node;

}

## 语法分析

语法分析过程的同时构造语法分析树，语法分析树的定义在ast.h中。错误检测与恢复则使用了Yacc的机制，当错误出现后，语法分析过程会匹配到最近的右花括号或分号，并及时报错。对语义上存在错误的地方，语法分析过程并不会报错，而将继续。语法分析过程中可能存在的错误，都是关键性的，在抛出错误之后，语法分析不会提供可用的语法分析树（语法分析树中将包含有null）。

例如，在declarations中，我们加入了这样的规则：

declarations : var\_dec SEM declarations { $$ = A\_StmtsExp(yylineno, $1, $3); }

| fun\_dec declarations { $$ = A\_StmtsExp(yylineno, $1, $2); }

| var\_dec SEM { $$ = $1; }

| fun\_dec { $$ = $1; }

| error SEM {}

| error RBRACE {}

;

这就可以允许在遇到错误时，报错并继续语法分析过程。默认的报错行为即：

static void **yyerror**(const char \*msg)

{

**fprintf**(stderr, "%d: %s\n", **yyget\_lineno**(), msg);

}

## 抽象语法树

抽象语法树直接在Yacc程序的语法分析过程中构造。抽象语法树的相关头文件定义了一系列用于表示语法树结构的函数和数据结构，其中的节点定义如下：

typedef struct **Node\_** {

enum

{

A\_NUL, A\_INT, A\_FLOAT, A\_BOOLEAN, A\_STRING,

...

} kind;

Pos pos;

union {

int booleanVal;

long long intVal;

double floatVal;

char \*stringVal;

char \*name;

struct { struct **Node\_**\* exp; Op op; } preUnaryExp;

struct { struct **Node\_**\* exp; Op op; } postUnaryExp;

...

struct { struct **Node\_**\* id; struct **Node\_**\* init; } varInitExp;

struct { struct **Node\_**\* id; } varDecExp;

} u;

} Node;

该结构的节点通过提供的函数可以包装成树状结构，在语法分析过程不出问题的前提下，这棵树中不会存在NULL节点。叶子结点处会使用相应的类型将数据包裹起来。这一设计借鉴自教材虎书。

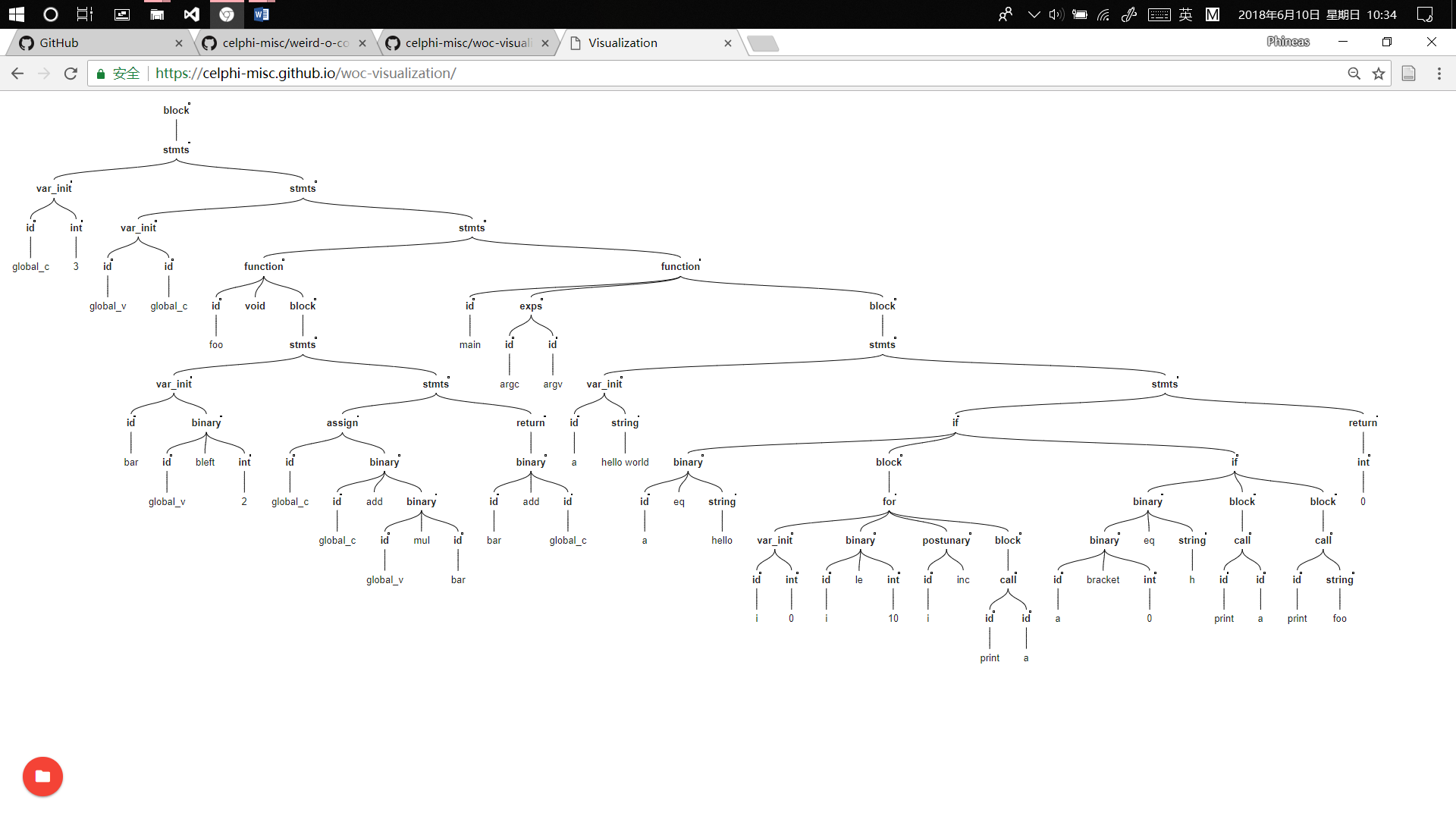
## 中间表示树

【待补充】

## 可视化

可视化使用了开源的树状图SVG库，Treant.js。在可视化树的程序中，我们用JavaScript的原生特性将JSON解析为JavaScript对象，并用自己编写的函数将其转化为Treant对象的初始化参数，最终由Treant.js库完成SVG的绘制。

例如下图：



其对应的源代码为：

var global\_c = 3;

var global\_v = global\_c;

function **foo**() {

var bar = global\_v << 2;

global\_c += global\_v \* bar;

return bar + global\_c;

}

function **main**(argc, argv)

{

var a = "hello world";

if(a == "hello") {

for(var i = 0; i < 10; i++)

{

**print**(a);

}

} else if(a[0] == 'h')

{

**print**(a);

} else {

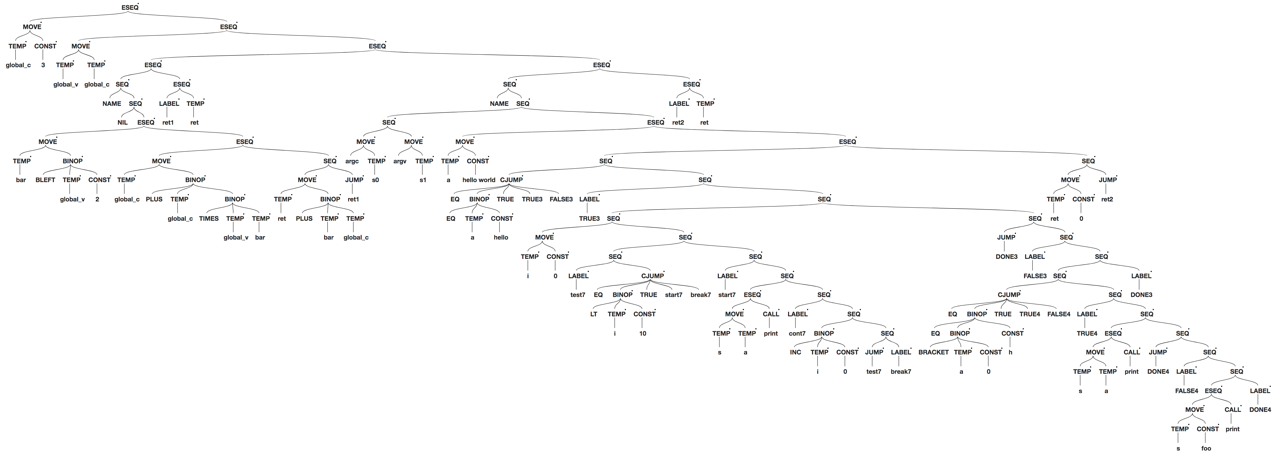
**print**("foo");

}

return 0;

}

由此生成的中间表示为：



# 使用与测试

## 编译

在上文1.4中，我们介绍了如何使用我们的程序编译得到源代码的抽象语法树和中间表示。这里我们以一段没有任何实际意义，但正确的代码为例：

var \_\_var = 3;

var val = \_\_var;

function foo() {

var bar = val << 2;

\_\_var += val \* bar > 0 ? \_\_var : val >> \_\_var;

switch (bar) { case 1: return foo(1); break; case 2: return foo(2); default: return ++bar; }

return bar + \_\_var;

}

function main(argc, argv)

{

var a = "hello world";

function hello(para1, para2) { return null; }

if(a == "hello") {

for(var i = 0; i < 10; i++)

{

print(a);

}

} else if(a[0] == 'h')

{

print(a);

} else {

print("foo");

}

return 0;

}

这段代码中涉及到了各种运算符（包括算数、比较、移位、三目等等），也涉及到了常见的控制语句，甚至是函数内定义函数。我们将看到，实现的程序可以正确地对这段代码作出词法和语法分析，并将结果以JSON的形式输出。

在我们执行命令：

celphi-mac:weird-o-compiler celestialphineas$ ./woc test.woc -a

输出结果为test.ast.json：

{"node":"block","stmts":{"node":"stmts","left":{"node":"var\_init","id":{"node":"id","name":"\_\_var"},"init":{"node":"int","val":3}},"right":{"node":"stmts","left":{"node":"var\_init","id":{"node":"id","name":"val"},"init":{"node":"id","name":"\_\_var"}},"right":{"node":"stmts","left":{"node":"function","id":{"node":"id","name":"foo"},"para\_list":{"node":"void"},"block":{"node":"block","stmts":{"node":"stmts","left":{"node":"var\_init","id":{"node":"id","name":"bar"},"init":{"node":"binary","left":{"node":"id","name":"val"},"op":"bleft","right":{"node":"int","val":2}}},"right":{"node":"stmts","left":{"node":"assign","left":{"node":"id","name":"\_\_var"},"right":{"node":"binary","left":{"node":"id","name":"\_\_var"},"op":"add","right":{"node":"trinary", ...

是不含空白字符的JSON字符串。

我们尝试编译为中间表示：

{"node":"ESEQ","0":{"node":"MOVE","0":{"node":"TEMP","0":{"node":"global\_c"}},"1":{"node":"CONST","0":{"node":"3"}}},"1":{"node":"ESEQ","0":{"node":"MOVE","0":{"node":"TEMP","0":{"node":"global\_v"}},"1":{"node":"TEMP","0":{"node":"global\_c"}}},"1":{"node":"ESEQ","0":{"node":"ESEQ","0":{"node":"SEQ","0":{"node":"NAME"},"1":{"node":"SEQ","0":{"node":"NIL"},"1":{"node":"ESEQ","0":{"node":"MOVE","0":{"node":"TEMP","0":{"node":"bar"}},"1":{"node":"BINOP","0":{"node":"BLEFT"},"1":{"node":"TEMP","0":{"node":"global\_v"}},"2":{"node":"CONST","0":{"node":"2"}}}},"1":{"node":"ESEQ","0":{"node":"MOVE","0":{"node":"TEMP","0":{"node":"global\_c"}},"1":{"node":"BINOP","0":{"node":"PLUS"},"1":{"node":"TEMP","0":{"node":"global\_c"}},"2":{"node":"BINOP","0":{"node":"TIMES"}, ...

这一中间表示的形式也是借鉴自教材虎书。

## 错误检测与恢复

我们尝试在源代码中加入一些错误：

var \_\_var = 3;

var val = \_\_var;

function foo() {

var bar = val << 2;

\_\_var += val \* bar > 0 ? \_\_var : val >> \_\_var;

switch (bar) { case 1: return foo(1); break; case 2:

return ++foo(2); /\* 这里在非左值上自增 \*/ default: return ++bar; }

return bar + \_\_var;

}

function main(argc, argv)

{

var a = "hello world";

function hello(para1, para2) { return null; }

if(a == "hello") {

for(var i = 0; i < 10; i++)

{

print(a) // 这里又少了分号

print("bar");

}

} else if(a[0] == 'h' // 这里括号不配对

{

print(a);

} else {

print("foo");

}

return 0;

}

我们可以测试一下输出：

celphi-mac:weird-o-compiler celestialphineas$ ./woc test.woc -a

9: syntax error, unexpected LPAREN, expecting SEM

21: syntax error, unexpected ID, expecting SEM

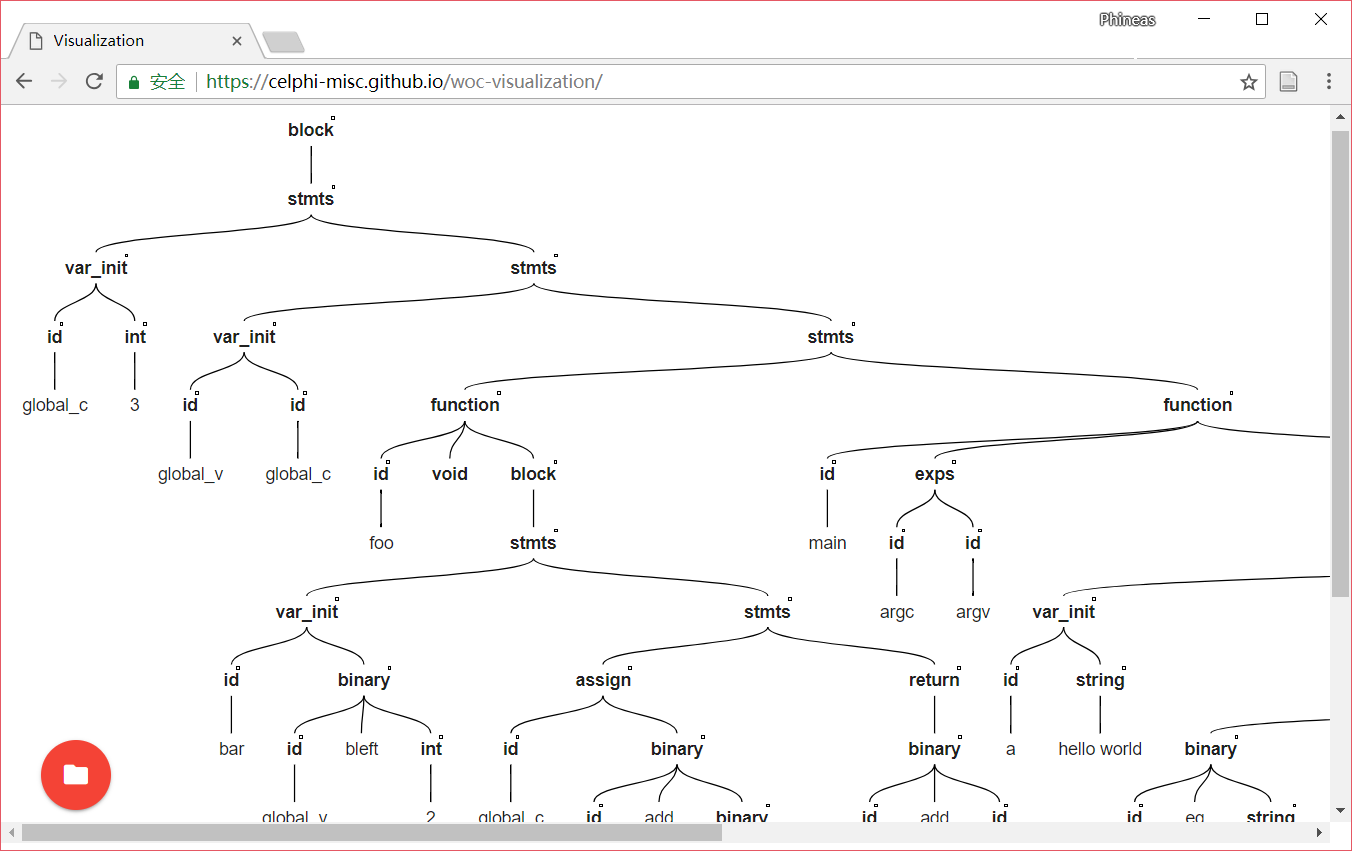
24: syntax error, unexpected LBRACE, expecting RPAREN

Unable to generate AST for file test.woc

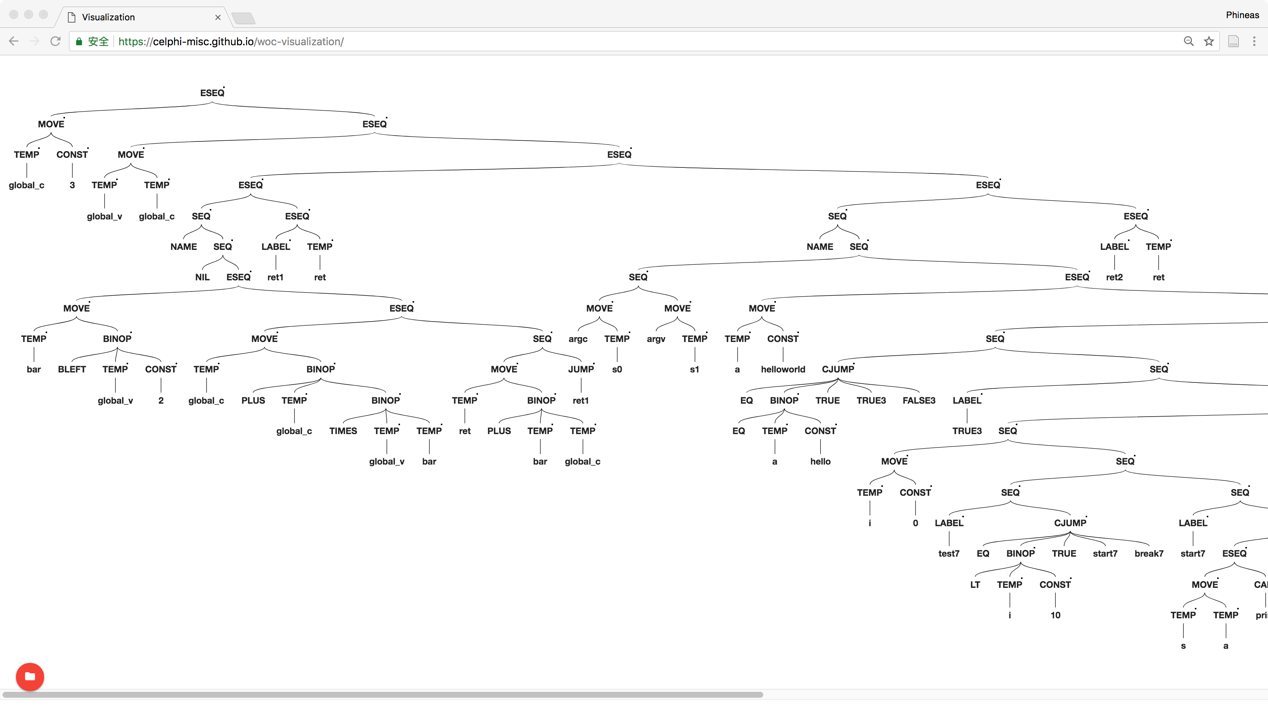
可以看到程序可以找到三处错误。但目前我们的错误恢复还不是很完备，在一些情况下会无法继续向下词法分析。

## 可视化

我们将抽象语法树和中间表示可视化为树状图。打开上文中提到的GitHub page，



在左下角打开文件的图标处可以打开编译器生成的JSON，随即就可以在窗口中绘制出来抽象语法树或中间表示。下图是打开中间表示的结果：



树上的这些节点也可以折叠，从而方便查看。

# 进一步工作

* 实现非原子的对象。（目前仅有数值和字符串这样的原子类型）
* 完成一个中间表示的解释器作为后端，让代码运行在解释器上。（这样省去了寄存器分配，也保证了平台无关）
* 将项目做成一个B/S的Web应用，允许用户与浏览器交互编程，并看到自己代码的语法分析树和中间表示。

# 团队分工

* 词法分析：殷叶航
* 语法分析：殷叶航
* 抽象语法树：银叶航、周洋
* 中间表示：周洋
* 将结果可视化：殷叶航