# 类C编译器

小组成员：林奇恒3160102242，吴涵杰3161012244，张思淡3160102245。

张思淡负责lex和yacc的编写，林奇恒负责CodeGen的编写，吴涵杰负责ObjGen和语法树可视化。

## 支持的数据类型包括：

* void
* int
* double
* char
* string
* bool
* 数组（包括多维数组）

## 支持的主要语法包括：

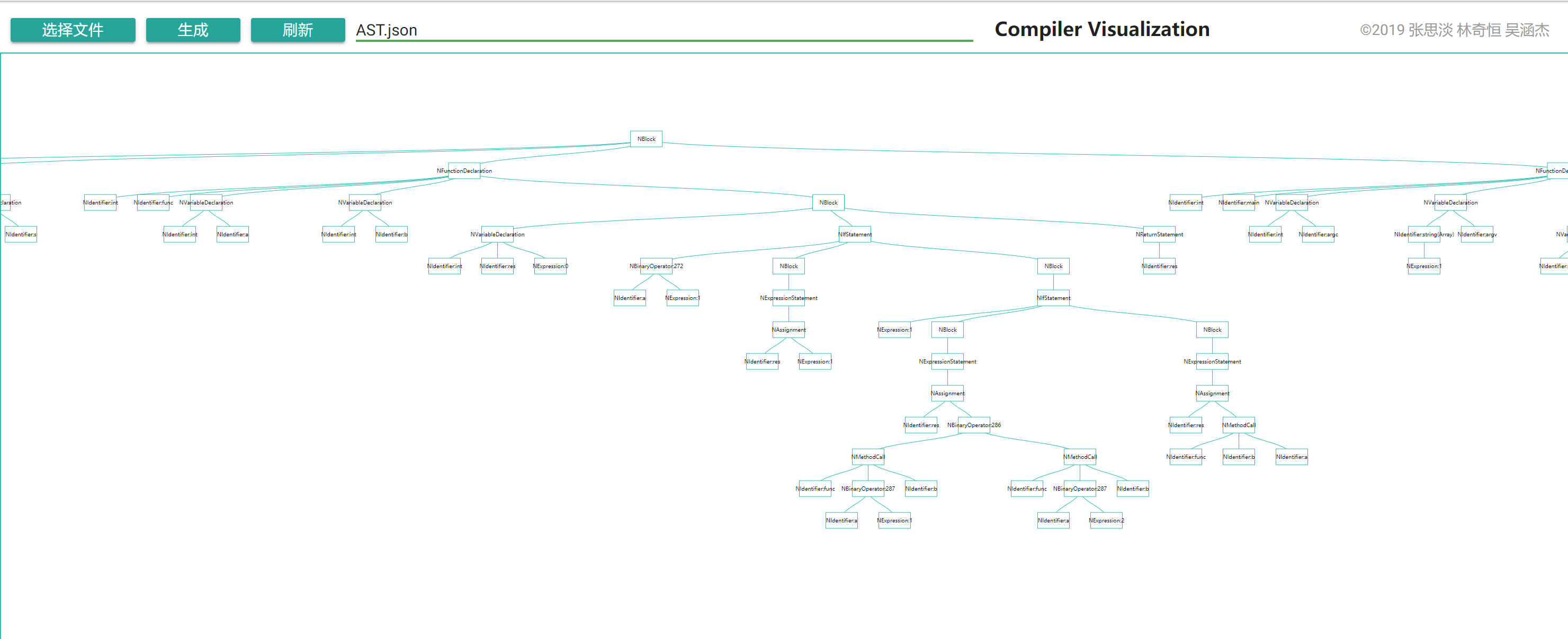
* 变量的声明、初始化（一维数组初始化）
* 函数声明，函数调用（传递任意支持的参数类型）
* 外部函数声明和调用
* 控制流语句if-else、for、while及任意层级的嵌套使用
* 单行注释（//）和（#）
* 二元运算符、赋值、函数参数、隐式类型转换
* 全局变量的使用

## 2.语法树可视化

我们把语法树变成JSON，然后通过JavaScript库d3进行可视化。

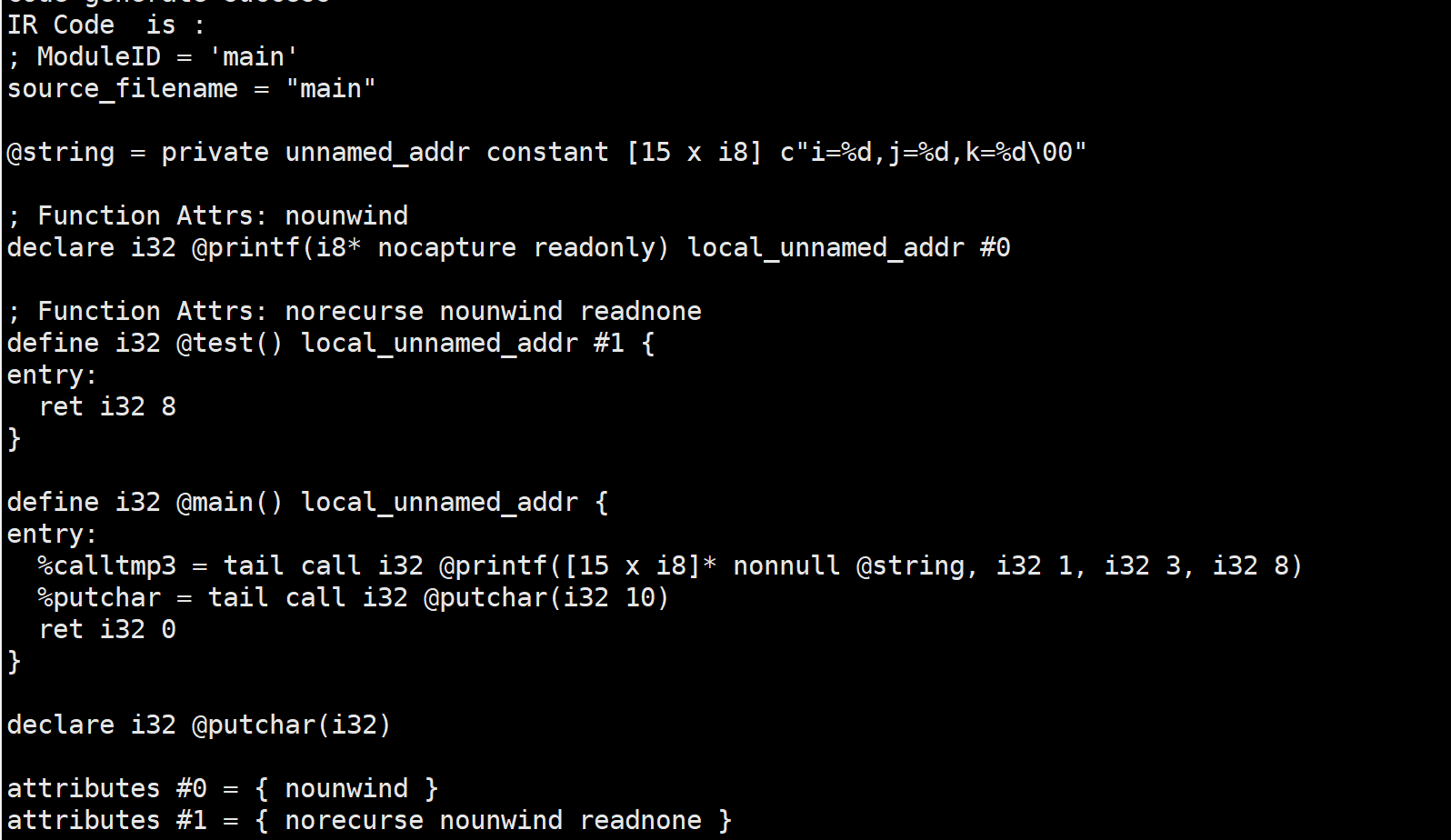
yacc分析完，生成语法树后，把语法树用JSON的格式存到了本地，文件名为AST.json。

可视化效果如下：



## 3.中间代码

因为我们使用了LLVM的API，所以中间代码是LLVM的bit code。



## 项目编译和测试：

以Debian/Ubuntu为例，需要安装的依赖有

build-essential llvm-3.9 clang 等

* 编译项目

flex -o token.cpp token.l

bison -d -o grammar.cpp grammar.y

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o grammar.o grammar.cpp

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o token.o token.cpp

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o CodeGen.o CodeGen.cpp

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o main.o main.cpp

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o ObjGenerate.o ObjGenerate.cpp

g++ -c `llvm-config-3.9 --cppflags` -std=c++11 -o TypeSystem.o TypeSystem.cpp

g++ `llvm-config-3.9 --cppflags` -std=c++11 -o Cutecompiler grammar.o token.o CodeGen.o main.o ObjGenerate.o TypeSystem.o `llvm-config-3.9 --libs` `llvm-config-3.9 --ldflags` -lpthread -ldl -lz -lncurses -rdynamic -L/usr/local/lib

* 使用编译成果二进制

cat tests/opt.myc | ./Cutecompiler

./test

## 4.目标代码

使用clang++对中间代码进行编译即可获得可以运行的二进制代码。

void syscall\_clang(string inputfilename, string outputfilename)

{

string clang = "clang++ -O3 -o";

string tab = " ";

string command = clang + tab + outputfile + tab + inputfile;

cout << "run command: " << command << endl;

system(command.c\_str());

}

斐波那契示例代码：

extern int printf(string format)

extern int puts(string s)

int func(int a, int b){

int res = 0

if( a <= 1 ) {

res = 1

}else if( 1 ){

res = func(a-1, b) + func(a-2, b)

}else{

res = func(b, a)

}

return res

}

int main(int argc, string[1] argv){

int i

argc = 10

for( i = 1 ; i<argc; i=i+1){

printf("i=%d, func=%d", i, func(i, argc))

puts("")

}

return 0

}

运行效果如下：



## 5.错误处理

我们使用C++的异常处理作为错误提示。

我们在main函数中增加了try…catch,,语句块，如果抛出异常，编译会终止。

我们把自己定义的异常都定义为std::logic\_error的类型。

比如说使用未定义函数：

Function \*calleeF = context.myModule->getFunction(this->id->name);

if (calleeF==nullptr)

{

throw std::logic\_error("Function name not found");

}

如果使用了未定义函数，我们将抛出异常，然后在main函数中抓取，并终止程序：

catch (const std::logic\_error &e)

{

cout << "logic\_error error catched!" << endl;

std::cout << e.what() << std::endl;

return 1;

}

示例代码如下：

//unknownFunction

int main(){

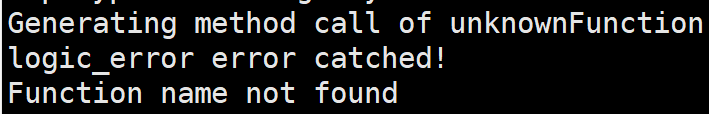
double i =0

unknownFunction()

return 0

}

运行结果：



我们还做了函数参数个数，类型转换，运算符的检查。

示例代码如下：

int main(){

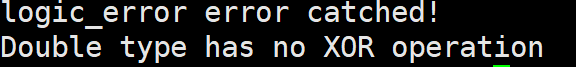
double i =0

i=i^1

return 0

}

运行结果：

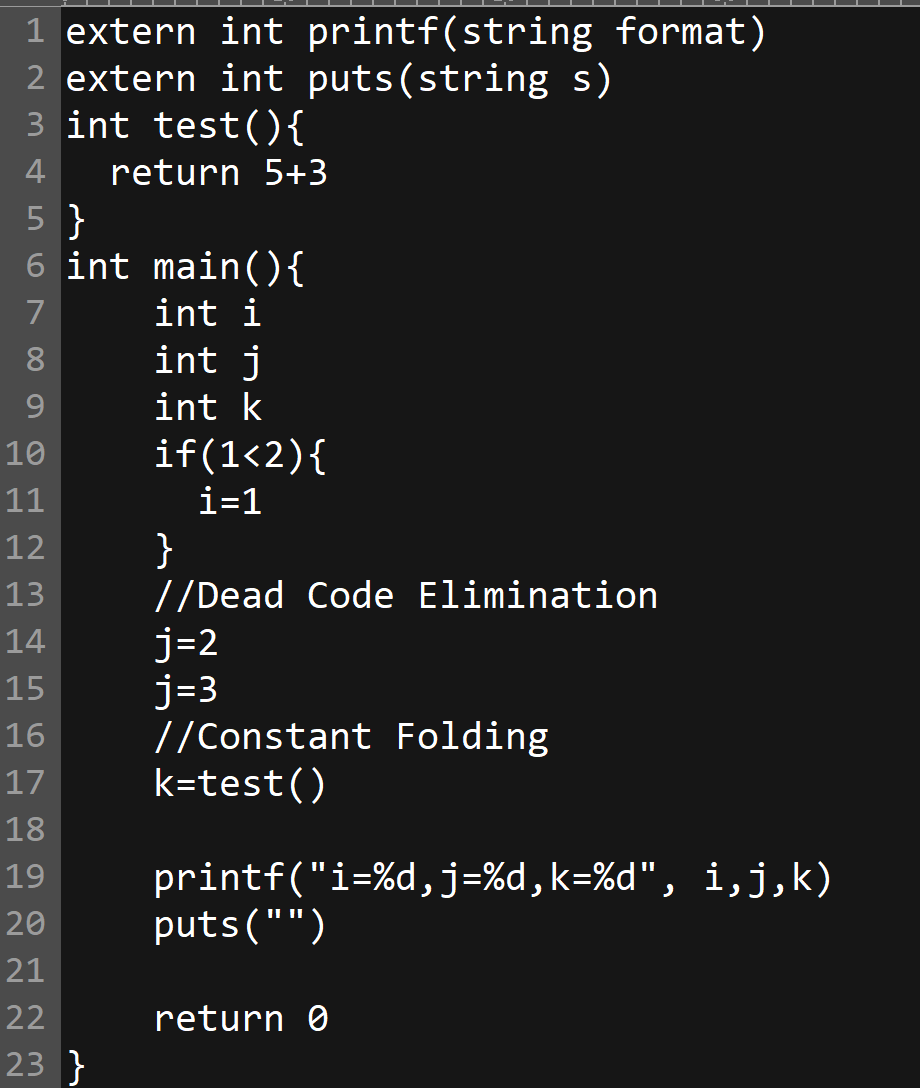


## 6.优化

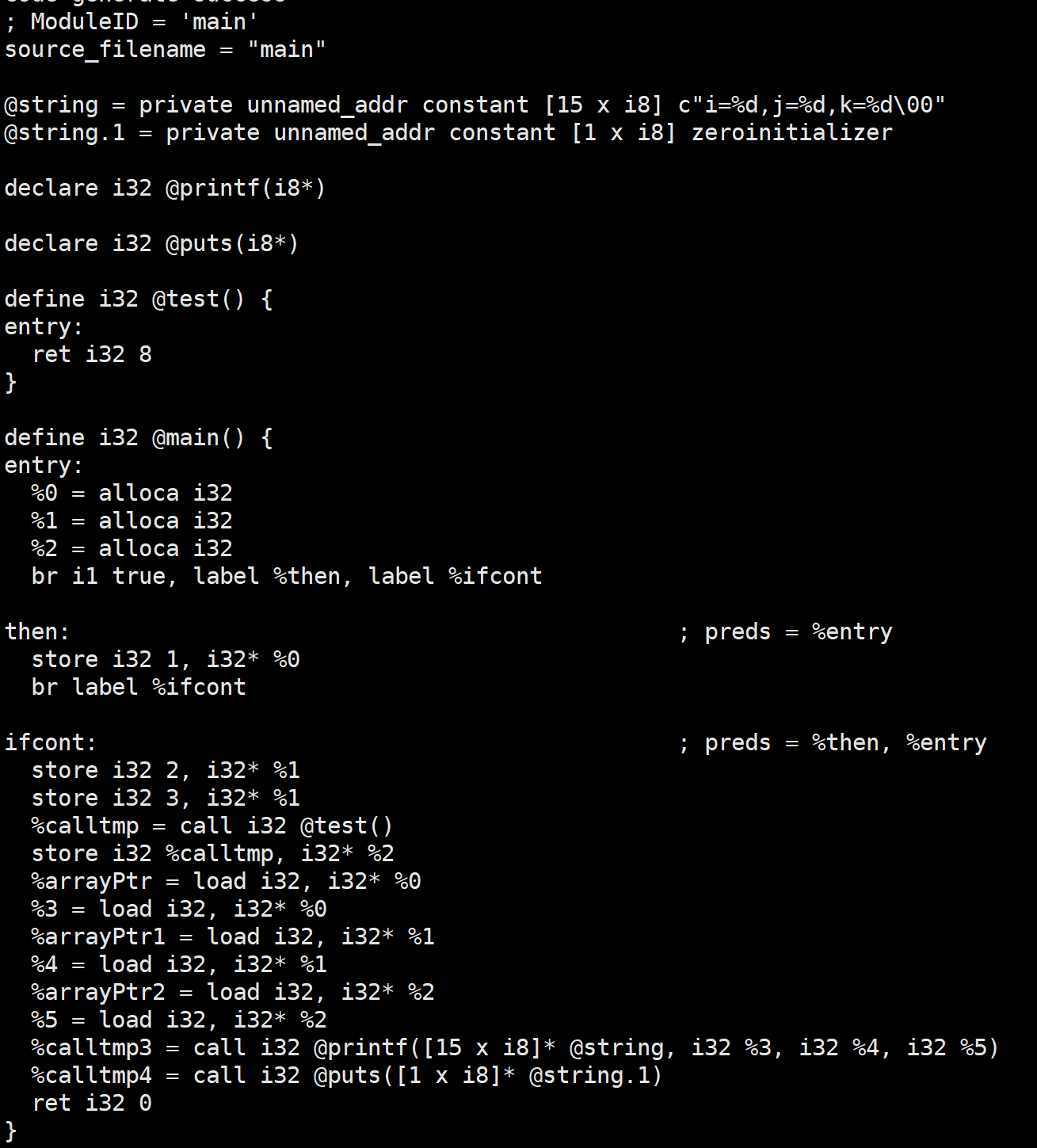
我们小组使用的优化是LLVM的API：llvm::legacy::PassManager，使用 PassManagerBuilder 进行构建。

我们启用了 SLP vectorizer 和 Loop Vectorizer， 并且做了常量折叠。

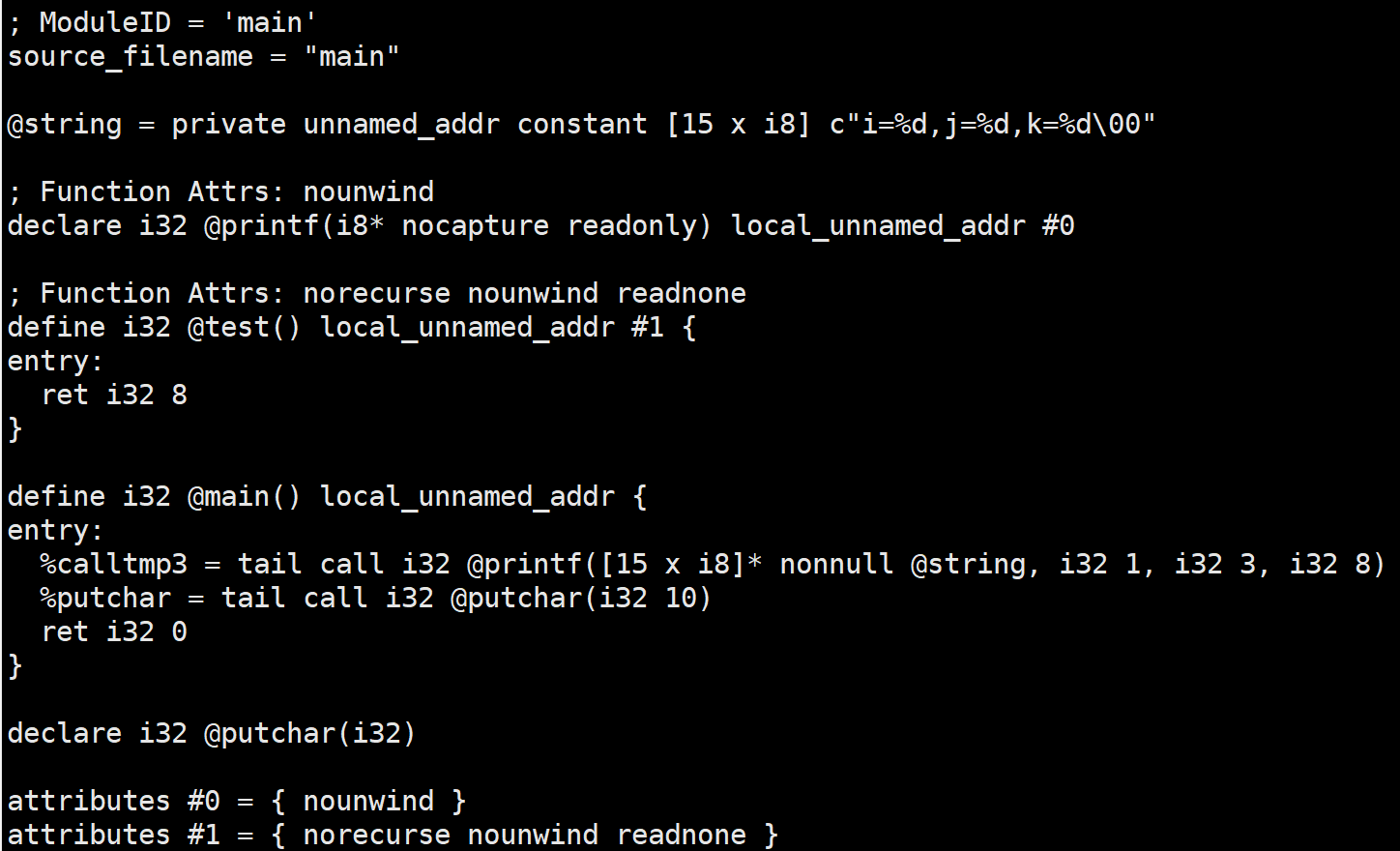
测试程序如下：



没进行优化的代码如下：



优化后的代码如下：



那些一定进去循环和判断都主动进行了运算和替换。