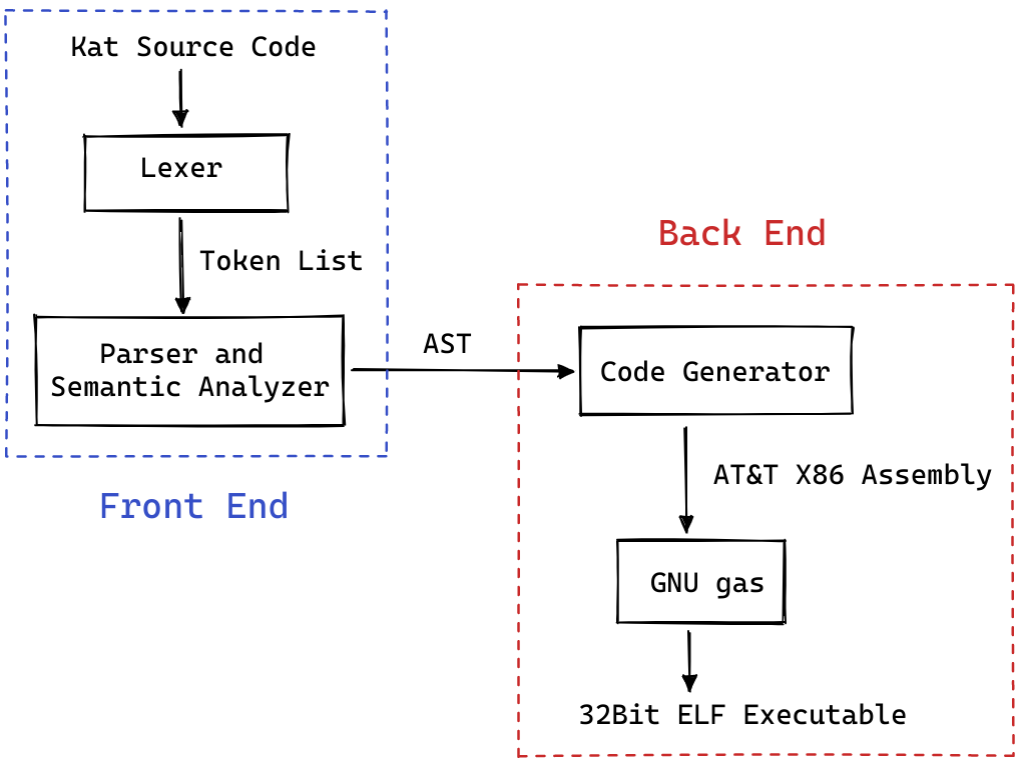
Kat: 一个编译器的实现

学号	姓名	专业	邮箱
19335109	李雪堃	计算机科学与技术 (超算)	i@xkun.me

Kat: 一个编译器的实现
- (一) 编译器设计
  - (1) 编译器架构
  - (2) 词法分析
  - (3) 语法分析
  - (4) 语义分析
  - (5) 代码生成和运行时
- (二) 示例程序和结果
  - (1) 词法分析结果
  - (2) 语法分析结果
  - (3) 错误处理
    - (3.1) 语法错误
    - (3.2) 语义错误
  - (4) 代码生成和程序运行
- (三) 总结和展望

(一) 编译器设计

(1) 编译器架构



Kat 编译器主要分为以下几个部分：

- Front End
  - Lexer：词法分析器，输入 kat 源代码，输出 token 序列。
  - Parser：语法分析器，输入 token 序列，输出抽象语法树 AST。
  - Semantic Analyzer：语义分析器，语义分析与语法分析同时进行，在我的设计中，语法树的节点有一个指向符号的指针，符号包含类型、大小等语义信息。
- Back End
  - Code Generator：代码生成器，输入 AST，遍历 AST 生成 AT&T x86 汇编指令。
  - Assembler And Linker：汇编器和链接器使用 GNU gas 和 GNU ld，从汇编代码生成最后的 ELF 文件。

下面，对每个模块的核心数据结构、采取的策略和方法，作适当的解释和说明。

(2) 词法分析

Token 的相关数据结构声明和定义在 `src/include/lex.h` 中。

Token 数据结构定义为 `token_t`，token 的类型由 `TK_TYPE` 枚举指定。Token 被分为关键字、标识符、数值、字符、字符串、标点符号 6 个类型。当然，需要额外标识 EOF token。

Token 中需要包含指向源代码 token 开始处的指针，以及 token 的长度，还有 token 的字面值 (对于数值、字符、字符串而言) 。此外，为了之后的错误处理，token 还需要包含所在源代码的行数，方便用户定位问题。

Token 序列被组织为一个单向链表，所以每个 token 还有指向下一个 token 的指针。

```
typedef enum TK_TYPE
{
  TK_KW,
  TK_ID,
  TK_NUM,
  TK_CHR,
  TK_STR,
  TK_PUNCT,
  TK_EOF,
} TK_TYPE;

typedef struct token_t
{
  TK_TYPE type;
  char *begin;
  size_t len;
  size_t line;

  union {
    int64_t ival;
    long double fval;
  };
  char cval;
  char *sval;

  struct token_t *next;
} token_t;
```

词法分析的实现在 `src/lex.c` 的 `lex()` 函数中，该函数以源代码字符流作为输入，输出 token 链表。

如果是构造 DFA 来实现词法分析 (that would be very stupid)，代码不好扩展和维护，这里使用**模拟 NFA + 字符串匹配**的方法来进行词法分析。

- 标识符和关键字的识别：id 和 keyword 实际上可以用同一种模式进行识别，只要维护一个 keyword 列表，首先统一识别为 id，然后与 keyword 进行匹配，如果有匹配则识别 token 为 keyword，这也是大多数编译器采取的做法。
- 标点符号的识别：标点符号不仅包含程序的分界符 (比如括号、分号等)，还包含运算符，这里采用的是最长匹配策略，即按照长度依次标点符号，比如说当前指向的字符是 `<`，先进行字符串匹配 `<=`，再匹配 `<`。

(3) 语法分析

Kat 语言语法的 **EBNF** 表示：

- 标识符
  - 标识符只能由字符、下划线、数字组成
  - 标识符必须以字母或下划线开头

```
identifier = (letter | underscore) , {letter | underscore | digit} ;
letter = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" ;
digit = "0" | "1" | ... | "9" ;
underscore = "_" ;
```

- 表达式

```
expression = primary {operator primary} ;
operator = "+" | "-" | "*" | "/" | "&&" | "||" | ">" | "<" | ">=" | "<=" | "==" |
"!=" ;
primary = ["+" | "-"] ("(" expression ")" | number | identifier | function-call) ;
```

- 函数
  - 函数定义由 `func` 关键字指定
  - 函数的每个参数指定为 `identifier: type` 的形式，参数列表可以为空
  - 函数的返回值指定为 `=> type` 的形式，返回值可以没有
  - kat 包含 `char`、`int`、`str`、`bool` 四种类型
  - 函数体是一个语句块，语句块由若干个语句组成，可以没有语句

```
function = "func" identifier "(" parameter-list ")" ["=>", type] block ;
parameter-list = [parameter {"," parameter}] ;
parameter = identifier ":" type ;
block = "{" {statement} "}" ;
type = "char" | "int" | "str" | "bool" ;
```

- 语句

  kat 包含 3 种类型的语句，语句块 `block` 由若干个 `statement` 组成

```
block = "{" {statement} "}" ;

statement =
[ declaration-statement
| expression-statement
| control-statement
] ;
```

- 声明语句
  - 变量声明由 `let` 关键字指定
  - 变量可以初始化或不初始化

```
declaration-statement = "let" identifier ":" type ["=" expression] ";" ;
```

- 表达式语句
  - 表达式语句由表达式组成，可以是赋值语句

```
expression-statement = [identifier "="] expression ";"
```

- 控制语句
  - `if-else` 语句
  - `while` 语句
  - `break` 语句
  - `continue` 语句
  - `return` 语句

```
control-statment =
[ "if" "(" expression ")" block      (* if-elif-else statement *)
  {"elif" block}
  ["else" block]
| "while" "(" expression ")" block   (* while statement *)
| "break" ";"                        (* break statement *)
| "continue" ";"                     (* continue statement *)
| "return" expression ";"            (* return statement *)
] ;
```

- 程序
  - 函数是 kat 的一等公民，一个 kat 程序由若干个函数组成

```
program = {function} ;
```

下面介绍语法分析阶段的核心数据结构和用到的分析方法。

抽象语法树 AST 的节点类型 `ND_TYPE` 和节点数据结构 `node_t` 的定义在 `src/include/parse.h` 中。

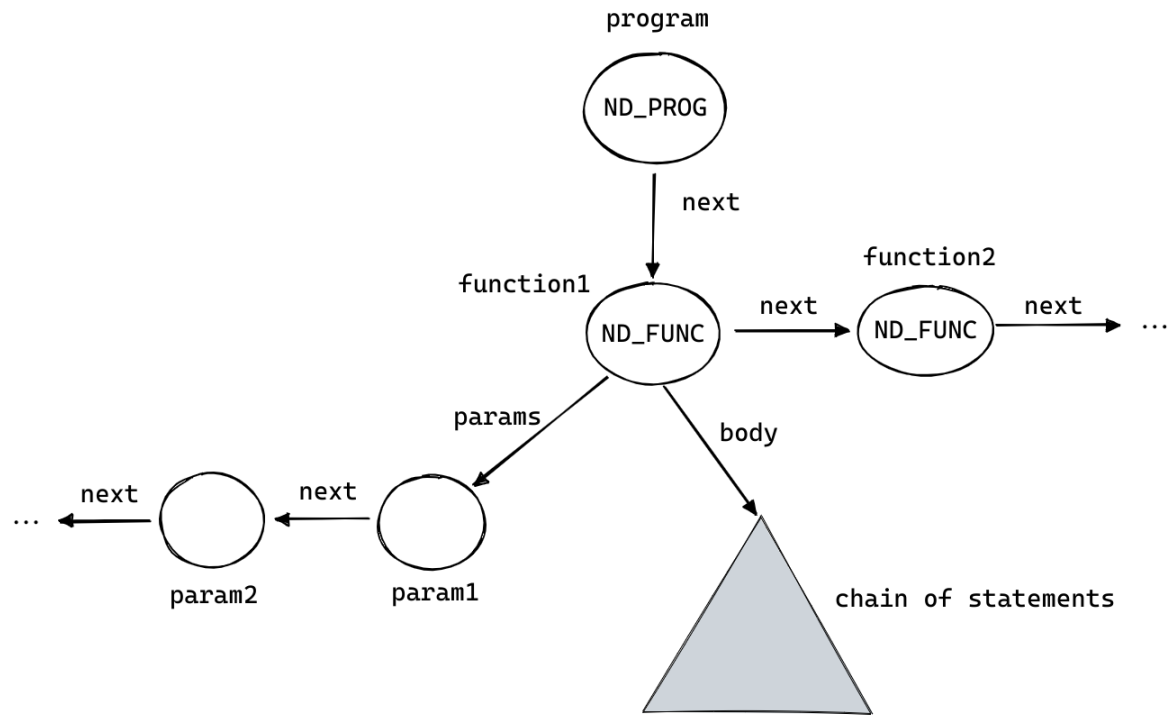
`node_t` 中包含指向符号 `symbol_t` 的指针，在语义分析的过程中可以记录在 AST 节点中，用于代码生成。

```
typedef enum ND_TYPE
{
  ND_NIL,       // do nothing
```

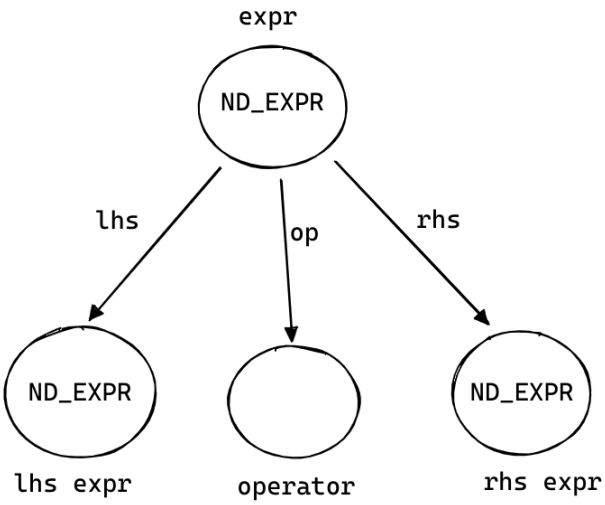
```
  ND_PROG,      // program
  ND_VAR,       // variable
  ND_FUNC,      // function
  ND_EXPR,      // expression
  ND_FNCALL,    // function call
  ND_BLOCK,     // statement block {..}
  ND_DECL_STMT, // declaration statement
  ND_EXPR_STMT, // expression statement
  ND_COND,      // condition
  ND_IF,        // if statement
  ND_WHILE,     // while statement
  ND_RETURN,    // return statement
  ND_NUM,       // number
  ND_LPAREN,    // "(" (used for opp, but will not appear in AST)
  ND_RPAREN,    // ")" (used for opp, but will not appear in AST)
  ND_ASSIGN,    // =
  ND_POS,       // + (unary, not implemented)
  ND_NEG,       // - (unary, not implemented)
  ND_ADD,       // + (binary)
  ND_SUB,       // - (binary)
  ND_MUL,       // *
  ND_DIV,       // /
  ND_LOGAND,    // &&
  ND_LOGOR,     // ||
  ND_EQ,        // ==
  ND_NE,        // !=
  ND_LT,        // <
  ND_LE,        // <=
  ND_GT,        // >
  ND_GE,        // >=
} ND_TYPE;

typedef struct node_t
{
  ND_TYPE type;

  // used when node type is ND_VAR or ND_FUNC
  union {
    symbol_t *var;
    symbol_t *func;
  };

  // function call or definition
  // the head of parameter list
  // parameters are organized to linked list
  struct node_t *params;

  // function body or statement block
  // the head of functions or statements
  // functions and statements are organized to linked list
  union {
    struct node_t *body;
    struct node_t *block;
  };

  // lhs, rhs and op
```

```
  // used by expression and condition
  // also used by declaration statement and expression statement
  struct node_t *lhs;
  struct node_t *op;
  struct node_t *rhs;

  // condition of if or while statement
  // used when node type is ND_IF or ND_WHILE
  struct node_t *cond;

  // if-else statement
  // used when node type is ND_IF
  struct node_t *if_stmt;
  struct node_t *else_stmt;

  // while statement
  // used when node type is ND_WHILE
  struct node_t *while_stmt;

  // the next node
  // used when node is function or statement
  struct node_t *next;

  // literals
  union {
    int64_t ival;
    long double fval;
  };
  char cval;
  char *sval;
} node_t;
```

对于一个 kat 程序，它基本的 AST 构造如下，函数组织为链表，对于每个函数，它的参数组织为链表，它的函数体是语句块，语句块是由单条语句连接成的链表。



对于一个 kat 表达式，它的 AST 构造如下，分为左运算数、运算符和右运算数，如果是嵌套多级表达式，lhs 和 rhs 可以一直递归地表示。



其余的 AST 构造就不多展示了，根据上面的注释和指针变量的名字可以很清楚知道各个指针指向的含义。

根据 kat 语法定义的特点，我们可以看出递归下降是比较好 parse 的，比如对于变量声明和函数声明，它们的开头分别用的是不同的关键字 `let` 和 `func`，不用 lookahead 就能区分它们，而且，对于不能用当前 token 区别的语法结构，我们可以向后 peek 一个甚至多个 token，这样写出的代码逻辑比较清楚，而且易于扩展和维护。

另外值得一提是，对于表达式的 parse，我们选择的是 OPP (运算符优先级分析)。经过稍微浏览 gcc 关于 C 语言 parse 部分的源代码以及 stackoverflow 上的一些问答，OPP 是绝大部分编译器选择 parse 表达式的方式，最重要的原因之一就是它可以 parse 出一个干净的语法树，而且对于语法定义的要求更低。

注意到 kat 表达式的 EBNF 为：

```
expression = primary {operator primary} ;
operator = "+" | "-" | "*" | "/" | "&&" | "||" | ">" | "<" | ">=" | "<=" | "==" | "!=" ;
primary = ["+" | "-"] ("(" expression ")" | number | identifier | function-call) ;
```

你可以发现，我们没有定义诸如 `term`、`factor` 之类的非终结符，这是因为如果采用多级的文法规则，使用递归下降时 parse 出的语法树会多出 `term`、`factor` 等不必要的节点，而且会使 AST 深度增加。实际上，我们最需要的只是运算符的优先级和结合性，而 OPP 能产生清晰、简洁的 AST。而通过修改文法的方式来 "适配" parse 我个人认为从根本上是错误的。

对于 OPP 的算法和过程，这里不赘述。表达式的 OPP 实现在 `src/parse.c` 中的 `parse_expr()` 函数，它使用了 `src/stack.c` 实现的栈来辅助构造 AST。

所以，我们语法分析选择的是递归下降 + 运算符优先分析的方法。语法分析模块的实现在 `src/parse.c` 中，提供的最终接口是 `parse()`，输入 token 序列，返回 AST。

(4) 语义分析

语义分析的主要代码分布在下面的文件中：

- `src/include/symbol.h` 和 `src/symbol.c`：类型、符号结构体的定义和相关函数
- `src/include/scope.h` 和 `src/scope.c`：符号表、作用域数据结构定义和相关函数
- `src/parse.c`：语义分析与 parse 同时进行，parse 过程中处理作用域、符号符号表构建、类型和类型检查等

类型和符号的数据结构在 `src/include/symbol.h` 中定义。`KAT_TYPE` 是 kat 基本数据类型，其中 `KAT_NIL` 是为了处理函数的返回值为空的情况，此时返回值类型标识为 `KAT_NIL`。`type_t` 中定义了类型的名称、大小等。由于当前没有支持 compound data type 的打算，所以暂时没有添加成员类型。

`symbol_t` 中包含指向 `type_t` 的多个指针，符号的名称，以及变量在栈上的 offset 等，目前没有支持全局变量的打算，所以没有添加全局地址的信息。目前`symbol_t` 只会是两种符号，栈上的局部变量，或者函数。

```
typedef enum KAT_TYPE
{
  KAT_INT,
  KAT_CHAR,
  KAT_STR,
  KAT_BOOL,
  KAT_NIL
} KAT_TYPE;

typedef struct type_t
{
  char *name;
  size_t size;
  KAT_TYPE kind;

  struct type_t *next;
} type_t;

typedef struct symbol_t
{
  char *name;
  bool is_var;
  bool is_func;

  // variable's type or function's return type
  union {
    type_t *type;
    type_t *return_type;
  };

  size_t params_num;
  type_t *params_type;

  int offset;

  token_t *token;

  struct symbol_t *next;
} symbol_t;
```

下面是符号表和作用域的核心数据结构。

对于符号表的数据结构，我选择的是绝大多数编译器同样采用的 hash 表，hashmap 的实现在 `src/include/hashmap.h` 和 `src/hashmap.c` 中。

符号表一般和作用域一起处理，我选择的是比较蠢的方式。由于暂时没有支持全局变量，所以对于变量而言，只需要处理块作用域，`scope_t` 中存储了指向符号表的指针，作用域组织为链表，每当进入一个嵌套的块作用域，就在链表头插入一个新的 scope。按照这种方式，对于符号的查找，只需要从作用域链表头开始一直遍历到结尾即可。比如，这样就可以依次上一级作用域查找符号定义了。变量作用域是 `var_scope`。

函数作用域是 `func_scope`，与变量作用域独立，相对于整个程序而言，同样组织为链表。

```
typedef struct scope_t
{
  hashmap_t *symbol_table;
  struct scope_t *next;
} scope_t;

extern scope_t *var_scope;
extern scope_t *func_scope;
```

符号的查找、添加，作用域的进入 `enter_scope()`、离开 `leave_scope()` 等函数在 `src/scope.c` 中。这里不赘述了。

(5) 代码生成和运行时

我选择生成的目标代码是 32 位 x86 汇编，因为我只对这个平台比较熟悉。生成的汇编是 AT&T 语法的 (because intel sucks)。代码生成模块的实现在 `src/include/codegen.h` 和 `src/codegen.c` 中。

我只实现了函数的代码生成 (指的是 `%ebp` 和 `%esp` 那些)，以及算术表达式、声明语句、表达式语句、`retuen` 语句的代码生成。限于本人能力和时间，没有完成布尔表达式、`if` 语句和 `while` 语句的生成。所以我的程序基本上只能做计算器。

另外值得一说的是，表达式生成我采取的是栈式计算，会产生巨多 `push` `pop` 指令，非常低效，唯一的好处就是不考虑寄存器的使用了，这是限于时间的无奈之举，捏麻的大作业太多了。算术表达式代码生成是 `codegen.c` 的 `gen_expr()` 函数。

`i386/i686` 平台的调用约定可用下图说明，调用函数前将参数依次压栈，然后调用函数。函数内先保存 `%ebp` (其中保存了旧的 `%esp` 的值，然后将现在的 `%esp` 复制给 `%ebp`)，然后分配栈空间 (将 `%esp` 寄存减去栈的大小)，最后恢复 `%esp` 和 `%ebp` 的内容。函数的返回值存放在 `%eax` 中。

```
HIGH ADDRESS
               program stack
             ==================
             | 2-nd parameter |  <- 12(%ebp)
             +=================+
             | 1-st parameter |  <- 8(%ebp)
             +=================+
             | return address |  <- 4(%ebp)
             +=================+
             | old %esp value |  <- (%ebp)
             +=================+
             | 1-st local vrb |  <- -4(%ebp)
             +=================+
 (%esp) ->   | 2-nd local vrb |  <- -8(%ebp)
             ==================
LOW ADDRESS
```

下面是一个通用的函数模板，中间部分生成函数体。生成函数的代码在 `src/codegen.c` 的 `gen_func()` 函数中。

```
.section .text
.global func
func:
  push %ebp           # push the old %ebp value on the stack
  movl %esp, %ebp     # copy the old %esp to %ebp

  subl $8, %esp       # reserve 8 bytes of memory space on the stack
                      # for local variable used within the function

  pushl %edi          # if this function must use these preserved registers
  pushl %esi          # then you must push them on the stack
  pushl %ebx

  # <function code>

  popl %ebx           # and pop them off when the function is ready to return to the
calling program
  popl %esi
  popl %edi

  movl %ebp, %esp     # recover old %esp
  popl %ebp           # recover old %ebp
  ret                 # return

  # the return value is stored in %eax
  # normally, %eax can hold a 4-byte integer value
  # for using 64-bit long integer values, the return value is placed in the edx:eax
register pair
```

前面提到，函数开始时，我们在保存栈指针后，需要分配栈空间，为此需要预先知道栈的大小，`codegen.c` 中的 `calculate_stack_size()` 函数遍历函数体中的每条语句，只有当遇到声明语句时才会增长栈，与此同时计算声明变量的 offset (相对于 `%ebp`，即函数调用前原 `%esp` 的值)，这样就可以通过 offset 获取局部变量了。

```
// calculate how many bytes needed by the local variables
// also set the offset off the stack for these variables
static size_t calculate_stack_size(node_t *node)
{
  // iterate through all the stmts in function body
  // stack only grows when delaration appears
  size_t stack_size = 0;
  node_t *stmt = node->body;
  while(stmt) {
    if (stmt->type == ND_DECL_STMT) {
      stack_size += stmt->lhs->var->type->size;
      stmt->lhs->var->offset = -stack_size;
    }
    stmt = stmt->next;
  }
  return stack_size;
}
```

另外，为了能够展示结果，我们需要一个运行时函数，能够输出一些东西。为此，我们将 libc 中的 `printf()` 进行了一层封装，生成一个 `print` 函数，专门用来输出一个 `int`。

```
static void gen_data()
{
  emit(".section .data");
  emit("msg:");
  emit("  .asciz \"hello, friends :^)\\n\"");
  emit("number_formatter:");
  emit("  .asciz \"%d\\n\"");
  emit("");
}

// print the integer stored in %eax
static void gen_runtime_print()
{
  emit(".type print, @function");
  emit(".globl print");
  emit("print:");
  emit("  pushl %%ebp");
  emit("  movl %%esp, %%ebp");
  emit("  pushl %%eax");
  emit("  pushl $number_formatter");
  emit("  call printf");
  emit("  add $8, %%esp");
  emit("  movl %%ebp, %%esp");
  emit("  popl %%ebp");
  emit("  ret");
  emit("");
}
```

我们最终生成的是汇编代码，所以还需要 assembler 和 linker 来生成 ELF，我选择的是 GNU gas 和 GNU ld。

在 `main.c` 中，有这么一行代码，它调用 `gcc`，指定 `-m32` 选项生成 32 位 ELF。多亏了 `x86_64` 架构引以为傲且惊人的向后兼容性，我们可以直接运行最后的可执行文件。

```
execl("/usr/bin/gcc", "gcc", "-m32", output_file_path, "-o", argv[2], (char *) NULL);
```

(二) 示例程序和结果

关于如何复现下面的结果，请查看 `readme.pdf`。

(1) 词法分析结果

词法分析的示例程序为 `test/lex/lex.kat`。`lex.kat` 覆盖了基本的词法单元结构，关键字、类型、标识符、数字等。

Dump token list 的实现在 `src/lex.c` 中的 `dump_token_list()` 函数

```
func add(a: int, b: int) => int {
  return a + b;
}

func main(argc: int, argv: str) => int {
  let x: int = (1 + 2) * 3;
  let y: int = 4 * (5 - 6);
  let res: int = add(x, y) + 10;

  let z: bool = (x == 9);
  let c: char;
  if (z) {
    c = 'Y';
  } else {
    c = 'N';
  }

  let i: int = 0;
  let sum: int = 0;
  while (i <= 10) {
    sum = sum + i;
  }

  return 0;
}
```

程序输出的 token list 在 `test/lex/tokens.txt` 中。由于实在太长这里就不贴了。

输出的 token 格式如下：

```
{<token-type>: token-name at line N}
```

- `token-type` 是 token 类型，有 `keyword`、`identifier`、`punctuator`、`number`、`character`、`string` 和 `eof`
- `token-name` 是 token 对应的字符串
- `N` 是 token 所在源代码的行数

(2) 语法分析结果

语法分析的示例程序为 `test/parse/parse.kat`。覆盖了基本的语法结构，比如函数、`if` 语句、`while` 语句、声明语句、表达式语句等。

Dump AST 的相关实现在 `src/parse.c` 中，是最后面的一些以 `dump` 开头的函数。目前我只实现了算术表达式的 parse，类型也只支持 `int`。

```
func add(a: int, b: int) => int {
  return a + b;
}

func main(argc: int, argv: str) => int {
  let x: int = (1 + 2) * 3;
  let y: int = 4 * (5 - 6);
  let res: int = add(x, y) + 10;

  let flag: int;
  if (x >= 9) {
    flag = 1;
  } else {
    flag = 0;
  }

  let i: int = 0;
  let sum: int = 0;
  while (i <= 10) {
    sum = sum + i;
  }

  return 0;
}
```

程序输出的 AST 在 `test/parse/ast.txt` 中。下面对 AST 的子树分别进行说明。

`add` 函数对应的 AST 如下。对于函数，会输出函数名、参数列表和返回值，然后输出函数体，含义非常明显。缩进表示在 AST 中的深度。

```
Function: add (a: int, b: int) => int
    FuncBody:
        ReturnStmt:
            Expr:
                Variable: a int
                Add: +
                Variable: b int
```

`main` 函数中前 3 行对应的 AST 如下。可以看出，OPP 可以生成干净、简洁的 AST，表达式结构非常清晰，而且运算优先级和结合性 (目前都是左结合) 正确。其中的声明语句有 `(initialized)` 的标识，表明变量声明时有被初始化。其中的函数调用 `add(x, y)`，输出函数名和实参列表。

```
Function: main (argc: int, argv: str) => int
    FuncBody:
        DeclStmt (initialized):
            Variable: x int
```

```
            Assign: =
            Expr:
                Expr:
                    Num: 1
                    Add: +
                    Num: 2
                Multiply: *
                Num: 3
        DeclStmt (initialized):
            Variable: y int
            Assign: =
            Expr:
                Num: 4
                Multiply: *
                Expr:
                    Num: 5
                    Subtract: -
                    Num: 6
        DeclStmt (initialized):
            Variable: res int
            Assign: =
            Expr:
                FunCall: add
                    param1:
                        Variable: x int
                    param2:
                        Variable: y int
                Add: +
                Num: 10
```

`if` 语句部分的 AST 如下 (缩进承接上面)。可以看到，对于声明时未初始化的变量 `flag`，`DeclStmt` 后标识了 `(unintialized)`。目前只实现了 `if-else` 语句的 parse，未支持 `elif`。`Condition` 是 `if` 的条件，`IfBlock` 是条件为真时执行的语句，`ElseBlock` 是条件为假时执行的语句，含义非常明显。`ExprStmt` 是表达式语句。

```
        DeclStmt (uninitialized):
            Variable: flag int
        IfElseStmt:
            Condition:
                Expr:
                    Variable: x int
                    GreaterThanOrEqualTo: >=
                    Num: 9
            IfBlock:
                ExprStmt:
                    Variable: flag int
                    Assign: =
                    Num: 1
            ElseBlock:
                ExprStmt:
                    Variable: flag int
                    Assign: =
                    Num: 0
```

`while` 语句部分的 AST 如下 (缩进承接上面)。`Condition` 和 `WhileBlock` 是 `WhileStmt` 的子节点，含义非常明显。可以看到 parse 的结果完全正确。

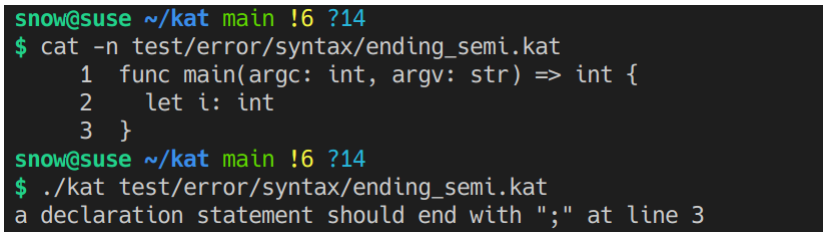
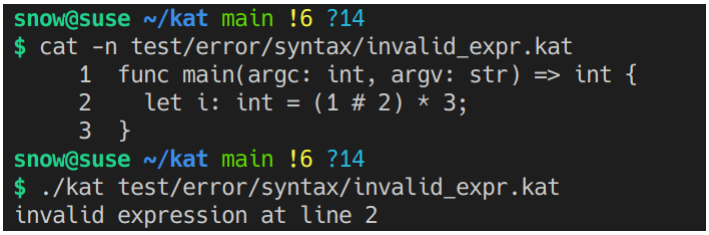
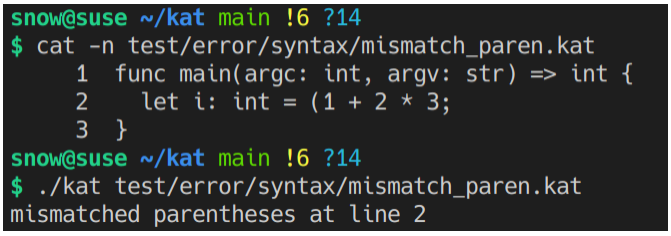
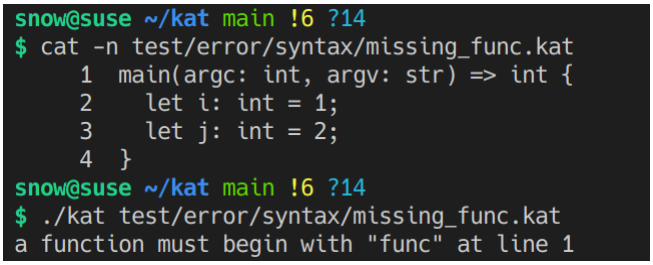
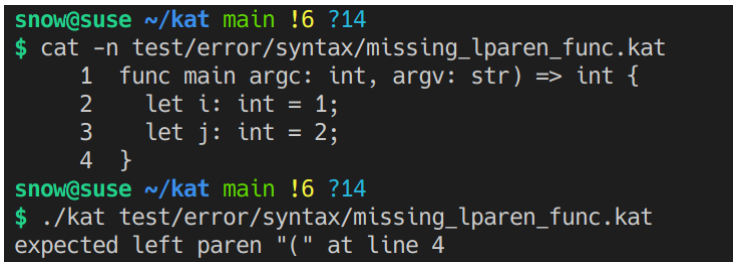
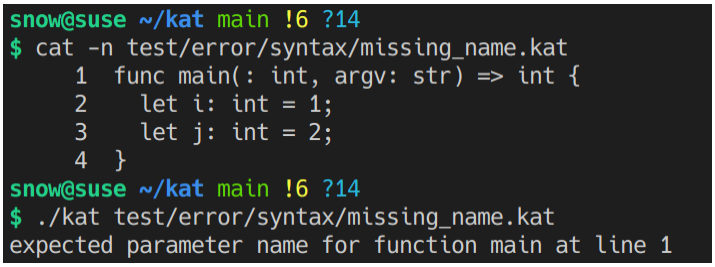
```
        DeclStmt (initialized):
            Variable: i int
            Assign: =
            Num: 0
        DeclStmt (initialized):
            Variable: sum int
            Assign: =
            Num: 0
        WhileStmt:
            Condition:
                Expr:
                    Variable: i int
                    LessThanOrEqualTo: <=
                    Num: 10
            WhileBlock:
                ExprStmt:
                    Variable: sum int
                    Assign: =
                    Expr:
                        Variable: sum int
                        Add: +
                        Variable: i int
```

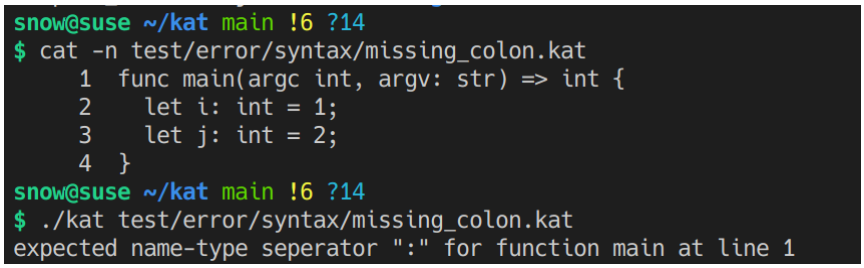
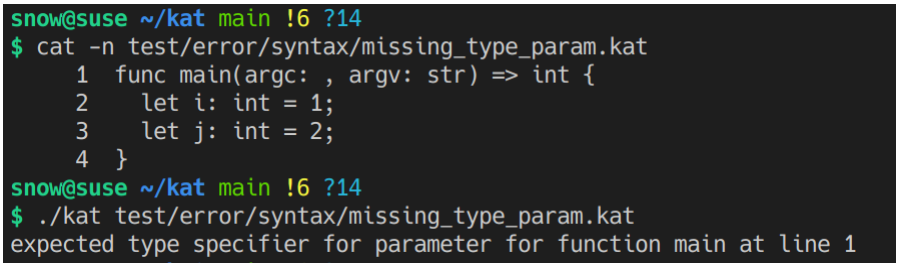
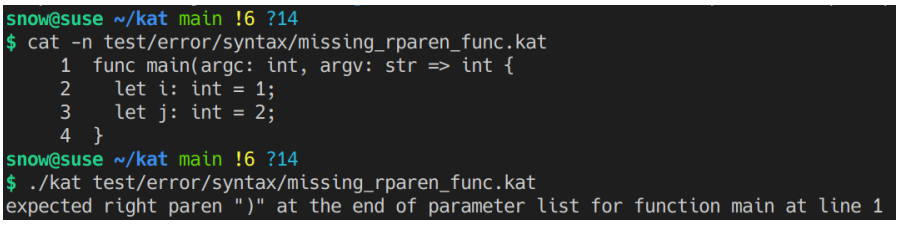
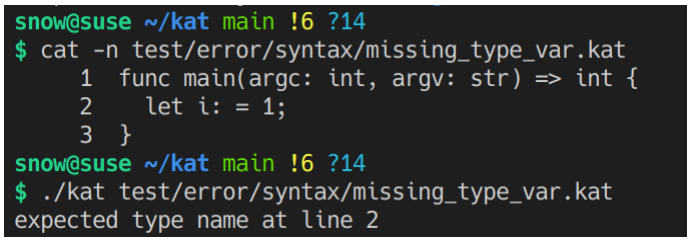
(3) 错误处理

kat 拥有较为完善的错误处理机制，kat 可以识别绝大部分语法错误，以及大部分常见的语义错误，kat 的错误处理采用 panic 模式。错误处理部分的示例程序位于 `test/error` 中。

(3.1) 语法错误

下面展示的语法错误示例程序都在 `test/error/syntax` 中。

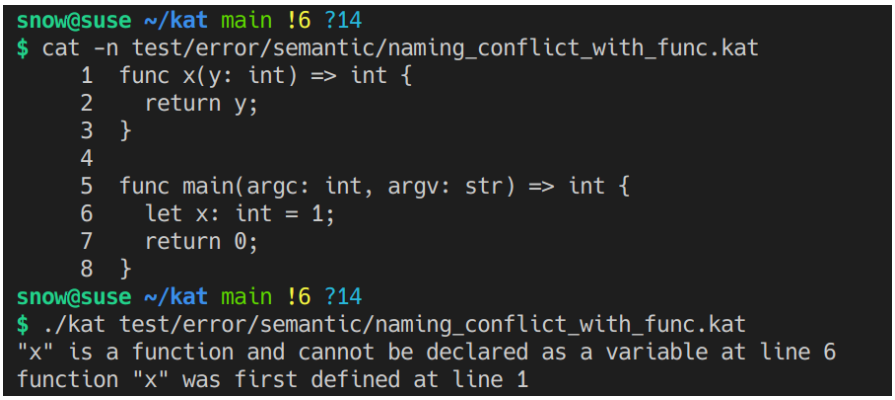
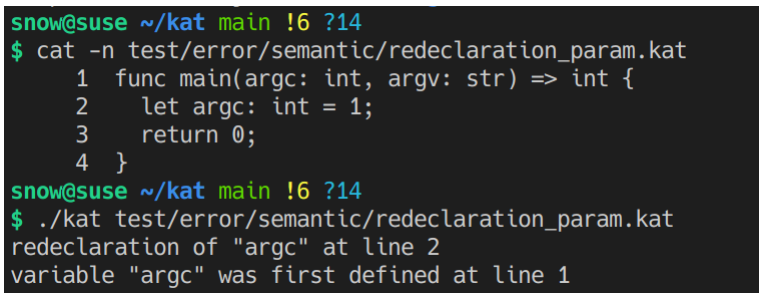
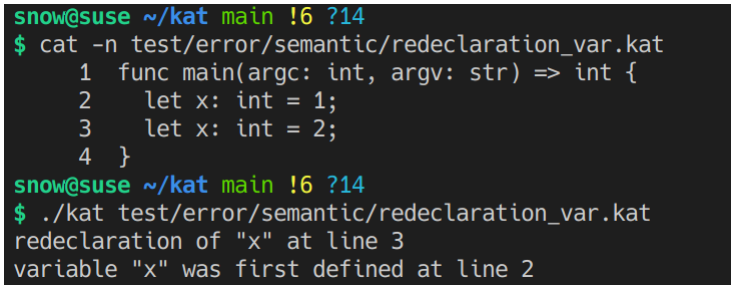
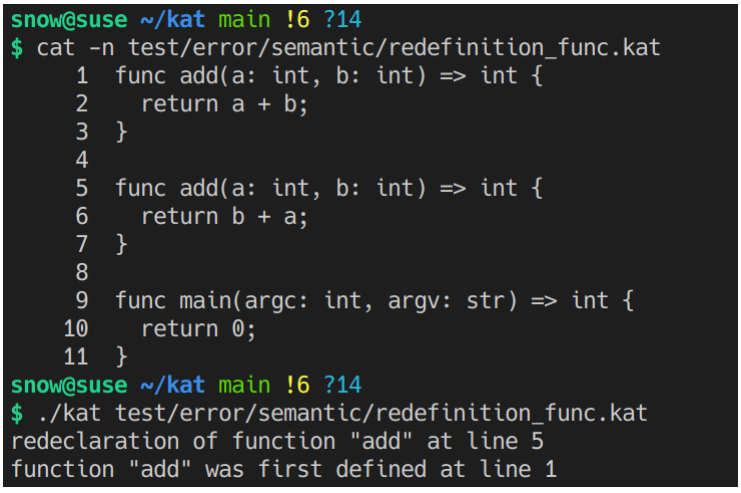
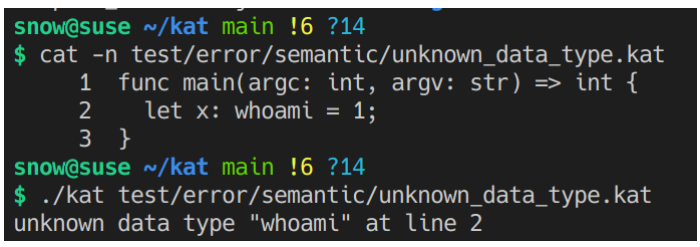
运行结果	错误描述
	语句末尾没有 `;`
	非法表达式
	表达式括号不匹配
	函数定义前缺失 `func` 关键字
	函数参数列表缺失左括号 `(`
	函数参数名缺失

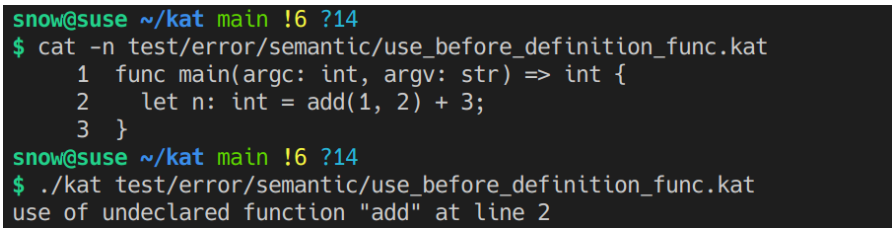
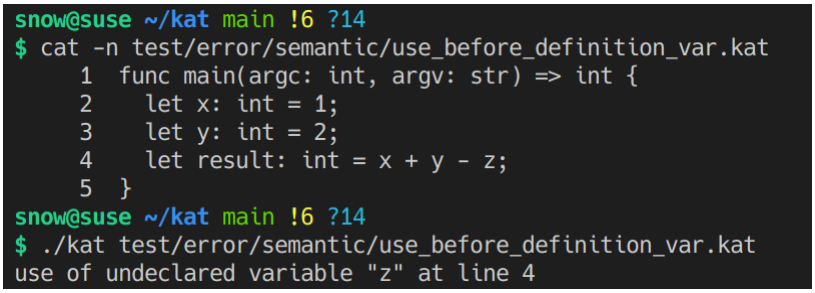
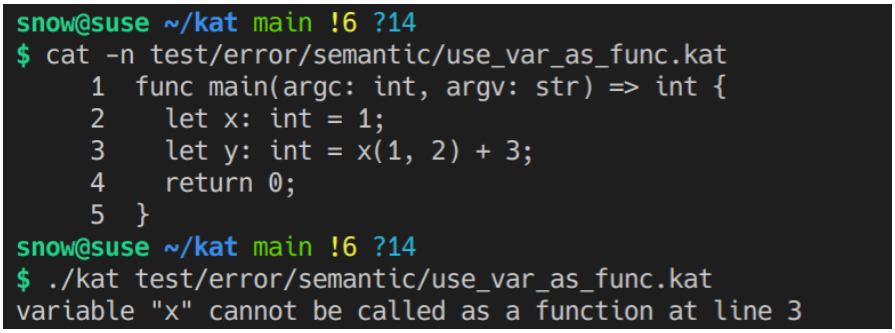
运行结果	错误描述
	函数参数列表缺失 `:` 分隔符
	函数参数列表缺失类型
	函数参数列表缺失右括号 `)`
	声明语句缺失类型

还有部分语法错误没有展示，比如声明语句缺失 `:`、语句块缺失 `{` 等，限于篇幅不展示了。

(3.2) 语义错误

下面展示的语义错误示例程序都在 `test/error/semantic` 中。

运行结果	错误描述
	声明先前定义函数同名的变量
	函数内声明与参数同名的变量
	同级作用域内变量重复声明
	函数重复定义
	变量声明未知数据类型

运行结果	错误描述
	使用未定义的函数
	使用未定义的变量
	将变量作为函数使用

(4) 代码生成和程序运行

示例程序在 `test/hello.kat`，作用是表达式的计算和输出。

在前面设计部分说过，我们在代码生成阶段会生成一个运行时 `print` 函数专门用来输出单个 `int`。所以，我们先需要定义这个函数，以通过语义检查，函数体为空，因为会自动生成。

```
func print(a: int) {}

func main(argc: int, argv: str) => int {
  let a: int = (1 + 2);
  let b: int = (3 - 4) * 5;
  let c: int = a * b;
  print(c);
}
```

上面的程序期望输出 `-15`，可以看到运行结果是正确的。第一行输出的 `hello, friends :^)` 是我注入的后门，目前每个 kat 程序运行时第一行都会输出这条信息。

```
snow@suse ~/kat main !6 ?11
$ make clean && make
[kat] clean
[kat] compiling codegen.c => codegen.o
[kat] compiling hashmap.c => hashmap.o
[kat] compiling lex.c => lex.o
[kat] compiling main.c => main.o
[kat] compiling parse.c => parse.o
[kat] compiling scope.c => scope.o
[kat] compiling stack.c => stack.o
[kat] compiling symbol.c => symbol.o
[kat] linking kat
[kat] build done
snow@suse ~/kat main !6 ?11
$ ./kat test/hello.kat test/hello
snow@suse ~/kat main !6 ?11
$ test/hello
hello, friends :^)
-15
```

汇编代码和可执行文件都会生成在相同目录下，且名字相同 (文件扩展名不同 `.s`)。下面是 `hello.kat` 生成的汇编 `hello.s`。虽然 work，但是生成了极其低效的算术表达式汇编指令，可以说跟 shit 一样。

```
.section .data
msg:
  .asciz "hello, friends :^)\n"
number_formatter:
  .asciz "%d\n"

.section .text
.type print, @function
.globl print
print:
  pushl %ebp
  movl %esp, %ebp
  pushl %eax
  pushl $number_formatter
  call printf
  add $8, %esp
  movl %ebp, %esp
  popl %ebp
  ret

.type main, @function
.globl main
main:
  pushl %ebp
  movl %esp, %ebp
  subl $12, %esp
  push $msg
  call printf
  add $4, %esp
  pushl $1
  pushl $2
  popl %edi
```

```
popl %eax
addl %edi, %eax
pushl %eax
popl %eax
movl %eax, -4(%ebp)
pushl $3
pushl $4
popl %edi
popl %eax
subl %edi, %eax

pushl %eax
pushl $5
popl %edi
popl %eax
imul %edi, %eax
pushl %eax
popl %eax
movl %eax, -8(%ebp)
pushl -4(%ebp)
pushl -8(%ebp)
popl %edi
popl %eax
imul %edi, %eax
pushl %eax
popl %eax
movl %eax, -12(%ebp)
pushl -12(%ebp)
call print
movl %ebp, %esp
popl %ebp
ret
```

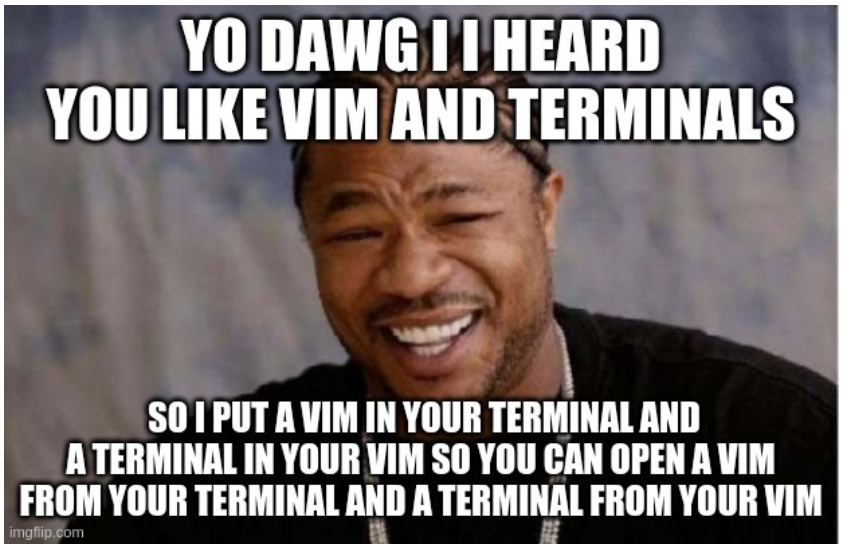
(三) 总结和展望

写编译器太好玩了，我非常享受与计算机、CPU 对话的过程，虽然这次实现有太多不足，主要遗憾在代码生成部分，由于最近多个作业缠身，所以只能挤出半天时间完成，做的非常仓促，还请助教见谅。其他模块相对完善得多。

BUT，暑假就要到了，我觉得这个项目很好玩，有值得继续的必要，下面整几个 FLAG：

- 移植 kat 到 krix 上 (我的玩具操作系统)
- 代码生成完善，控制语句的支持、寄存器分配和指令选取
- 整理这些 nasty 的代码 (可能需要更 nasty 的 macro ...)

OK，就到这了，最后整个表情包。


YO DAWG I I HEARD
YOU LIKE VIM AND TERMINALS
SO I PUT A VIM IN YOUR TERMINAL AND
A TERMINAL IN YOUR VIM SO YOU CAN OPEN A VIM
FROM YOUR TERMINAL AND A TERMINAL FROM YOUR VIM
imgflip.com