華東師範大學
EAST CHINA NORMAL UNIVERSITY

编译实践技术

实验报告

学生姓名	李嘉睿，陈可欣，孔露萍
学　　号	10175101250，10175101179，10175101180
专业班级	2018 级编译原理与技术
指导教师	琚小明
学　　院	软件工程学院
完成时间	2021 年 1 月 2 日

華東師範大學
EAST CHINA NORMAL UNIVERSITY

目录

一、 背景与现状分析

现如今，复杂而泛在的软件架构支撑着全球经济，编译器和高级语言正是这些软件的基石。IBM 第一款优化编译器发布至今已 60 年了。这正是对编译器发展现状进行评估，对未来发展方向进行探讨的一个契机。

20 世纪 50 年代编译器领域刚刚起步，其研究焦点局限于从高级语言到机器码的转换以及优化程序对时间和空间的需求。此后，该领域产生了大量的有关程序分析与转换、代码自动生成以及运行时服务等方面的新知识。同时，编译算法也被用于便利软件和硬件开发、提高应用程序性能、检测或避免软件缺陷和恶意软件等方面。编译领域与其它方向越来越多地相互交叉渗透，这些方向包括计算机体系结构、程序设计语言、形式化方法以及计算机安全等。到目前为止，编译器领域最为突出的成就是高级语言的广泛使用。从银行、企业的管理软件，到高性能计算和各种万维网应用，今天的绝大多数软件都是用高级语言编写并经过编译的。

众多的编译算法，包括词法分析、类型检查和推导、数据流分析、基于数据依赖性分析的循环变换以及软件流水等，都是计算机科学中的奇迹。通过集成到各种功能强大而应用广泛的工具中，这些方法极大地影响着计算领域的实践。与早期的编译器实现相比，今天的编译算法明显变得越来越复杂。早期的编译器采用简单直观的技术对程序进行词法分析，而今天的词法分析技术则基于形式语言和自动机理论，这使得编译器前端的开发更为系统化。

二、 相关技术介绍

本项目并未使用词法生成器或者语法生成器。因此本节主要概述自行设计实现的以文法作为输入，LL 分析表以及同步集合作为输出的 llgen 程序的设计与实现。

（一） llgen 概述

llgen 用于生成供 LL(1) parser 使用的 parse table 以及错误恢复时使用的同步集合。parse 表最终以二维常量整型数组表示，表每行表示：*non-terminal 符号在 terminal 符号作为 lookahead 符号的情况下，对应的操作。操作分为两类：

1. 推导出的 production 的索引。
2. 推导错误，用 0 表示。

同步集合最终以二维常量整型数组表示，表每行表示当前 non-terminal 符号因为 lookahead 符号无法继续推导进入 panic 模式时可供选择的下一个 terminal 符号集合。

（二） 输入格式

输入文件被用于生成 LL(1)算法的 parse 表。该文件由三个段构成：

1. **terminal 段：**该段用于定义所有 terminal，第一行应当为`[terminal]`。之后各行为各个 terminal 符号，两个 terminal 之间用空白符（若没有特殊说明，本文空白符均包括‘ ’， ‘\t’，‘\n’，'\r'）分隔。
2. **non-terminal 段：**该段用于定义所有的 non-terminal，第一行应当为`[nonterminal]`。两个 non-terminal 之间用空白符分隔。
3. **production 段：**该段用于定义所有的 production，第一行应当为`[production]`。之后每行表示 production。head 和 product 之间使用`->`以及一个或多个空白符分隔。两个不同 head 的 production 之间用`$`分隔。相同 head 的 production 之间使用`|`以及一个或多个空白符分隔。

   例如，一个支持四则运算的语法分析规范输入文件如下：

```
[terminal]
+ - * \
( )
num name
[nonterminal]
goal
expr
term
factor
[production]
goal -> expr $
expr -> term expr' $
expr' -> + term expr' | - term expr' | epsilon $
term -> term * factor | term / factor | factor $
term' -> * factor term' | / factor term' | epsilon $
factor -> ( expr ) | num | name $
```

（三） 数据结构

1. 索引定义

llgen 程序内部需要维护三种索引分别如下：

terminal 索引：和 token 的 enum 类型中的值相同，因此要求 terminal 段的输入和 enum 类型**顺序相同**。同时要求按 enum 中定义的顺序作为输入。

non-terminal 索引：在 terminal 基础上依据输入顺序。

production 索引：按输入顺序依次递增

2. 索引映射的维护

动态字符串数组：维护索引到 terminal 和 non-terminal 的映射

哈希表：维护 terminal 和 non-terminal 字符串到索引的映射

3. production 的表示

使用动态数组稀疏表表示。每一个 nonterminal 对应一个动态数组，用于存储同一 head 对应的所有 production。单个 production 也是用动态数组表示。因此属于三维数组结构

4. first，follow，first+集合的表示

使用动态数组表示所有集合，索引和 nonterminal 索引一一对应。每一项用一个 hashset 表示。

（四） 算法

- 计算 first 集合

```
for each terminal i {
    first[i] = i
}
for each non-terminal i {
    first[i] = empty
}
while (first sets still change) {
     for each productions {
         for each production {
             first[head] += first[product[0]] - epsilon
             for i = 1; i < len(product) - 1 && epsilon in first[i-1];
i++ {
                 first[head] += first[product[i]] - epsilon
             }
```

```
            if i == len(product) - 1 and epsilon in first[product[i]]
{
                first[head] += epsilon
            }
            first[head] += first
        }
    }
}
```

- 计算 follow 集合

```
for each non-terminal i {
    follow[i] = empty
}
follow(S) = {eof}
while (follow sets 变化) {
    for each production {
        RECORD = follow[head]
        for i = len(product) - 1 to 0 {
            follow[product[i]] += RECORD
            if (epsilon in first[product[i]]) {
                RECORD += first[product[i]] - epsilon
            } else {
                RECORD = first[product[i]]
            }
        }
    }
}
```

- 生成 parse table

```
#define ERROR -1
for each non-terminal i {
    for each terminal j except for epsilon {
        if j in fisrt+[i] {
            table[i][j] = i
        }
    }
}

for other item in talbe {
    item = ERROR
}
```

三、 功能需求分析

（一） 词法

需求：词法部分以 toy 文件源码作为输入，按需生成 token。

（二） 语法

需求：词法部分以词法部分生成的 token 作为输入，最终生成语法树。

（三） 语义

需求：语义分析部分主要检查变量是否定义。同时构造符号表。

（四） 代码生成

需求：代码生成部分以语法树作为输入，首先生成结构更加简化的抽象语法树，进而在抽象语法树基础上生成线性中间表示——三地址码。

（五） 出错处理

需求：编译错误应当分成词法、语法、语义三类。

（六） 基本输入输出显示

需求：支持交互式和文件输入两种基本输入输出。

（七） 图形界面输入输出显示

需求：应当支持图形界面的源码编辑、编译、运行以及中间表示显示操作，同时应当提供文件保存、新建、打开等功能。

（八） 代码执行

需求：应当能够执行三地址代码，并最终打印各个变量的值作为执行结果。

四、 功能设计

（一）词法

本小结主要介绍词法分析器 scanner 的设计。

1. 数据结构

token 结构的定义如下：

```
typedef struct {
    char *start;
    int length;
    TokenType type;
    int line_num;
    int line_pos;
} Token;
enum TokenType {
    // keywords
    INT, REAL, IF, THEN, ELSE, WHILE,
    // literal
    IDENTIFIER, NUM_INT, NUM_DOUBLE,
    // delimeters
    LEFT_PAREN, RIGHT_PAREN, // "(", ")"
    LEFT_BRACE, RIGHT_BRACE, // "{", "}"
    COMMA, // ","
    DOT, // "."
    SEMICOLON, // ";"
```

```
    // operators
    PLUS, MINUS, // "+", "-"
    STAR, SLASH, // "*", "/"
    // one or two character tokens
    EQUAL, EQUAL_EQUAL, // "=", "=="
    LESS, LESS_EQUAL, // "<", "<="
    GREATER, GREATER_EQUAL, // ">", ">="
    BANG_EQUAL // "!="
}
```

2. 按照自动机 DFA 来实现词法分析

词法分析时首先跳过空白字符并判断是否到达文件结尾。接着跳过注释。然后获取 lookahead 符号并判断是否是数字或 ID。接着根据 lookahead 符号使用 switch 判断 token 类型。最后若没有一项匹配成功的则报告错误并重新匹配。

具体地使用 DFA 进行词法分析如下：

```
/**
 * Return a token each time being called or an error token with message.
 */
T scan_token() {
    // TODO(ljr): complete scanner
    char ch;
    scanner.start = scanner.cur;
    if (is_at_end()) {
        return make_token(T_EOF);
    }
```

```
skip_whitespace();
if (is_at_end()) {
    // eof after whitespace
    return make_token(T_EOF);
}

// ch = advance();
ch = peek();

// comment
while (ch == '/' && next() == '/') {
    ch = comment();
    if (is_at_end()) {
        // eof after comment
        return make_token(T_EOF);
    }
    skip_whitespace();
    ch = peek();
    if (is_at_end()) {
        // eof after whitespace
        return make_token(T_EOF);
    }
}
// number
if (is_digit(ch)) {
    return number();
```

```
}

    // identifier
    if (is_alpha(ch)) {
        return identifier();
    }

    switch (ch) {
    // delimiters
    case '(': advance(); return make_token(T_LEFT_PAREN);
    case ')': advance(); return make_token(T_RIGHT_PAREN);
    case '{': advance(); return make_token(T_LEFT_BRACE);
    case '}': advance(); return make_token(T_RIGHT_BRACE);
    case ',': advance(); return make_token(T_COMMA);
    case ';': advance(); return make_token(T_SEMICOLON);
    // operators
    case '+': advance(); return make_token(T_PLUS);
    case '-': advance(); return make_token(T_MINUS);
    case '*': advance(); return make_token(T_STAR);
    case '/': advance(); return make_token(T_SLASH);
    // one or two character tokens
    case '=': return match('=') ? make_token(T_EQUAL_EQUAL) :
make_token(T_EQUAL);
    case '<': return match('=') ? make_token(T_LESS_EQUAL) :
make_token(T_LESS);
    case '>': return match('=') ? make_token(T_GREATER_EQUAL) :
make_token(T_GREATER);
```

```
    case '!': if (match('=')) return make_token(T_BANG_EQUAL);
    default:
        break;
}

    // advance();
    return error_token();
}
```

3. 扫描设计

scanner 本身状态维护如下，包含了当前扫描的 token 的起始位置、当前扫描到的位置、行号和行位置：

```
typedef struct {
    const char *start; // current beginning of the lexme
    const char *cur; // current character
    int line_num;
    int line_pos;
} Scanner;
```

4. 关键字与标识符 ID 的分离

对于关键字和标识符 ID，最初在判断首字符为字母后，同时落入`identifier()`的 token 生成函数中如下：

```
In scan_token():
// identifier
```

```
    if (is_alpha(ch)) {
        return identifier();
}
In identifier():
static inline T identifier(void) {
    assert(is_alpha(peek()));
    while (is_digit(peek()) || is_alpha(peek())) advance();
    return make_token(get_identifier_type());
}
```

在获取紧接着的字母和数字后，使用 switch 根据结果判断是 ID 还是关键字，并生成对应 token 如下：

```
static TT get_identifier_type(void) {
    // (int, if), real, then, else, while

    switch (*scanner.start) {
        case 'r': return check_keyword(1, 3, "eal", T_REAL);
        case 't': return check_keyword(1, 3, "hen", T_THEN);
        case 'e': return check_keyword(1, 3, "lse", T_ELSE);
        case 'w': return check_keyword(1, 4, "hile", T_WHILE);
        case 'i':
            if (scanner.start[1] == 'n') {
                return check_keyword(1, 2, "nt", T_INT);
            } else {
                return check_keyword(1, 1, "f", T_IF);
            }
        default: break;
    }
```

```
    return T_IDENTIFIER;
}
```

5. 词法返回值

采取按需扫描策略，导出如下接口

```
Token scan_token();
```

在语法分析 parser 需要时可直接调用获取 token。由于 scanner 的设计隐藏了错误的 token，因此每次 parser 调用`scan_token()`时获得的都是有效的 token 返回值。

（二）语法

本小节主要介绍 LL（1）语法分析器 parser 的设计。

1. 数据结构

语法树节点结构如下：

```
// 节点类型
typedef enum {AST_Stmt, AST_Expr, AST_Term} NodeType;
// stmt 节点类型
typedef enum {
    STMT_PROGRAM
    STMT_Stmt, STMT_Compound,
    STMT_Stmts, STMT_If,
    STMT_While, STMT_Assign
} StmtType;
```

```
// expr 节点类型
typedef enum {
    EXPR_Bool = STMT_Assign + 1,
    EXPR_BoolOp,
    EXPR_Arith,
    EXPR_ArithPrime,
    EXPR_Mult,
    EXPR_MultPrime,
    EXPR_Simple
} ExprType;
// ast 节点结构
struct {
    NodeType type;
    union {
        StmtType stmt;
        ExprType expr;
    } NonTerminalType;
    union {
        Token *token; // terminal
        char *name; // non-terminal
    } attr;
    DArray ptrs; // arr for children
} AstNode
```

2. 语法分析表存储

使用 llgen 程序生成的语法分析表及同步集合为 int 型常量二维数组，即最终存储在可执行文件的静态区。如下：

```
// ll(1) parser table
const int LL_PARSE_TABLE[14][29] = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 2, 0, 0, 3, 4, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 7, 0, 0, 7, 7, 0, 0, 0, 7, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 8},
    {0, 0, 0, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 12, 12, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 17,
13, 15, 14, 16, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 18, 18, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
```

```
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 21, 0, 0, 0, 0, 21, 19, 20, 0, 0, 0,
21, 21, 21, 21, 21, 0, 0, 21},
    {0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25, 0, 0, 0, 0, 25, 25, 25, 23, 24,
0, 25, 25, 25, 25, 25, 0, 0, 25},
    {0, 0, 0, 0, 0, 0, 0, 26, 27, 28, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0}
};
// snychronized set
const int SYNCHRONIZED_SET[14][29] = {
{11 ,27},
{3 ,5 ,7 ,6 ,11 ,12},
{3 ,7 ,6 ,5 ,11 ,12 ,27},
{3 ,6 ,7 ,11 ,12 ,28},
{3 ,7 ,6 ,5 ,11 ,12},
{3 ,7 ,5 ,6 ,11 ,12},
{3 ,6 ,5 ,7 ,11 ,12},
{7 ,10 ,8 ,9},
{7 ,8 ,9 ,21 ,23 ,22 ,25 ,24},
{7 ,10 ,9 ,8 ,15 ,22 ,23 ,21 ,24 ,25},
{10 ,15 ,17 ,16 ,22 ,23 ,21 ,24 ,25 ,28},
{7 ,10 ,8 ,9 ,15 ,16 ,17 ,22 ,23 ,21 ,24 ,25},
{10 ,15 ,16 ,17 ,19 ,18 ,22 ,23 ,21 ,24 ,25 ,28},
{7 ,10 ,9 ,8 ,15 ,18 ,19 ,16 ,17 ,22 ,23 ,21 ,24 ,25},
};
```

3. 分析堆栈构建与初始化

伪代码表示的算法如下：

```
// 基于表驱动的算法实现生成语法树
word <- NextWord() // lookahead 的 token
push eof onto Stack
push the start symbol, S onto Stack
focus <- top of Stack // 当前需要 match 的位置
loop {
    if focus == eof and word == eof:
        success and break
    else if focus in T or focus = eof:
        if focus matches word:
            pop Stack
            word <- NextWord()
        else
            report error
    else:
        if Table[focus, word] is A->B1B2...Bk:
            pop Stack
            for i <- k to 1:
                if Bi not epsilon
                    push Bi onto Stack
        else
            report error
    focus <- top of Stack
}
```

4. 输出语法树构建

在通过分析堆栈分析的同时进行语法树的构建，具体方法如下：使用一个额外的语法树栈 s 记录语法树信息：

1. 每一个 non-terminal 展开时，构造一个数组节点入栈
2. 每个数组节点维护一个"当前处理到第 i 个节点的索引"
3. 当 i 表明该数组节点各个子节点均处理完成，则该节点 pop 出栈，同时更新此时栈内 top 节点的信息

最终得到的最后一个 pop 出的节点即语法树的根节点。

5. 采用 LL(1)算法的语法分析结构

最终生成的语法树以 program 为根节点的多儿子树形结构，例如对

```
{ a = 1 + 1; }
```

生成的语法树如下：

```
program
  compoundstmt
    {
    stmts
      stmt
        assgstmt
          a
          =
          arithexpr
            multexpr
              simpleexpr
```

```
                1
              multexprprime
                epsilon
            arithexprprime
              +
              multexpr
                simpleexpr
                  1
                multexprprime
                  epsilon
              arithexprprime
                epsilon
          ;
      stmts
        epsilon
  }
```

（三）语义

1. 在语法树基础上对类型进行语义分析同时构建符号表

遍历语法树，根据遇到的 non-terminal 的 token 类型进行分类讨论：

1. 当第一次遇到某个变量的`assgstmt`即赋值语句时，将第一个子节点的 ID token 对应的变量插入符号表。
2. 当遇到`simpleexpr`且第一个子节点为`ID`时查表判断该变量是否存在，若未查到则输出表示语义错误的信息，反之继续遍历。

（四）代码生成

本小节主要介绍代码生成的实现。语义分析完成之后，保留下来的语法树存在很多冗余节点，采取 SDT 语法制导翻译首先转换成 ast 树。进而考虑再生成三地址码。语法树 -> ast 树 -> 三地址码。

1. 数据结构

三地址码数据结构如下：

```
#define MAX_LINE 1024

typedef enum {
    OPERAND_VAR, OPERAND_CONST,
    OPERAND_TEMP, OPERAND_LABEL
} OperandKind;

typedef struct {
    enum { VARIABLE, CONSTANT } kind;
    union {
        Token token;
        int tempvar_index; // 表示中间变量索引
        int label_index; // 表示 label 索引
    } u;
} *Operand;

typedef struct {
```

```
19.     enum { ADD, SUB, MUL, DIV, ASSIGN, EQUAL, LESS, LESS_E, GREATE,
    GREATE_E } op;
20.     Operand *arg1;
21.     Operand *arg2;
22.     Operand *result;
23.} InterCode;
24.
25. InterCode *codes[MAX_LINE];
```

2. 语法树生成抽象语法树

抽象语法树复用了语法树的节点表示形式，只是在自顶向下遍历过程中生成去除了冗余节点的另一棵树。具体地，采取以下策略：

1. 对于 A->epsilon 的节点均删除，子节点为空的中间节点也删除
2. 对于除`{`和`}`外的 terminal，直接生成节点

最终生成的抽象语法树应当仅包含下列类型节点：

1. non-terminal 类型：stmts，ifstmt，whilestmt，assgnstmt 四种。
2. terminal 类型：operator 操作符（分为二元和 bool 类型两类），operand（根据 token 类型可分为 ID 和 NUM 两种）。

具体地生成过程的语法指导翻译模式如下：

1. 对于语句类型的语法树子树：

- 对于 program，返回 compoundstmt 结果
- 对于 stmt，返回子节点结果
- 对于 compoundstmt，返回 stmts 结果
- 对于 stmts，循环构造节点包含所有 stmt 结果
- 对于 ifstmt，构造节点返回 boolexpr，stmt1，stmt2 结果

- 对于 whilestmt，构造节点返回 boolexpr，stmt 结果
- 对于 assgstmt，构造节点返回 ID 和 arithexpr 结果

2. 对于表达式类型的语法树子树：

- 对于 boolexpr，以 boolop 为根，左右作为子树
- 对于 boolop，返回和当前相同的节点
- 对于 arithexpr 和 multexpr，
  - 生成左子树
  - 生成右子树
  - 若右子树非空，则以右子树根为根
- 对于 arithexprime 和 multexprprime
  - 以第一个子节点为根
  - 如果 arithexprprime 非 epsilon，则 arithexprprime 的返回值作为右子树，multexpr 作为右子树的左节点
  - 如果 arithexprprime 为 epsilon，则 multexpr 返回值作为右子树
- 对于 simpleexpr
  - 若为叶子，返回相同节点
  - 若为(arithexpr), 返回 arithexpr

3. 抽象语法树生成三地址代码

自顶向下遍历抽象语法树时翻译生成，同时传入上层提供的变量名。具体翻译模式如下：

1. 语句的翻译

a) ifstmt

```
label1 = new_label()
label2 = new_label()
label3 = new_label()
```

```
[code1, cmp] = translate_expr_cond(stmt.1, label1, label2)
code2 = translate_Stmt(Stmt1, sym_table)
code3 = translate_Stmt(Stmt2, sym_table)
return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
```

b) whilestmt

```
label1 = new_label()
label2 = new_label()
label3 = new_label()
code1 = translate_cond(stmt.0, label2, label3)
code2 = translate_stmt(stmt.1)
return [LABEL label1] + code1 + [LABEL label2] + code2 + [goto label1] + [LABEL label3]
```

c) assignstmt

```
if T_[operation] {
   t1 = new_temp()
   code1 = translate_expr_arith(stmt.1, t1)
   code2 = [stmt.0 := t1]
   code = code1 + code 2
} else {
   code = [stmt.0 := stmt.1 ]
}
return code
```

d) stmts

遍历每一个节点翻译即可

2. 表达式的翻译

a） expr_arith(expr, place)

```
if (expr.left is expr) {
    t1 = new_temp()
    code1 = translate_expr_arith(expr.left, t1)
} else {
    t1 = expr.left
}
if (expr.right is expr) {
    t2 = new_temp()
    code2 = translate_expr_arith(expr.right, t2)
} else {
    t2 = expr.right
}
code3 = [place = t1 + t2]
return    code1 + code2 + code3
```

b） expr_cond(expr, label_true, label_false)

```
if (expr.left is expr) {
    t1 = new_temp()
    code1 = translate_expr_arith(expr.left, t1)
} else {
    t1 = expr.left
}
if (expr.right is expr) {
```

```
    t2 = new_temp()
    code2 = translate_expr_arith(expr.right, t2)
  } else {
    t2 = expr.right
  }
  code3 = [if t1 expr.op t2 goto label_true]
  return code1 + code2 + code3 + [goto label_false]
```

（五）符号表

表小节介绍符号表的设计。符号表在语义分析阶段构造。采用哈希表的数据结构。

1. 符号表的结构

```
typedef struct {
    int cnt;
    int bucket_size;
    bool (*cmp) (const void *a, const void *b);
    unsigned (*hash) (const void *key);
    unsigned timestamp;
    struct t_entry {
        struct t_entry *next;
        const void *key;
        void *value;
    } **buckets;
} Table;
```

2. 符号表的查询、添加、修改等操作

符号表的哈希函数选用 FNV-1a 哈希函数以减少哈希碰撞的机会，具体如下：

```
unsigned hash_str(const void *key) {
    unsigned hash = 2166136261u;
    char *p = key;

    for (; *p != '\0'; p++) {
        hash ^= *p;
        hash *= 16777619;
    }
    return hash;
}
```

符号表支持插入、查询、销毁操作。均通过自行实现的哈希表相关函数封装而成。

（六）出错处理

本小节介绍错误处理的设计。

1. 出错信息

若代码存在错误，本项目编译生成的错误信息格式如下：

```
Error type: LEX_ERROR at (Line [行号], Pos: [位置]): [说明文字]
```

2. 出错定位

本项目主要将编译器生成的错误信息按阶段分成三类：词法错误 LEX_ERROR，语法错误 SYNTAX_ERROR，语义错误 SEMANTIC_ERROR。其定义如下：

1. **词法错误**：出现未定义字符以及不符合词法单元定义的字符。
2. **语法错误**：主要分成两类语法错误。1.分析栈栈顶是 terminal 符号，但是与当前 lookahead 符号不同，判定为缺少该符号。2.分析栈栈顶是 non-terminal 符号 A，当前的 lookahead 符号是 a，但是 LL_PARSE_TABLE[A][a]为 0，即无法继续推导，判定为匹配失败。
3. **语义错误**：由于无类型系统，因此使用为初始化的变量，即符号表中不存在的变量。

3. 错误处理

下面分别讨论词法、语法、语义的错误处理机制：

1. **词法**：当程序出现词法错误时，采取的方式是，跳过当前错误部分，返回下一个正常的 token。对于 parser 来说，其调用`scan_token()`接受到的均是正常的 token，因此相当于对 parser 隐藏了词法错误信息。

   例如，对于如下程序：

   ```
   {
       $$$$$
       x = 3;
   }
   ```

   parser 仍然能够正常生成语法树。

2. **语法**：和出错定位中相同，语法错误又分成两类分别进行 panic mode 处理。
   a) **栈顶是终结符号且和当前 token 不匹配**：输出错误信息“Missing [栈顶的终结符号]”，同时从分析栈中弹出该符号，并继续分析。

b) **栈顶是非终结符号 A，且当前 token 是 a。而 LL_PARSE_TABLE[A][a]为空：**根据同步集合不断忽略 token，直到能够同步。注意：若当前 lookahead token 能够表示一个语句的结束，如`T_SEMICOLON`，`T_RIGHTBRACE`则认为该语句结束，因此 pop stack，继续分析，而非忽略 token。

3. **语义**：由于 toy 语言不存在类型，因此语义分析只需要检查变量是否定义。在语义分析阶段，对每一个作为右值的变量查询符号表，若符号表中不存在，则输出变量未定义的错误信息。

（七）虚拟机执行

本小节主要介绍三地址码执行虚拟机的设计。

1. 虚拟机架构

虚拟机架构基于以下两点规则：

1. 使用一个 pc 变量表示当前遍历到的三地址码数组的索引。
2. 不考虑寄存器，所有变量均放在内存中，从内存中获取。

2. 数据结构

```
3. typedef struct {
4.     int pc;
5.     DArray *code;
6.     DArray *var;
7.     DArray *temp_var;
8. } vm;
```

3. 执行过程

- 预处理

  遍历三地址码数组，记录 label 声明对应的 pc。初始化 var、tempvar 数组。

- 指令处理

  四则运算指令：分别处理两个操作数，然后完成赋值

  if 条件指令：处理条件表达式，根据结果跳转或者继续执行

  goto 指令：设置 pc 的值

- 操作数处理

  OPERAND_VAR: 对于变量，插入 var 数组中，模拟对内存操作。

  OPERAND_TEMP：对于临时变量，插入 temp_var 数组中，模拟对内存操作。

  OPERAND_CONST: 对于 const 变量，直接计算出值

  OPERAND_LABEL: 对于 label 的跳转，直接设置 pc

- 结果打印

  按变量出现的顺序打印最终变量结果。

五、 功能实现

本节主要介绍 UI 的设计与实现， 使用 Qt 进行界面设计。

（一）基本的输入、输出界面

提供功能如下：

文件操作功能中包含打开文件、保存文件、新建文件以及退出文件。

帮助功能中可显示相关信息以及打开文档。

按键功能有编译程序、运行程序以及重置功能。

页面转换功能可以在编辑页面和中间结果页面之间进行切换。

编辑界面：

中间结果界面：

（二）界面的布局与设计

编辑界面：

主要可通过“文件”→“打开”选择相关 toy 文件输入到编辑界面，然后通过按钮进行编译和运行，查看编译结果和运行结果，还可对该输入框进行重置，清空输入框内容。

编译功能：

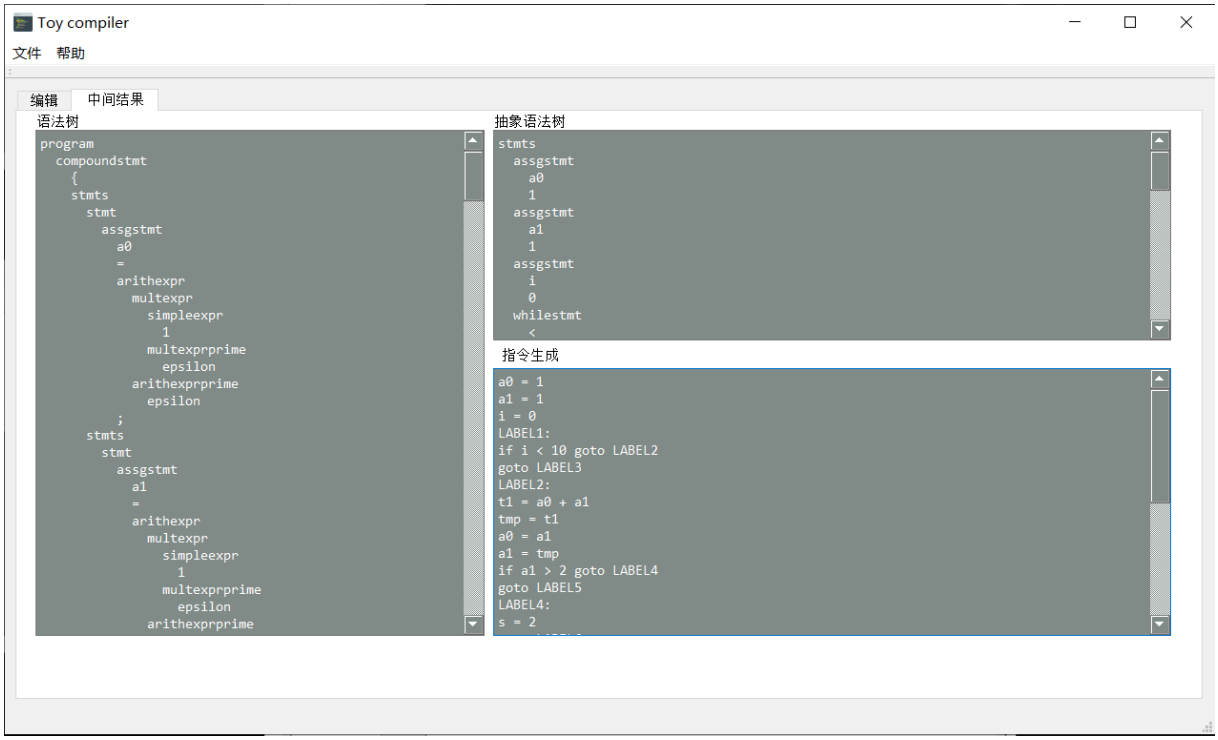

Toy compiler
文件 帮助
编辑
中间结果
{
a0 = 1;
a1 = 1;
i = 0;
while (i < 10) {
tmp = a0 + a1;
a0 = a1;
a1 = tmp;
if (a1 > 2) then {
s = 2;
} else {
s = 1;
}
输出:
compile success.
编译
运行
重置

运行功能：


Toy compiler
文件 帮助
编辑
中间结果
{
a0 = 1;
a1 = 1;
i = 0;
while (i < 10) {
tmp = a0 + a1;
a0 = a1;
a1 = tmp;
if (a1 > 2) then {
s = 2;
} else {
s = 1;
}
输出:
a0: 1
a1: 2
i: 0
tmp: 2
s: 0
t: 0
ret: 2
编译
运行
重置

中间结果界面：

可查看输入的 toy 文件所生成的语法树、抽象语法树和生成指令。



六、 测试

本节主要介绍项目中使用的测试技术。从测试方法上看，主要分为单元测试和集成测试两种。

（一）单元测试

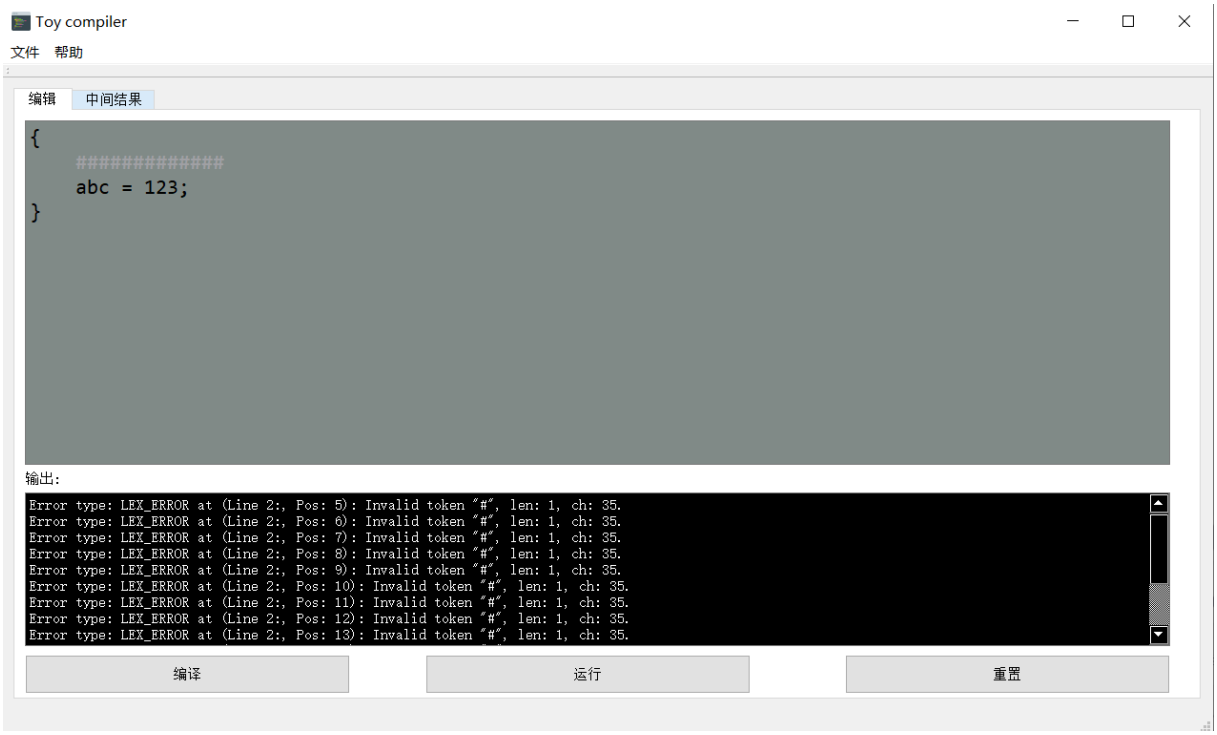
本项目设计实现了 c 语言的自动化单元测试框架，可以针对导出的函数接口编译生成静态库，链接到测试代码执行测试。单元测试主要用于测试项目内部自行实现的各种数据结构以及便于独立测试 scanner 等。

（二）自测用例

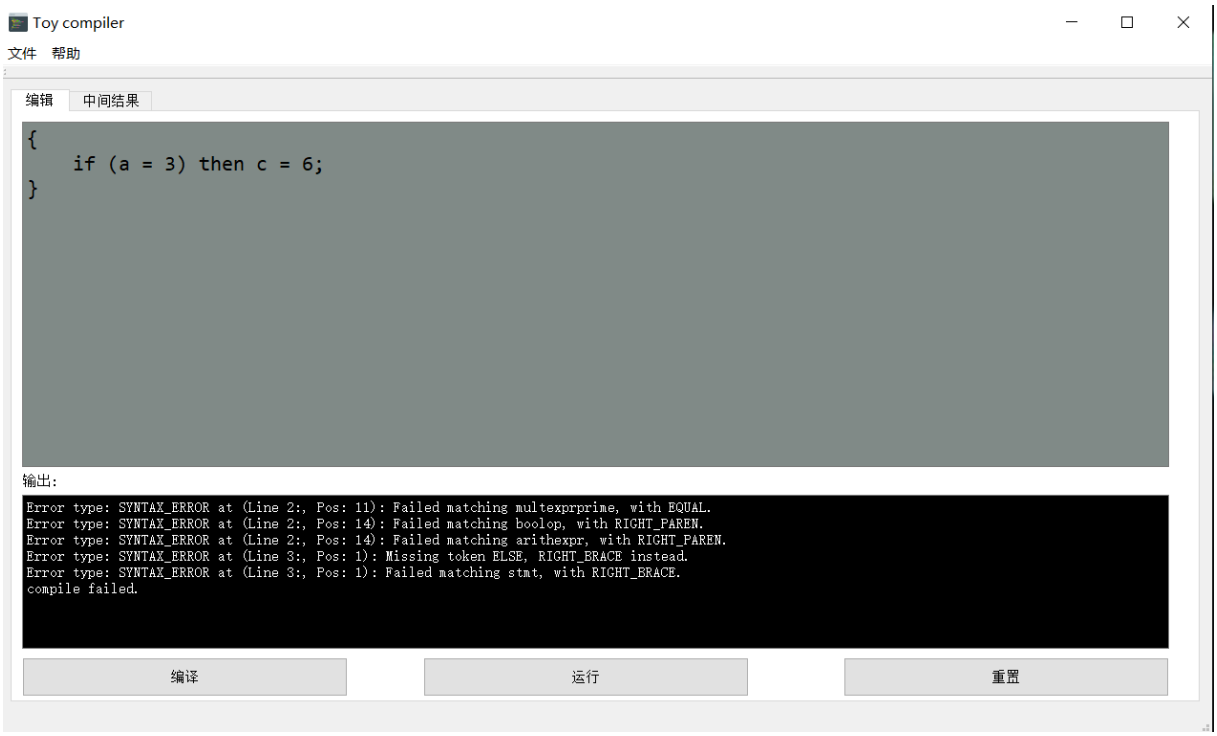
本小节主要介绍集成测试用例。主要分为词法、语法、语义错误用例以及无编译错误的可执行用例的测试。具体如下：

可执行用例测试：（文件可正常编译并运行显示结果）

词法错误测试：（编译失败则不能运行）



语法错误测试：（编译失败则不能运行）



语义错误测试：（编译失败则不能运行）

（三）检查用例

七、 总结

（一）成员分工

组长：李嘉睿（40%），组员：陈可欣（30%），孔露萍（30%）

（二）总结与展望

该项目主要是通过代码实现自行设计编译器，实现其主要功能，该项目中主要考虑词法、语法、语义三类问题。在词法部分以 toy 文件源码输入，按需生成 token。在语法方面是以改词法生成的 token 作为输入，最终生成语法树。语义分析部分主要检查是否定义变量，并构造符号表。并且实现编译器的交互操作，通过前端 UI 输入相关用例，可查看用例代码的编译、运行的结果，以及程序运行的中间结果抽象语法树和生成的字节码指令。该项目开发流程，主要是确定项目方向和需求分析，查阅参考相关资料和文献，再制定项目计划开发工期表，再根据开发工期表来分配项目部分和开发时间，先实现后端相关算法，再设计前端 UI 界面，之后调试两端接口，实现编译器的交互并输出相关结果。项目开发过程基本按照项目计划开发工期时间完成，再通过多项测试用例测试后，及时发现问题在查阅资料和相关网站后予以解决。

项目结果和原定项目计划基本符合，但是在语法树的实现上还存在一些问题。在原定开发计划中，要在前端 UI 展示生成的语法树相关图片，由于代码能力限制最后选择直接在输出框处输出语法树。后续可再对相关问题进行资料查询和研究，实现该语法树图片的展示。

八、 参考文献

[1] *Engineering a compiler*

[2] *编译原理实验手册1，2，3，4* 南京大学计算机科学与技术系 许畅等

[3] *Compilers: Principles, Techniques, and Tools*

[4] *Compiler Construction: Principles and Practice*