

# 编译器设计文档

---

18231047 王肇凯

## 编译器设计文档

- 整体架构

- 词法分析

  - 最初设计

  - 实现与完善

- 语法分析

  - 最初设计

  - 实现与完善

- 错误处理

  - 最初设计

  - 实现与完善

    - 错误类

    - 符号表

- 代码生成

  - 最初设计

    - 中间代码：

    - MIPS代码：

  - 实现与完善

- 竞速优化

  - 窥孔优化

  - 常量传播

  - 寄存器分配

  - 循环跳转优化

  - 乘除法优化

## 整体架构

---

```

1  int main() {
2      cout <<
3      "::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::"
4      << endl;
5      cout << "::"
6      "::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::"
7      << endl;
8      cout << "::" << endl;
9      cout << "wzk's compiler V1.0"
10     << endl;
11     cout << "wzk's compiler V1.0"
12     << endl;
13     cout << "wzk's compiler V1.0"
14     << endl;
15     cout << "wzk's compiler V1.0"
16     << endl;
17     cout << "wzk's compiler V1.0"
18     << endl;
19     cout << "wzk's compiler V1.0"
20     << endl;
21     cout << "wzk's compiler V1.0"
22     << endl;
23     cout << "wzk's compiler V1.0"
24     << endl;
25     cout << "wzk's compiler V1.0"
26     << endl;
27     cout << "wzk's compiler V1.0"
28     << endl;
29     cout << "wzk's compiler V1.0"
30     << endl;

```

```
31
32     return 0;
33 }
34
```

## 词法分析

### 最初设计

以一个函数 `get_token()` 为核心，产生token并输出至文件；主函数中读取输入文件所有字符，循环调用此函数读取字符生成token，直至文件末尾。

`get_token()` 内部参考课本上的设计，读取非空白符，连续读取标识符并判断是否是保留字，连续读取整数，读取单个字符判断是否为一元分隔符，连续读取判断是否为二元分隔符。若以上一条满足，将当前token的字符串表示及分类分别存入 `token` 和 `symbol` 并输出至文件，否则产生异常。

### 实现与完善

为了方便以后扩展，将词法分析写为一个类 `TokenAnalyze`，在初始化时指定是否输出分析结果至文件，并调用其 `analyze()` 方法进行词法分析主过程。将 `token`、`symbol` 等全局变量作为该类的成员变量。

```
1  class Lexer {
2  public:
3      char ch{};
4      string token;
```

```

5     string symbol;
6     string source;
7     int pos = 0;
8     int line_num = 1;
9     int col_num = 1;
10    bool replace_mode;
11
12    Token analyze();
13    Token get_token();
14    int read_char();
15    void retract();
16    static string special(char);
17    static string reserved(string);
18    explicit Lexer(const string& in_path, bool replace);
19 };

```

`analyze()` 最开始将文件内容读入成员变量 `source`，并维护整数值 `pos` 记录正在读取的字符串下标。这样做是为了方便回退，只需 `pos--` 即可。从而 `read_char()` 每次读取字符时只需更新 `ch=source[pos]` 并使 `pos++`，更新行号 `line_num` 和列号 `col_num`（见后）。

在 `reserver()` 和 `special()` 分别维护两个 `map` 型变量记录保留字和非歧义分隔符。这里歧义指的是 `>` 与 `>=` 等需要多读取一个字符才能区分的分隔符，进行特判。

对于 `INTCON`，记录其值在成员变量 `int_v` 中。

词法分析异常包含以下几类：

- 引号不匹配：读取到文件结尾仍未发现右双引号/右单引号
- 字符个数过多：两个单引号内字符多于一个
- 未知字符：所有判断条件均不满足的字符

为了更好的异常处理提示信息，记录了正在读取的字符的行数和列数，并在报错时输出行数列数。同时输出的还有提示信息，指出异常属于以上某个特定类别。

词法分析的输出是 `Token` 类的若干实例。每个实例存储了一个词的语法成分、内容、行号、列号等信息。

```
1  class Token {
2  public:
3      string type;
4      string str;
5      string original_str;
6      int v_int = -1;
7      char v_char = 'E';
8      int line{};
9      int column{};
10     int pos{};
11
12     Token(string t, string s, int l, int c, int p);
13 };
```

## 语法分析

### 最初设计

采用递归子程序法进行自顶而下的分析。在调用子程序前先读入一个 token，然后每个子程序内通过读入 token 以及递归调用其他子程序来分析一种非终结符号，并将语法成分输出到文件中。

分析文法得知不存在左递归，为了避免回溯，采用了预读的方法，即在多个选择间存在冲突时提前读 1~3 个 token 进行判断出唯一选择，并回退到预读前的 token，然后调用该选择的子程序。

当读入的终结符号与预期不同时，由于避免了回溯，因此直接产生异常，保存至错误表中。

其中注意到 <有返回值函数调用> 与 <无返回值函数调用> 的语法完全一致，需要根据语义区分。因此建立简单符号表（等到语义分析时再加以完善），在函数定义时在符号表中添加“函数标识符 — 有无返回值”的映射，在需要区分函数调用时通过标识符查表，从而选择有返回值或无返回值函数调用。

## 实现与完善

设计 `Grammar` 类进行语法分析工作。将词法分析器作为语法分析器的成员变量，从而可以调用其 `analyze()` 方法产生新的 `Token`，以便进行语法分析主过程。

将符号表 `SymTable` 等全局变量作为该类的成员变量。

考虑到预读后需要回退至预读前的位置，因此将待输出内容按行保存至 `output_str` 中，在回退时删去上一个词法分析的输出行。

成员变量 `tk`，`sym`，`pos` 分别保存当前读到的词法分析结果、结果的词法成分、在所有词法分析tokens中的位置。

方法 `next_sym()`，`retract()`，`error()`，`output()` 分别进行读入 token、预读结束后回退、存储错误、输出语法成分。

各个递归子程序作为方法保存在类中。

```
1  class Grammar {
2  public:
3      GrammarMode mode;
4      Lexer lexer;
5      vector<string> output_str;
6      vector<Token> cur_lex_results;
7      int pos = 0;
8
9      Token tk;
10     string sym = "";
```

```
11     int local_addr = LOCAL_ADDR_INIT;
12
13     void error(const string &expected);
14     int next_sym(bool);
15     void retract();
16     int analyze();
17
18     explicit Grammar(const string &in_path, GrammarMode
mode);
19
20     void output(const string &name);
21     void save_to_file(const string &out_path);
22
23     void Program();
24     void ConstDeclare();
25     void ConstDef();
26     void UnsignedInt();
27     string Int();
28     string Identifier();
29     pair<DataType, string> Const();
30     void VariableDeclare();
31     void VariableDef();
32     void TypeIdentifier();
33     void SharedFuncDefHead();
34     void SharedFuncDefBody();
35     void RetFuncDef();
36     void NonRetFuncDef();
37     void CompoundStmt();
38     void ParaList();
39     void Main();
40     pair<DataType, string> Expr();
41     pair<DataType, string> Item();
42     pair<DataType, string> Factor();
43     void Stmt();
44     void AssignStmt();
45     void ConditionStmt();
46     pair<string, string> Condition();
```

```

47     void LoopStmt();
48     void CaseStmt();
49     void SharedFuncCall();
50     void RetFuncCall();
51     void NonRetFuncCall();
52     void StmtList();
53     void ReadStmt();
54     void WriteStmt();
55     void ReturnStmt();
56
57
58     void add_node(const string &name);
59     void add_leaf();
60     void tree_backward();
61     void dfs_show(const TreeNode &, int);
62     void show_tree();
63 };

```

`analyze()` 只需打开关闭输出文件流、读入第一个token、调用<程序>子程序即可。

<数字>，<标识符>等基础的非终结符号在词法分析时已经进行过判断，因此不必写子程序。

每个递归子程序在调用前需要先使用 `next_sym()` 读入一个token，然后根据右部各选择的首符号进行选择（必要时采用预读），对于非终结符号调用其子程序，终结符号则判断是否与预期一致，不一致则报错。

## 错误处理



# 最初设计

建立错误类，存储错误类型、行号、列号、额外提示信息；再将所有错误对象整合为一个数组存储，并输出到文件中。

为了识别语义错误，需要建立符号表管理各个层次的变量，并存储各种信息（如变量维度、函数返回值类型等等）。在读到标识符时进行增/查操作，在类型不一致时报错。此外还要求表达式的类型，判断其为char型或者int型。

错误处理的主要步骤均在语法分析中完成。语法分析时调用符号表类和错误类的静态方法完成添加符号表项、添加错误等任务。在语法分析结束时将错误格式化输出到文件中。

## 实现与完善

### 错误类

将错误类别以宏的形式进行定义，作为错误编码。

```
1  #define ERR_LEXER 'a'
2  #define ERR_REDEFINED 'b'
3  #define ERR_UNDEFINED 'c'
4  #define ERR_PARA_COUNT 'd'
5  #define ERR_PARA_TYPE 'e'
6  #define ERR_CONDITION_TYPE 'f'
7  #define ERR_NONRET_FUNC 'g'
8  #define ERR_RET_FUNC 'h'
9  #define ERR_INDEX_CHAR 'i'
10 #define ERR_CONST_ASSIGN 'j'
11 #define ERR_SEMICOL 'k'
12 #define ERR_RPARENT 'l'
13 #define ERR_RBRACK 'm'
14 #define ERR_ARRAY_INIT 'n'
15 #define ERR_CONST_TYPE 'o'
```

```
16 | #define ERR_SWITCH_DEFAULT 'p'
```

设计 `Error` 类存储单个错误的各种信息。 `Errors` 类用静态成员变量存储全部错误对象，并提供静态方法将错误输出至文件。

```
1 | class Error {
2 | public:
3 |     string msg;
4 |     int line{};
5 |     int column{};
6 |     int eid;
7 |     char err_code{};
8 |     string rich_msg;
9 | };
10 | class Errors {
11 | public:
12 |     static vector<Error> errors;
13 |     static void add(const string &s, int line, int col,
14 | int id);
15 |     static void add(const string &s, int id);
16 |     static void save_to_file(const string &out_path);
17 | };
```

错误信息输出示例如下

```
1 | Error in line 52, column 21: Para count mismatch (EID:
2 | d)
3 | Error in line 53, column 22: Para type mismatch (EID: e)
4 | Error in line 55, column 21: Para type mismatch (EID: e)
```

## 符号表

`SymTableItem`和`SymTable`类分别代表符号表项和整个符号表。设定了增加符号表项、查询符号表、栈式符号表增减层的方法。

```
1  enum STIType {
2      invalid_sti,
3      constant,
4      var,
5      tmp,
6      para,
7      func
8  };
9
10 enum DataType {
11     invalid_dt,
12     integer,
13     character,
14     void_ret
15
16 };
17
18 class SymTableItem {
19 public:
20     string name;
21     STIType stiType;
22     DataType dataType;
23     int dim = 0;
24     vector<pair<DataType, string>> paras;
25     int addr{};
26     int size{};
27     int dim1_size{};
28     int dim2_size{};
29     string const_value;
30
31     SymTableItem(string name, STIType stiType1, DataType
        dataType1, int addr);
```

```

32     string to_str() const;
33 };
34
35 class SymTable {
36 public:
37     static vector<SymTableItem> global;
38     static map<string, vector<SymTableItem>> local;
39     static unsigned int max_name_length;
40
41     static void add(const string &func, const Token &tk,
STIType stiType, DataType dataType, int addr, int dim1,
int dim2);
42     static void add_const(const string &func, const
Token &tk, DataType dataType, string const_value);
43     static int add_func(const Token &tk, DataType
dataType, vector<pair<DataType, string>> paras);
44     static SymTableItem search(const string &func, const
string &str);
45     static void show();
46     static void reset();
47 };

```

在最初的设计中符号表为栈式，但后续根据中间代码生成目标代码时，发现将每个函数的符号表均保存起来更加便于查找。最终的设计中符号表包括一个存储全局变量、函数的 `global` 成员变量，和一个将函数名映射到函数内部变量、参数的 `local` 变量。在向符号表添加项时需要指明作用域（某个函数内部或是全局）。

符号表显示效果如下

```

1  =====
2  NAME                KIND   TYPE  DIM
3  -----
4  func_switch_ch     func   int   0
5  func_switch_int    func   int   0
6  -----
7  c                  para   char  0
8  tmp                var    int   0
9  =====

```

在语法分析程序中增加对于 `error()` 函数的调用，以在适当地方进行报错，完成跳读，并将错误信息存储到 `Errors` 类的静态成员变量中。在整个程序分析结束后，将所有错误输出至文件。

## 代码生成

### 最初设计

代码生成部分涵盖范围最大，包括课本上存储分配、中间代码格式、语法制导翻译、语义分析等四章内容。在设计时将其分为两个阶段：

- 源代码->中间代码：即语义分析在语法分析子程序里生成四元式形式即可
- 中间代码->目标代码：扫描中间代码，同时查阅符号表得到变量的数据类型、维度等信息，生成目标代码。

### 中间代码：

设计为四元式的形式，即（运算符，操作数1，操作数2，结果），多个四元式存储在全局的静态变量中。在原先的每个语法分析子程序中增加语义分析的内容生成中间代码。本次作业涉及到的操作符包括加减乘除、读、写、赋值七种。

例如对于 `E = A op B op C op D` 的形式（op为同级运算符，例如乘除或加减），按照翻译文法生成的序列为：

```
1 op, A, B, #T1
2 op, #T1, C, #T2
3 op, #T2, D, #T3
4 :=, E, #T3
```

同时，在进入函数时产生 `FUNC void main` 的四元式用于标识作用域。

运算符包括以下种类：

```
1 #define OP_PRINT "PRINT"
2 #define OP_SCANF "SCANF"
3 #define OP_ASSIGN "!="
4 #define OP_ADD "+"
5 #define OP_SUB "-"
6 #define OP_MUL "*"
7 #define OP_DIV "/"
8 #define OP_FUNC "FUNC"
9 #define OP_END_FUNC "END_FUNC"
10
11 #define OP_ARR_LOAD "ARR_LOAD"
12 #define OP_ARR_SAVE "ARR_SAVE"
13 #define OP_LABEL "LABEL"
14 #define OP_JUMP_IF "JUMP_IF"
15 #define OP_JUMP_UNCOND "JUMP"
16
17 #define OP_PREPARE_CALL "PREPARE_CALL"
18 #define OP_CALL "CALL"
19 #define OP_PUSH_PARA "PUSH_PARA"
20 #define OP_RETURN "RETURN"
```

## MIPS代码：

语法分析结束后，根据中间代码借助符号表生成。将字符串以 `.asciiz` 存在数据区，全局变量存在 `$gp` 上方，其余变量存在内存中（后续优化为寄存器）。

在进行赋值和四则运算操作时，根据四元式的操作数在内存/寄存器/为常量分情况处理。

```
1  /* 对于a=b:
2  * a在寄存器, b在寄存器: move a,b
3  * a在寄存器, b在内存: lw a,b
4  * a在寄存器, b为常量: li a,b
5  *
6  * a在内存, b在寄存器: sw b,a
7  * a在内存, b在内存: lw reg,b  sw reg,a
8  * a在内存, b为常量 li reg,b sw reg,a
9  */
```

```
1  /* 对于a=b+c:
2  * abc都在寄存器/常量:          add a,b,c
3  * ab在寄存器/常量, c在内存 (或反过来):  lw reg2,c  add
    a,b,reg2
4  * a在寄存器, bc在内存:          lw reg1,b  lw reg2,c
    add a,reg1,reg2
5  * a在内存, bc在寄存器/常量:      add reg1,b,c  sw
    reg1,a
6  * ab在内存, c在寄存器/常量 (或反过来):  lw reg1,b  add
    reg1,reg1,c  sw reg1,a
7  * abc都在内存:                  lw reg1,b  lw reg2,c  add
    reg1,reg1,reg2  sw reg1,a
8  */
```

# 实现与完善

中间代码类实现如下：

```
1  class PseudoCode {
2  public:
3      string op;
4      string num1;
5      string num2;
6      string result;
7
8      PseudoCode(string op, string n1, string n2, string
r);
9  };
10
11 class PseudoCodeList {
12 public:
13     static vector<PseudoCode> codes;
14     static int code_index;
15     static vector<string> strcons;
16     static int strcon_index;
17
18     static string add(const string &op, const string
&n1, const string &n2, const string &r);
19     static void refactor();
20     static void show();
21     static void save_to_file(const string &out_path);
22 };
```

语义分析时调用静态方法 `PseudoCodeList::add` 生成四元式。

MIPS生成类维护变量记录当前函数作用域，方便在符号表中查找。

翻译过程基本依照前文设计。例如对于四则运算的处理如下：

```
1  bool a_in_reg = in_reg(code.num1);
2  bool b_in_reg = in_reg(code.num2);
```



```

3  string a = symbol_to_addr(code.num1);
4  string b = symbol_to_addr(code.num2);
5  string reg = "$k0";
6
7  if (a_in_reg) {
8      if (b_in_reg) {
9          generate("move", a, b);
10     } else if (is_const(code.num2)) {
11         generate("li", a, b);
12     } else {
13         generate("lw", a, b);
14     }
15 } else {
16     if (b_in_reg) {
17         generate("sw", b, a);
18     } else if (is_const(code.num2)) {
19         generate("li", reg, b);
20         generate("sw", reg, a);
21     } else {
22         generate("lw", reg, b);
23         generate("sw", reg, a);
24     }
25 }

```

同时为了方便debug，在每次翻译一条中间代码时生成一条注释，以表示连续的几条语句的目的。

```

1  # == #T170 = #T169 * num2 ==
2  mul $t2, $t1, $s0

```

## 竞速优化

优化包括以下种类：

- 局部窥孔优化
- 常量传播
- 寄存器分配
- 循环跳转优化
- 乘除法优化

## 窥孔优化

一些较少但是比较有效果的优化，例如：

- 连续的两个临时变量赋值语句 `#T1=A+B, #T2=#T1` 可以进行合并
- 连续的相同种类常数运算 `A=B/5, C=A/6` 合并
- 加减乘0、乘除1可以优化为赋值语句

## 常量传播

结果为临时变量的四则运算可以直接删去，然后存储其值即可（实际实现时存储在符号表中）；

四则运算中的两个操作数若为临时变量且其常数值可计算，也可以直接进行替换；

通过这样的方式，数组元素赋值 `a[1][2]=d`（列数为8）的语句从 `#T1=2, #T2 = 1 * 8, #T3 = #T1 + #T2, a[#T3]=d` 精简为 `a[10][2]=d`。

# 寄存器分配

局部变量和临时变量（四元式的中间结果）分别存在s寄存器和t寄存器，若寄存器无空闲则放在 `$sp$` 下方。维护变量记录t寄存器和s寄存器的使用情况以方便分配。s寄存器在进入新的函数时释放，t寄存器在第一次被读取时释放。

函数调用时保存当前已使用的寄存器至 `$sp`，调用结束时从内存恢复。

## 循环跳转优化

对于 `while (a>b) a++;`

优化前: `label1: jump_if a<=b label2, a=a+1, jump label 1`

优化后: `jump_if a<=b label2, label1: a=a+1, jump_if a>b label1`

从而循环体内每次循环可以减少一次跳转，大大降低了跳转开销。

## 乘除法优化

乘除法时判断操作数是否有2的自然数次幂的常数，若符合条件则可以用移位代替。需注意除法在被除数为负数时 `div` 和 `sra` 表现并不一致（前者向下取整，后者向上取整），采用的处理方式是判断被除数的正负，若为负数则先将被除数符号取反，做除法后再将结果取反。

- `b=a*8` 可以翻译为 `sll $s1, $s0, 3`
- `b=a/8` 可以翻译为：
  - `bgez $s0, label1`
  - `subu $t0, $zero, $s0`
  - `sra $s1, $t0, 3`

- `subu $s1, $zero, $s1`
- `j label2`
- `label1: sra $s1, $s0, 3`
- `label2:`

由于乘除法的惩罚是ALU运算和跳转的数倍，上述优化可以大幅减少开销。

此外，`div $t1, $t2, $t3`在实际运行时被Mars处理为扩展指令，翻译成四条：`bne $t2, $zero, label1`，`label1: break`，`div $t2, $t3`，`mflo $t1`，其中前两条为检查除数是否为0，保证源程序正确的前提下可以被略去。因此将除法简化为`div $t2,$t3`，`mflo $t1`。