编译原理 I
课程设计报告

班级：1620104

学号：162010112

姓名：陈立文

2023-02

目录

1. 设计任务

1.1 课程设计题目

一个 PASCAL 语言子集（PL/0）编译器的设计与实现。

1.2 PL/0 语言的 BNF 描述

```
<prog> → program <id>; <block>
<block> → [<condecl>][<vardecl>][<proc>]<body>
<condecl> → const <const>{,<const>};
<const> → <id>:=<integer>
<vardecl> → var <id>{,<id>};
<proc> → procedure <id> ([<id>{,<id>}]) ;<block>{;<proc>}
<body> → begin <statement>{;<statement>}end
<statement> → <id> := <exp>
              |if <lexp> then <statement>[else <statement>]
              |while <lexp> do <statement>
              |call <id> ([<exp>{,<exp>}])
              |<body>
              |read (<id>{, <id>})
              |write (<exp>{,<exp>})
<lexp> → <exp> <lop> <exp>|odd <exp>
<exp> → [+|-]<term>{<aop><term>}
<term> → <factor>{<mop><factor>}
<factor>→<id>|<integer>|(<exp>)
<lop> → =|<>|<|<=|>|>=
<aop> → +|-
<mop> → *|/
<id> → l{l|d}    （注：l 表示字母）
<integer> → d{d}
```

注释：

<prog>:程序	<block>:块、程序体	<condecl>:常量说明	<const>: 常量
<vardecl>:变量说明	<proc>:分程序	<body>:复合语句	<statement>:语句
<exp>:表达式	<lexp>:条件	<term>:项	<factor>: 因子
<aop>:加减运算符	<mop>:乘除运算符	<lop>:关系运算符	

该文法没有左递归和回溯问题，同时没有空字 ε，同时已经解决了表达式的二义性问题，是一个 LL(1)文法，可以直接进行 LL(1)分析。

2. 系统设计

2.1 系统结构

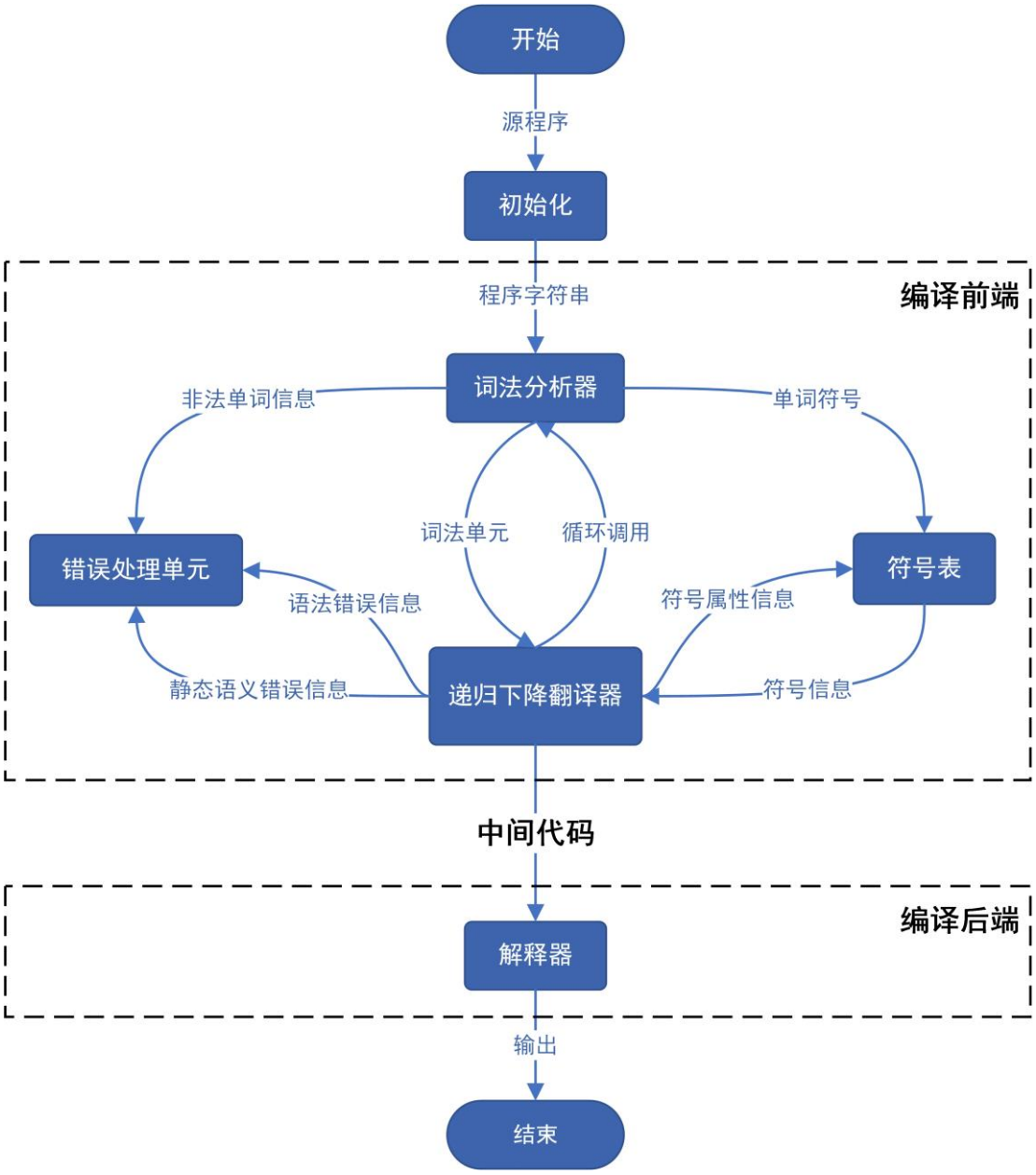


图 2.1 编译器的系统结构

编译器系统的结构如图 2.1 所示。箭头方向代表数据的流向。

为了方便中文字符的识别，编译器统一编码格式为 UCS-2，编译器中的所有字符均使用占用 2 字节的宽字符类型。

下面将分别介绍各部件的设计与实现方式。

2.2 初始化

基本介绍：

本模块对应代码中的 PL0.h 和 PL0.cpp 文件。

本模块用于完成编译器运行前的一些准备工作。

主要功能：

- **将 UTF-8 格式的源代码文本文件读取为 UCS-2 字符串，实现中文字符的识别**

准确地说，UCS-2 和 UTF-8 都是 Unicode 的实现方式，Unicode 为每一个字符的抽象定义了一个定长的 4 字节码点，但由于每一个字符都用 4 字节表示太浪费空间，而且也不兼容 ascii 码标准，于是采用变长多字节编码实现 Unicode，也就是 UTF-8 标准，如此不仅在单字节编码完全兼容 ascii 码标准，而且也节省了空间。而 UCS-2 就是将每一个码点用定长 2 字节存储，不做任何改变。虽然 UCS-2 不能编码所有的 Unicode 码点，但是对于码点只需 2 字节的中文字符已经足够，而 C++中的宽字符类型（2 字节）也正好满足了这一要求。所以只需将 3~4 字节的 UTF-8 格式中文字符转换为 2 字节的 UCS-2 格式，便可实现中文字符的识别。

注：UTF-8 编码格式（xxx 用来填充二进制 Unicode 码点）

1 字节　0xxxxxxx

2 字节　110xxxxx 10xxxxxx

3 字节　1110xxxx 10xxxxxx 10xxxxxx

4 字节　11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- **定义编译器的全局变量和全局宏定义，如层级，first、follow 集等**
- **初始化编译器的各个部件，创建必要的对象，开辟内存**
- **设置输入输出流为 UCS-2 格式**

具体实现：

函数声明	说明
void init();	初始化全局变量与各个部件，并统一输入输入流为 Unicode 格式
void readFile2USC2(string);	读取指定路径的 UTF-8 格式文件，将其转换为 UCS-2（宽字符）字符串

2.3 词法分析器

基本介绍：

本模块对应代码中的 Lexer.h 和 Lexer.cpp 文件。

词法分析器是一个子程序，接受从文件中读取出的 Unicode 程序字符串，依据图 2.2 所示的状态转换图，从头至尾分析字符串中出现的所有同步符号，每分析出一个词法单元便提供给语法分析器使用，循环往复直到程序末尾。

主要功能：

- **定义了同步符号的表示方式**

采用 one-hot 独热编码，每一个同步符号采用 31 位‘0’+1 位‘1’的方式组成独立的 4 字节编码。这样实现的好处是只需要位操作‘或’就可以简单地实现同步符号集合的定义，位操作‘与’便可以判断当前符号是否在某一集合中。如此实现运行效率高且代码简洁，可以很方便地定义 first 集与 follow 集；缺陷便是浪费空间，若定义更多的同步符号，32 位编码长度是不够的，需要耗费更多的空间。

- **实现同步符号的读取与识别**

依次读取程序字符串中的每一个符号，依照图 2.2 所示的状态转换图进行分析，直到读取到‘#’(‘#’是我手动在读取出来的程序字符串末尾添加的，作为程序的结束符号)。由于标识符和关键字可能会有重复的部分，想要识别出正确的符号，必须进行“超

前搜索”，直到超前搜索到界符，才会停止当前符号的识别。

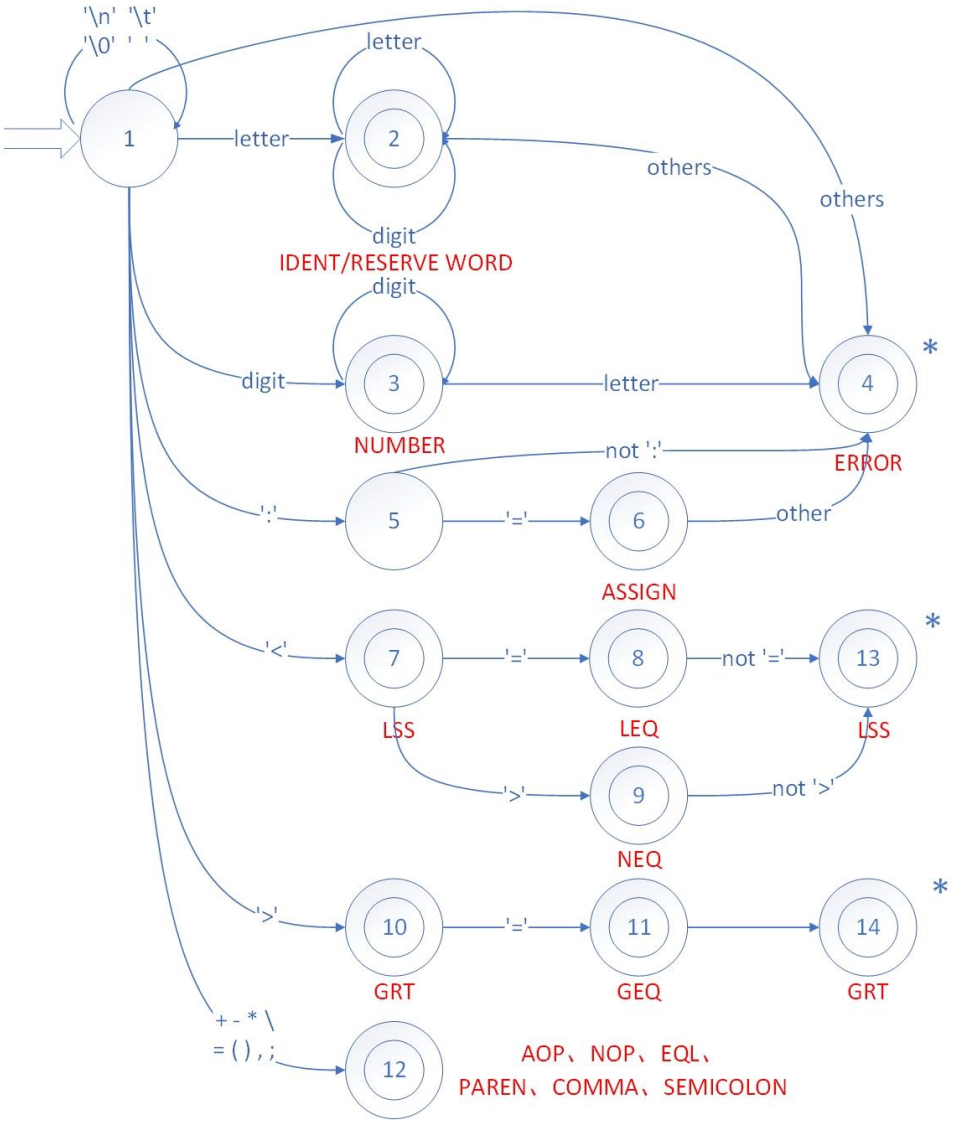


图 2.2 语法分析器的状态转换图

- **读取到非法字符时调用错误处理单元**

具体实现：

函数声明	说明
bool isDigit(wchar_t ch);	判断给定字符是否为数字
bool isLetter(wchar_t ch);	判断给定字符是否为字母
bool isTerminate(wchar_t ch);	判断给定字符是否为界符
int isOprator(wchar_t ch);	判断给定字符是否为运算符
void skipBlank();	使当前字符读指针跳过连续的空白符，直到

	遇到第一个非空白符
void getCh();	读取下一个字符
void retract();	回退超前搜索的字符
void contract();	将当前读取到的字符追加到同步符号字符串
int reserve(wstring str);	查看指定字符串是否为关键字
void getWord();	读取并识别当前读指针遇到的最近同步符号
void initLexer();	词法分析器初始化函数

流程图：详情见附件 4.3.1

2.4 错误处理单元

基本介绍：

本模块对应代码中的 ErrorHandler.h 和 ErrorHandler.cpp 文件。

错误处理单元接收来自词法分析器、语法分析器、语义分析过程中的错误信息，处理后将其输出至控制台。

主要功能：

- **定义错误码以及其对应的错误信息**

其他模块调用错误处理单元时，只需要提供错误码和额外信息便可打印出错误信息。错误码实际上是错误信息数组的下标，错误信息数组里存储的是一些带有占位符或不带的格式化字符串，占位符是为打印额外信息准备的。

- **准确地告知错误出现的位置**

由于不同类型的错误应当指示的报错位置是不同的，比如缺少‘;’的错误，只有读到了下一行的开始符号才能检测到，而非法标识符错误，在出现错误的地方就能立刻检测到。因此在词法分析器中，我维护了四个指针，即当前的行、列指针和上一个符号尾的行、列指针，可以让词法分析器为不同类型的错误打印不同的报错位置信息。

具体实现：

函数声明	说明
void error(unsigned int n, T… extra);	错误处理函数，将额外信息加入到错误码指定的格式化字符串中，然后调用打印函数
void printPreWord(const wchar_t msg[])	打印指定错误信息，错误位置为上一个符号尾的行、列指针
void printCurWord(const wchar_t msg[])	打印指定错误信息，错误位置为当前行、列指针
void over();	统计错误个数信息并打印
void initErrorHandler();	错误处理单元初始化函数

2.5 符号表

基本介绍：

本模块对应代码中的 SymTable.h 和 SymTable.cpp 文件。

维护分析过程中出现的所有符号，用于静态语义检查、中间代码生成以及运行时内存分配。

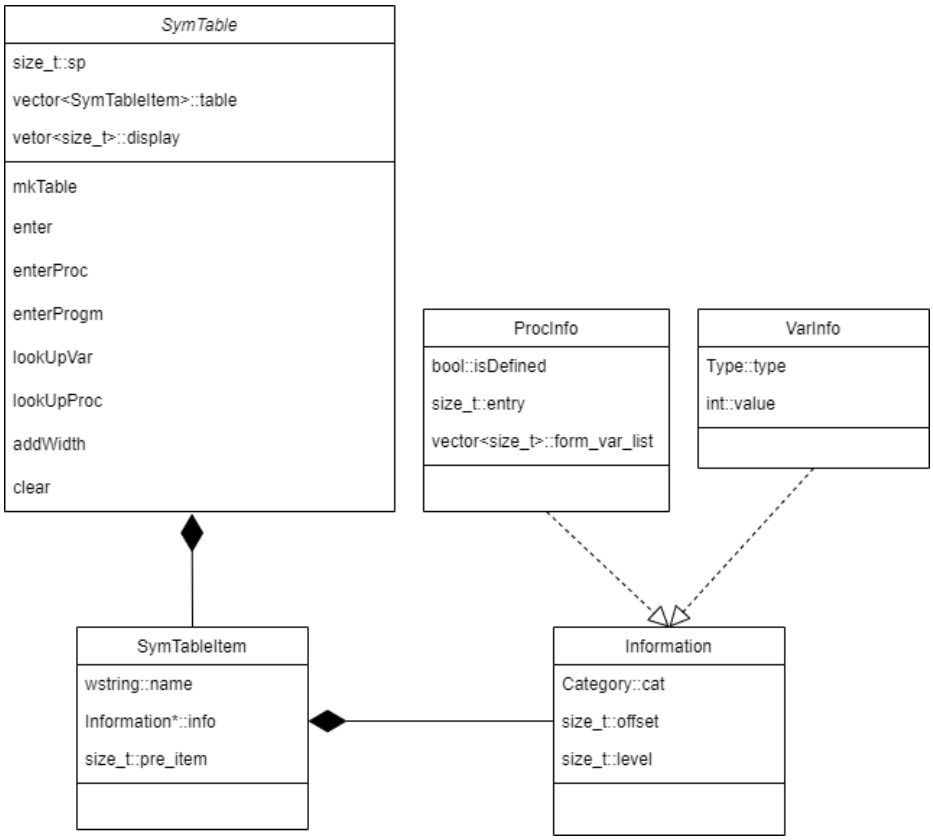


图 2.3 符号表的简单类图

如图 2.3 所示，我实现的符号表主要有三个部件：

sp 表示当前正在分析的子过程的地址，该属性是为了实现跨子过程时的回填；

table 表示符号表的主体，该表的每一项有 name 符号名、info 符号信息、pre_item 与其相连的上一个符号表项地址，共三个字段，其中 info 又根据符号类型的不同，分为 varInfo 变量信息和 procInfo 过程信息两类；

display 是用来维护不同层级最近登入的符号地址的数据栈，栈顶为当前的层级 level+1。

主要功能：

- **静态语义检查**

  上下文无关文法由于没有记忆性，无法检查出**符号是否声明、符号是否重定义、形参与实参列表是否匹配**的问题。此外，由于过程的嵌套定义产生了作用域的概念，相同名字的符号在不同的作用域也有不同的含义，内层作用域可以使用外层的符号，为了解决这些问题，需要借助符号表的帮助。

- **运行时内存分配**

  维护了一个全局 offset，用来记录变量在子过程中的相对地址，以及整个子过程所占用的内存空间大小。在登入变量符号时将当前 offset 作为符号的属性同时登入，并令 offset += 4（此变量占用的内存大小）；在过程的声明部分结束后，调用 addWidth()函数，将当前 offset 作为当前过程的所占空间的属性记入符号表，然后将 offset 置 0，用于下一个子过程。

当编译运行生成到内存分配相关的指令时，会依据符号表中的 offset 属性进行生成。

- **中间代码生成**

  除了上述的内存分配任务需要符号表以外，还有以下情况的中间代码生成需要：

  - 右值替换

    使用常量类型的变量为右值，不会在运行时刻分配内存地址，而是使用符号表中记录的值直接替换。

  - 跨子程序回填

    由于不同子程序之间的信息不能共享，对于部分跨子过程的回填操作，需要借助符号表的帮助，例如到主程序体的跳转指令的回填。在生成需要被回填的跳转指令时，我会将当前跳转指令的地址与当前过程的符号的 entry 属性绑定，遇到当前过程的过程体的第一条中间代码语句时，便将该语句的地址回填至所在过程符号绑定的 entry 指令处。

具体实现：

下面介绍符号登入符号表的具体流程：

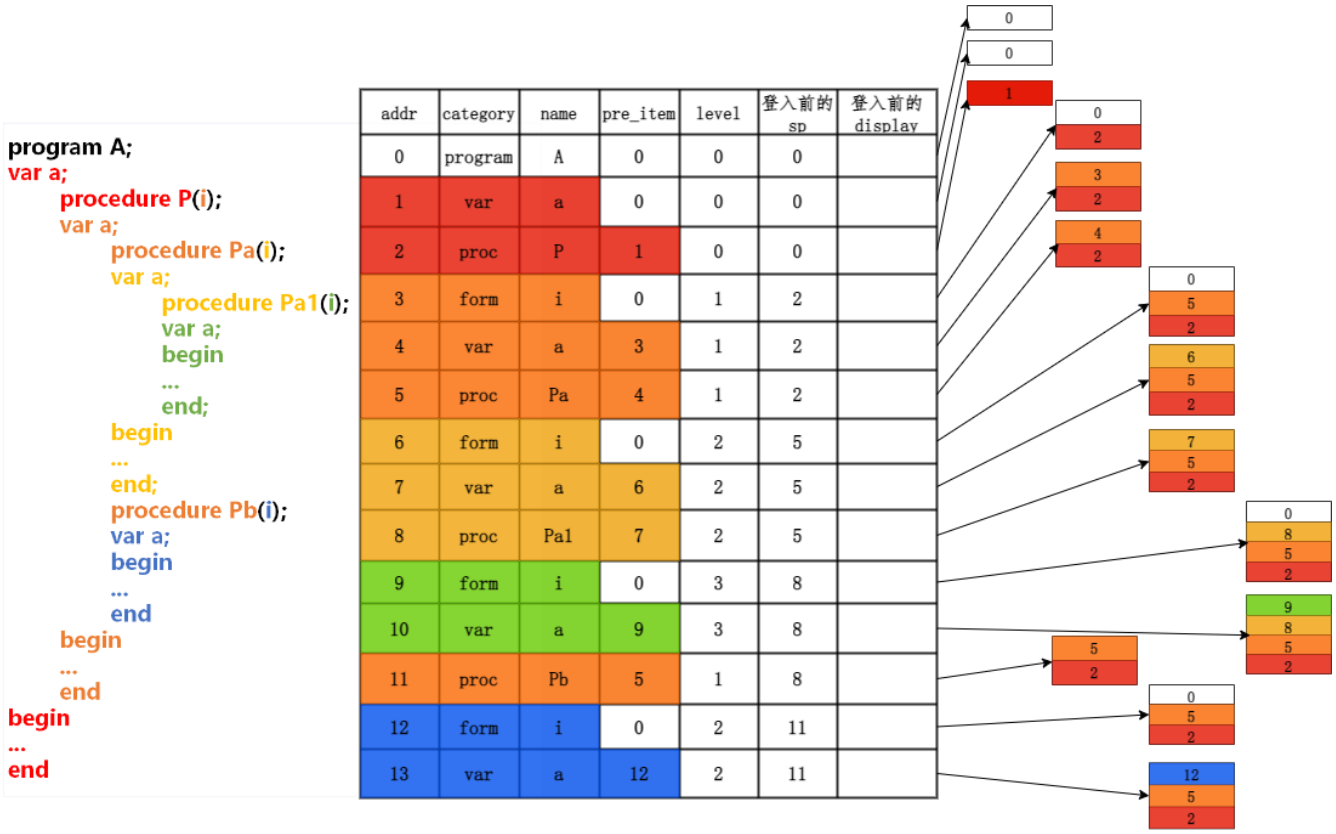


图 2.4 符号登入符号表的流程

如图 2.4 所示，最左边是待执行的程序源代码，中间是符号表本体，最右边是符号表的 display 表。符号表中的颜色对应源代码中的颜色，不同的颜色区分了不同的作用域。符号表的最终结果就是让相同颜色的符号能够依照 pre_item 这一项形成一条符号链，pre_item 为 0 时则代表该符号是当前符号链的第一项。而 display 表中维护的，也就是不同层级最近一次登入符号表的符号，当前符号只需要连接到 display 表对应层级的符号，就可以实现同层级符号的串联。

由于 program 符号的特殊性，其既不参与过程调用，也不参与符号重名的判断，因此当读取到 program　A 时，使用 enterProgm 将‘A’登入符号表的第 0 项，其他符号

表项无法访问这一符号。可以看到，此时 display 表只有 0 这一项。

若读取到一个变量符号的声明时，首先搜索在同一作用域是否有重名的变量符号，如果是常量符号，需要将其值写入符号表中。接着，将 display[level]写入当前符号表项的 pre_item 处，然后将当前项的 addr 写入 display[level]。

若读取到一个过程符号的声明时，首先搜索在同一作用域是否有重名的过程符号，比如上图中的 Pa 和 Pb，若没有重名符号则首先更改 sp，然后同样地，将 display[level]写入当前符号表项的 pre_item 处，并将当前项的 addr 写入 display[level]。

若遇到 level++时，也就是分析到过程的形参列表时，此时 display 表会将 0 入栈；若遇到 level--时，只需弹出栈顶单元就可。

下面介绍使用符号时的运行流程：

当需要使用符号时，首先应当根据符号的类型调用相应的查找函数，查找该符号是否已经在符号表中定义，若未查找到，则为未声明错误，调用错误处理单元。由于过程嵌套定义的特殊性，尽管在内层过程可以查找到外层的过程符号，但是由于过程内子过程的声明一定在过程体之前，所以内层过程不可能调用外层过程，因此我在过程信息中加入了 isDefined 标记位，在该过程分析至 begin 时赋值为 true，表示该过程是否已经被定义。如果查找到将要使用的过程符号未定义，则为未定义错误，调用错误处理单元。

由于我设计的符号表一次编译时仅有一个，因此采用了静态类与静态方法实现。

下面是静态类 SymTable 的成员函数：

函数声明	说明
static void mkTable();	创建子过程表（修改 sp 位置）
static void addWidth(size_t addr, size_t width);	计算过程占用的内存大小，将 width 直接抄入指定 addr 对应位置
static int enter(wstring name, size_t offset, Category cat);	将变量名及其偏移量、种属登入符号表
static int enterProc(wstring name);	将过程名登入符号表
static void enterProgm(wstring name);	将程序名登入符号表
static int lookUpVar(wstring name);	查找变量符号在符号表中的位置
static int lookUpProc(wstring name);	查找过程符号在符号表中的位置
static void clear();	清空符号表

流程图：详情见附件 4.3.1

2.6 递归下降翻译器

基本介绍：

本模块对应代码中的 Parser.h 和 Parser.cpp 文件。

由于语义分析采用递归下降的一遍扫描方式实现，会在语法分析的过程中计算属性，执行语义规则，因此无法将语义分析单元拆分为一个独立的模块，所以该语义分析和语法分析部件所在文件一致。

2.6.1 语法分析

主要功能：

- **为每一个非终结符实现递归的子程序**

由于给定文法没有左递归与回溯，各个非终结符的候选式也没有公共左因子，所以当面临某一输入的终结符，当前非终结符选择的候选式是唯一确定的，所以读取到对应产生式的左因子便进入对应的产生式。若在产生式中遇到终结符，就查看其是否与文法中的符号相匹配，若遇到非终结符，则调用其子程序继续分析。

- **错误恢复**

语法分析应当最大限度地分析出所有存在的语法错误，便于用户修改源程序。因此，若在语法分析过程中遇到了不符合文法的情况，调用错误处理单元后，应当尽可能地将分析恢复至正常，不会影响程序其余部分的分析。

为了实现这一功能，我的大致思路是：遇到错误时选定某些可以恢复分析的同步符号，比如当前非终结符的 follow 集、分号等，继续读取直到遇到这些符号时停止，然后继续正常的分析流程。

具体实现：

函数声明	说明
void judge(unsigned long s1, unsigned long s2, unsigned int n, T... extra);	用于错误恢复的函数，若当前符号在 s1 中，则读取下一符号；若当前符号不在 s1 中，则根据 n 和 extra 调用错误处理函数，接着循环查找下一个在中 s1 ∪ s2 的符号
void constDef();	<const>常量定义子过程
void condecl();	<condecl>常量声明子过程
void vardecl();	<vardecl>变量声明子过程
void proc();	<proc>过程定义子过程
void exp();	<exp>表达式子过程
void factor();	<factor>表达式因子子过程
void term();	<term>乘除表达式子过程
void lexp();	<lexp>条件表达式子过程
void statement();	<statement>语句子过程
void body();	<body>程序体子过程

void block();	<block>过程块子过程
void prog();	<prog>主程序子过程
void analyze();	执行语法分析的入口函数

2.6.2 语义分析

主要功能：

- **提供中间代码的参数**

语义分析的目的其实就是为了确定产生中间代码的参数。

首先是 op 参数的确定。op 的确定应当根据具体分析的语句功能来确定例如与表达式相关的<factor>、<term>、<exp>需要 opr 和 lod 指令计算并存储表达式的值，而<statement>中的 call、read、write 语句则需要执行相应的 cal、red、wrt 指令，与条件相关的 if else、while 语句则需要 opr、jpc、jmp 指令的参与。

接着是 L 层级参数的确定。需要提供 L 参数的指令只有 lod、sto、cal 三条，其中 lod 和 sto 指令的 L 是为了跨活动记录访问内存单元服务的，而对于 sto，当 L 为-1 时，sto 是一条传递形参的特殊指令。无论是过程还是变量符号的层级，都可以在符号表中直接查找到。

最后是 a 参数的确定。a 参数的含义有 4 种，偏移量（相对地址）、绝对地址（中间代码下标）、常量的值、opr 指令的操作码。对于前三种可以在符号表中直接查找到，而 opr 的操作码同 op 一样，也能根据具体分析的语句直接确定。

具体实现：

PL0 文法自顶向下一遍扫描的翻译模式：详情请见 4.2　PL0 文法的翻译模式。

2.7 中间代码生成

基本介绍：

本模块对应代码中的 PCode.h 和 PCode.cpp 文件。

定义了中间代码的结构，实现了中间代码的存储、生成、回填。

主要功能：

- **结构**

定义了一个结构体 PCode

中间代码的具体形式：

op	L	a

op 段代表伪操作码

L 段代表调用层的层级

A 段代表相对地址或值

由于中间代码的运行单元使用 display 表进行跨不同层级间的查找，所以只需知道被查找单元所在的层级便可直接访问对应的活动记录，因此中间代码的第二参数为层级而不是层差。

- **存储**

使用 C++的 vector 存储，vector 的成员函数 size()返回值即 nextquad。

- **生成**

根据指定参数生成 PCode 对象，然后直接调用 vector 的 push_back()方法。

- **回填**

使用 backpatch()函数，通过下标访问直接修改指定中间代码。

具体实现：

函数声明	说明
static int emit(Operation op, int L, int a);	根据给出参数，生成一条中间代码并存储至存储单元
static void backpatch(size_t target, size_t addr);	回填函数，将 addr 回填至 target 处的中间代码的跳转地址处
static void printCode();	打印出存储单元中的所有中间代码
static void clear();	清除中间代码的存储单元

2.8 解释器

基本介绍：

本模块对应代码中的 Interpreter.h 和 Interpreter.cpp 文件。

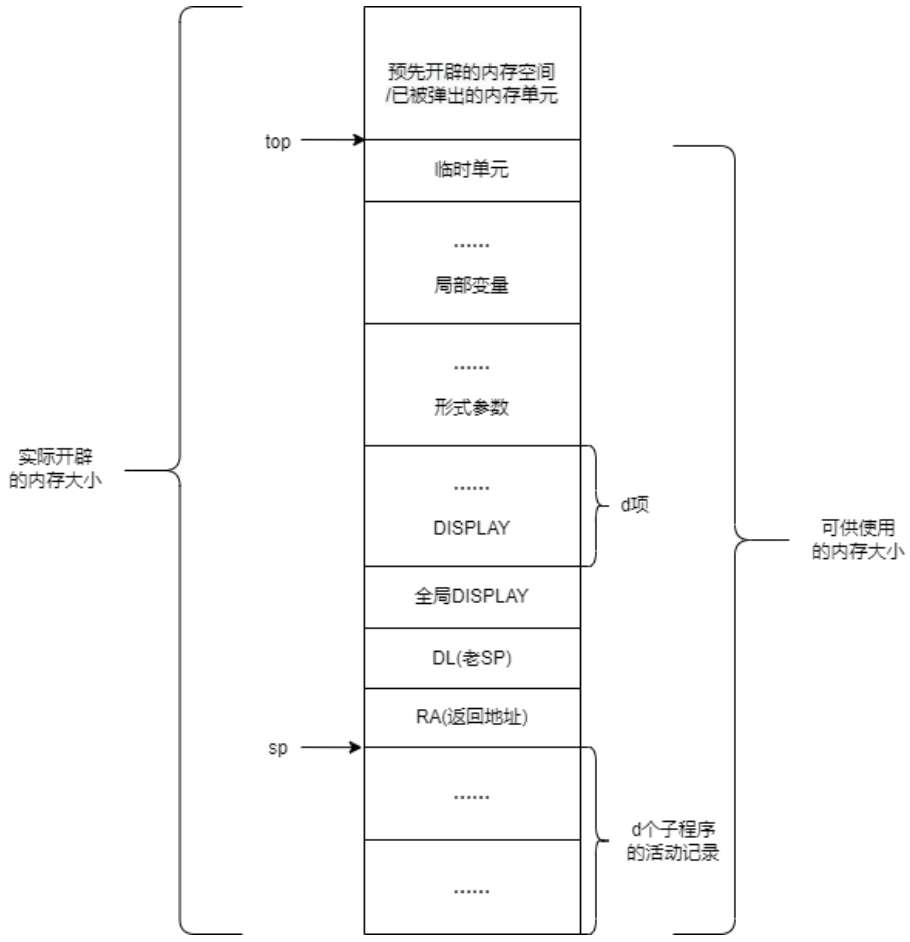


图 2.5 活动记录的格局

一个存储器：

running_stack 是一个数据栈，存储中间代码运行过程中的活动记录。
四个寄存器：
栈顶指示器寄存器 top：指向数据栈 running_stack 的可用内存栈顶 + 1；
内存大小寄存器 size：表示当前内存的实际大小；
基地址寄存器 sp：存放当前运行过程的数据区在 running_stack 中的起始地址；
程序地址寄存器 PC：存放下一条要执行的指令地址；

主要功能：

- **定义活动记录的格局**

  如图 2.5 所示，一个子过程的活动记录被分为如下几个部分：
  RA
  DL
  全局 DISPLAY
  DISPLAY 表
  形式参数
  局部变量
  其中全局 DISPLAY 用于记录主调过程 DISPLAY 表的基地址，当新的子过程的活动记录生成 DISPLAY 表时，也就是调用 call, L, a 时，根据当前全局 DISPLAY 指向的 DISPLAY 表复制 L 项，并将当前子过程的基地址交给 DISPLAY[L]，即可生成当前活动记录的 DISPLAY 表。

- **定义内存的分配方式**

  内存分配采用纯栈式分配方式，根据中间代码指示的偏移量确定不同变量所在的位置。值得一提的是，我实现的数据栈有两种内存，从 0 到 top 范围，是可以直接操作的内存部分，而 0 到 size 范围，则是实际开辟的内存部分，对于 top 以上的部分，是由于调用子过程需要预先为形参开辟的空间，或者弹栈时只是移动指针但没有释放的空间，这部分除了某些特殊的中间代码可以操作，对于其他中间代码来说是不可见的。

- **按正确顺序执行存储单元中的中间代码**

  使用 PC 寄存器控制中间代码的运行流程，开始时自动执行第 0 条中间代码，每执行完一条中间代码后都会修改 PC 的值，指示下一条被执行的中间代码的下标，直到 PC 的值比所有中间代码的下标值大，则停止运行。

- **将中间代码转换为对数据栈的操作**

具体实现：

函数声明	说明
static void lit(Operation op, int L, int a);	取常量 a 放入数据栈栈顶
static void opr(Operation op, int L, int a);	根据 a 的值执行不同的运算，如四则运算，过程返回，条件运算等
static void lod(Operation op, int L, int a);	取层级 L 相对地址为 a 的变量放到数据栈的栈顶
static void sto(Operation op, int L, int a);	将数据栈栈顶的内容存入层级为 L，相对地址为 a 处
static void cal(Operation op, int L, int a);	调用入口地址为 a，层级为 L 的子过程
static void alc(Operation op, int L, int a);	为数据栈的可用内存开辟 a 个单元，即 top

	指针增加 a
static void jmp(Operation op, int L, int a);	无条件转移到地址 a
static void jpc(Operation op, int L, int a);	栈顶条件为假时转移到地址 a
static void red(Operation op, int L, int a);	从控制台读一个数据并存入数据栈栈顶
static void wrt(Operation op, int L, int a);	将数据栈栈顶内容写入控制台
static void run();	开始运行中间代码
static void clear();	清空运行数据栈

流程图：详情见附件 4.3.1

3. 课设总结

这次课设可以算是我大学生涯以来最特殊也最难忘的一次课设了。

为何特殊？由于新冠疫情防控突然放开，疫情肆虐，安排在学期末的本课设验收不得不延迟至寒假以后，因此这成了我第一次在假期中完成的课设。

为何难忘？因为这次课设不仅让我收获了许多专业知识，更令我的整个寒假变得焕然一新。那些如同着魔一般，每天睁眼闭眼全在思考如何写编译器的日子，疲惫不堪，却充实而快乐。

谈谈我的收获吧：

1. 搞懂了字符编码的问题

为了实现编译器对中文非法字符的识别，我第一次尝试正式地学习字符编码的问题。在此之前，我对字符编码的印象还停留在英文字符占 1 字节，中文字符占 2 字节，后来才发现自己是大错特错，正如这篇博文[3]所说的：**“It does not make sense to have a string without knowing what encoding it uses.”**，不知道字符编码就使用字符串是毫无意义的。读完这篇博文后，我真正了解了 Ascii、Unicode、UTF-8、UCS-2 之间的区别和联系，并由此统一了编译器内所有使用到的字符串为 UCS-2 编码格式。关于不同高级语言的字符编码问题，JAVA 的 char 字符默认为 UTF-16 格式，但 C++的 char 本质上就是一个字节，不带任何编码格式信息，因此编码问题需要交给程序员处理。

2. 了解了位掩码的妙用

关于使用位掩码表示同步符号的想法是我从这篇博文[2]中看到的，起初我还不以为然，只用一个二进制位表示不同的同步符号，其他位全部为‘0’，这是否过于浪费存储空间了？后来当我犹疑不决如何确定 first、follow 集的存储，才发现了位掩码的妙用，既能便捷地构建元素的集合(位或操作)，又能快速高效判断元素与集合的从属关系(位与操作)。我也不得不感慨位运算的强大和奇妙。

3. 深刻了解了编译器的工作原理

在手搓了一个编译器之后，许多的疑问都豁然开朗。

比如，为什么要强调初始化的问题，拿我自己的编译器举例，因为数据栈在执行弹栈操作时只是将栈顶指针移动，并未真正释放栈顶的空间，所以若之后再次使用到未释

放的内存，该内存单元中的值就是一个随机的值，所以初始化的重要性不言而喻。

还有，为什么深层的递归程序会导致爆栈，这是因为每次递归都会产生一个过程的活动记录，若递归程序没有到达终止条件，过程的活动记录就不会弹出，只进不出，内存栈无法容纳这么多活动记录，就产生了爆栈的错误。

此外，还有为什么右值不能取址的问题，对于本编译器，右值有两种情况，表达式求值的中间结果，常量值。对于第一种情况，它只会在数据栈的临时单元区中出现，使用完之后就会被销毁，生命周期十分短暂，对它取址是不现实的；对于第二种情况，编译器在编译过程会中将常量值存储在符号表中，并在需要的地方将其直接替换为相应的值，因此它根本不会出现在内存中，取址也就无从谈起。

类似的问题还有很多，比如为什么空白符号不会影响编译器的工作，编译器为什么要分为前端和后端，为什么局部变量可以覆盖同名全局变量等等，在手搓了编译器之后都有了清晰的认识。

4. 学会了编译环境的配置

我之前写 C++代码全都是在 VSCode 上写的，尽管只是一个编辑器，但是它小巧而优雅，插件强大，不像 Visual Studio 那样笨重，启动都要半天。但是 VSCode 最大的缺陷，就是不会默认配置编译环境，不仅需要自己下载编译器、链接器、还要手动配置启动命令，为了解决这些问题，我可谓是苦不堪言。

首先是选择编译器的问题，尽管我已经有现成的编译器 MinGW32 可用，但是这款编译器太旧，也现在没有人维护更新，使用它的调试器也不能直观的调试 STL，因此我更换为了 MinGW-w64。但其代码提示功能还有所欠缺，经朋友介绍，我将用于代码提示的编译器更换为 clang++，代码提示可以显示结构体的 padding，也能方便地查看源码。

然后是多文件编译的问题。VSCode 中没有配置好的链接器，MinGW-w64 的默认链接器会使用 MSVC 的，多文件编译时就会报许多莫名奇妙的错误，更换为 MingGW-w64 的链接器就正常了。由于当时我对链接器，编译器的理解并不透彻，所以遇到了许多困难。

5. 学会使用版本控制工具

本编译器的代码我全部上传至了 Github，不仅可以方便地回溯历史，不需要担心代码重构出现问题无法恢复，也方便了我与其他同学的沟通交流。如今浏览密密麻麻的 commit 列表，那种从无到有，平底起高楼的成就感也是油然而生。

其他的收获还有很多，比如了解了查阅英文资料、使用英文检索的重要性，写注释的重要性等，限于篇幅也就不再细说。

最后，我想感谢一下潘鹏飞和王筱瑀同学，在我写编译器期间，我和他们进行了许多技术上的交流，他们给了我很多帮助与启发，我编译器的完成离不开他们的帮助。感谢谢强老师验收时的耐心指导，帮我找出了符号表查找和参数传递的错误。希望自己再接再厉，努力提升自己的代码水平，争取完成更加有水平的项目。

4. 附件

4.1 测试文件说明

文件名	说明
test_error	本文件用于测试几乎所有编译器能识别出的错误类型，如非法字符，缺少符号，符号冗余，重定义错误，右值赋值错误等
test_fib	本文件实现了递归的斐波那契数列的计算，输入为 n，输出为 Fibonacci(0)到 Fibonacci(n)的所有值。为了检测参数传递是否正确，我为计算 Fibonacci(x)的过程添加了几个无效的形参
test_proc_call	本文件用于测试嵌套过程的调用，定义了变量 n，每调用一次子过程 n-1，根据 n 的最终值可知道进行了几次过程调用
test_proc_redef	本文件定义了一些重复的过程，用于测试哪些过程会提示重定义错误
test_sum	本文件实现了求和函数，提供输入 n，将会计算出 1 到 n 的所有整数和

4.2 PL0 文法的翻译模式

说明：下面的翻译模式中的属性不是和代码一一对应的，比如左边终结符的继承属性，我是采用符号表记录的，是对代码的简化表示，用于展示翻译的基本思路。

全局属性：

```
glo_offset      // 全局偏移量，单位是 Byte
nextquad        // 下一条将产生的中间代码入口
strToken        // 最近一次分析出的词法单元字符串
level           // 当前分析的子过程的层级
sp              // 当前正在分析的子过程的入口地址
```

<prog> → program <id>{

    mkTable()

    enterProgm(strToken)

    block.entry = emit(JMP, 0, \) // 为 block 的继承属性赋值 }

;<block>

<block> → [<condecl>][<vardecl>]{

    addWidth(sp, glo_offset) // 子过程变量声明结束，计算过程占用内存

    block.cur_proc = sp }

[<proc>]{

    body.entry = emit(INT, 0, block.cur_proc->width)

    backpatch(block.entry, body.entry) // 将过程体的中间代码入口地址回填，block.entry 是已经定义的继承属性

block.cur_proc->isDefined = true // 即将进入过程体，标识改为过程体已定义 }
<body>

<condecl> → const <const>{,<const>};

<const> → <id>{ id.sym_entry = enter(strToken, 0, CST) } // 右值不占用内存
:=<integer>{ setValue(id.sym_entry, integer) } // 右值直接登入符号表

<vardecl> → var <id>{
enter(strToken, glo_offset, VAR)
glo_offset += 4 }
{,<id>{
enter(strToken, glo_offset, VAR)
glo_offset += 4 };

<proc> → procedure <id1>{
mkTable()
id1.sym_entry = enterProc(strToken)
block.entry = emit(JMP, 0, \) } // 为 block 的继承属性赋值
([<id2>{
id2.sym_entry = enter(strToken, glo_offset, VAR)
glo_offset += 4
id1.sym_entry->form_var_list.add(id2.sym_entry) } // 在符号表中为过程添加形参入口
{,<id3>{
enter(strToken, glo_offset, VAR)
glo_offset += 4
id1.sym_entry->form_var_list.add(id3.sym_entry) }])
;<block>{
emit(OPR, 0, OPR_RETURN)
display.pop()// 子过程结束，层级减一，display 表弹栈
level-- }
{;<proc>}

<body> → begin <statement>{;<statement>}end

<statement> → <id>{
id.sym_entry = lookUpVar(strToken)
if (id.sym_entry == -1) error }
:= <exp> {
if(id.sym_entry->cat!=CST) // 常量不可被赋值
emit(STO,id.sym_entry->level, id.place) }

|if <lexp> then { lexp.false_entry = emit(JPC, 0, \) } // 条件为假时,跳转到 else 处或 if 语句外
<statement1>[else <N> <statement2>{ backpatch(N.entry, nextquad) }]

{// 如果没有 else 才执行下面的翻译模式
backpatch(lexp.false_entry, nexquad) }// 将 if 语句外的入口地址回填至 JPC

|while <M><lexp>{ lexp.false_entry = emit(JPC, 0, \)} do <statement>{
emit(JMP, 0, M.quad)
backpatch(lexp.false_entry, nextquad) }

|call <id>{
id.sym_entry = lookUpProc(strToken)
if (id.sym_entry == -1) error }
([<exp1>{
i = 0
emit(STO, -1, 3 + id.sym_entry->level + 1 + i) }
{,<exp2>{
i += 1
emit(STO, -1, 3 + id.sym_entry->level + 1 + i) }])

|read (<id1>{
id1.sym_entry = lookUpVar(strToken)
if (id1.sym_entry == -1) error
if(id1.sym_entry->cat != CST)
begin
emit(RED, 0, 0)
emit(STO, id1.sym_entry->level, id1.place)
end }
{,<id2>{
id2.sym_entry = lookUpVar(strToken)
if (id2.sym_entry == -1) error
if(id2.sym_entry->cat != CST)
begin
emit(RED, 0, 0)
emit(STO,id2.sym_entry->level, id2.place)
end } })

|write (<exp>{ emit(WRT, 0, 0) }{,<exp>}{ emit(WRT, 0, 0) })
|<body>

<M>→ε{ M.quad = nextquad }

<N>→ε{
N.entry = emit(JMP, 0 ,\) // 无条件跳转语句，跳转至 if 语句外
backpatch(lexp.false_entry, nextquad) } // 将 else 入口地址回填至 JPC

<lexp> → <exp> <lop> <exp>{

```
        switch(lop):
        case '<': emit(OPR, 0, OPR_LSS); break;
        case '<=': emit(OPR, 0, OPR_LEQ); break;
        case '>': emit(OPR, 0, OPR_GRT); break;
        case '>=': emit(OPR, 0, OPR_GEQ); break;
        case '<>': emit(OPR, 0, OPR_NEQ); break;
        case '=': emit(OPR, 0, OPR_EQL); break;
        defalut: break; }
| odd <exp>{ emit(OPR, 0 ,OPR_ODD) }
```

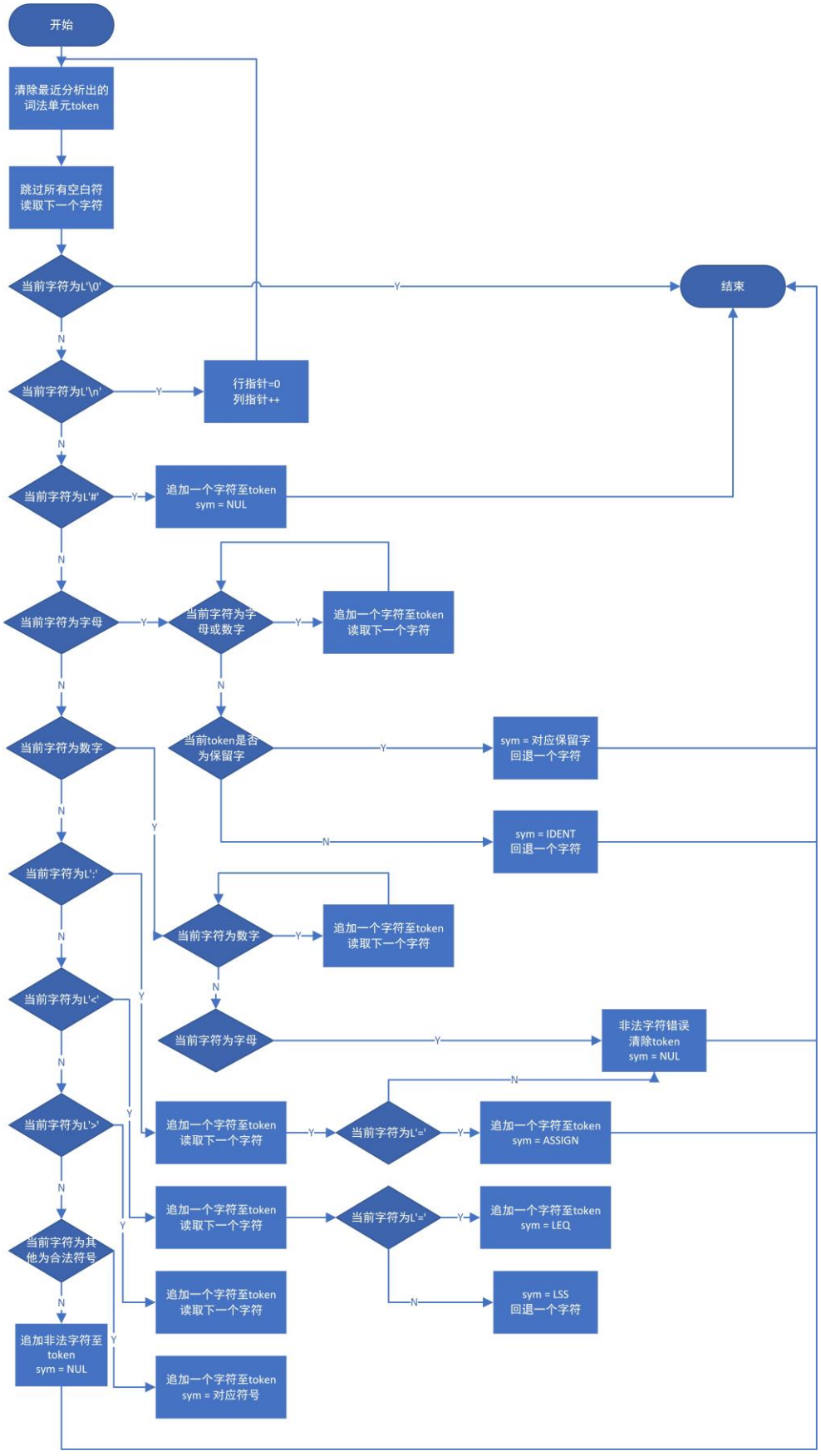
```
<exp> → <aop1>{
        if(aop1=='-')   aop1.flag = 1;
        else    aop1.flag = 0; }
<term>{
        if(aop1.flag == 1)      emit(OPR, 0 , OPR_NEGTIVE) } // 第一个 aop 为负号,产生取反代码
{<aop2> {
        if(aop2=='-')   aop2.flag = 1;
        else    aop2.flag = 0; }
<term>{
        if(aop2.flag == 1) // 根据加减号产生相应代码
                emit(OPR, 0, OPR_SUB);
        else
                emit(OPR, 0 ,OPR_ADD); }}
```

```
<term> → <factor>{<mop>{
        if(mop=='*')    mop.flag = 0;
        else    mop.flag = 1; }
<factor>{
        if(mop.flag == 0)  // 根据乘除号产生相应代码
                emit(OPR, 0, OPR_MULTI);
        else
                emit(OPR, 0 ,OPR_DIVIS); }}
```

```
<factor>→<id>{
        id.sym_entry = lookUpVar(strToken)
        if (id.sym_entry == -1) error
        if( id.sym_entry->cat == CST) // 读取到常量符号，产生 LIT 代码，否则为变量，产生
LOD
                emit(LIT, id.sym_entry->level, id.sym_entry->value)
        else
                emit(LOD, id.sym_entry->level, id.place) }
|<integer>{ emit(LIT, 0, w_str2int(strToken)) }
|(<exp>)
```

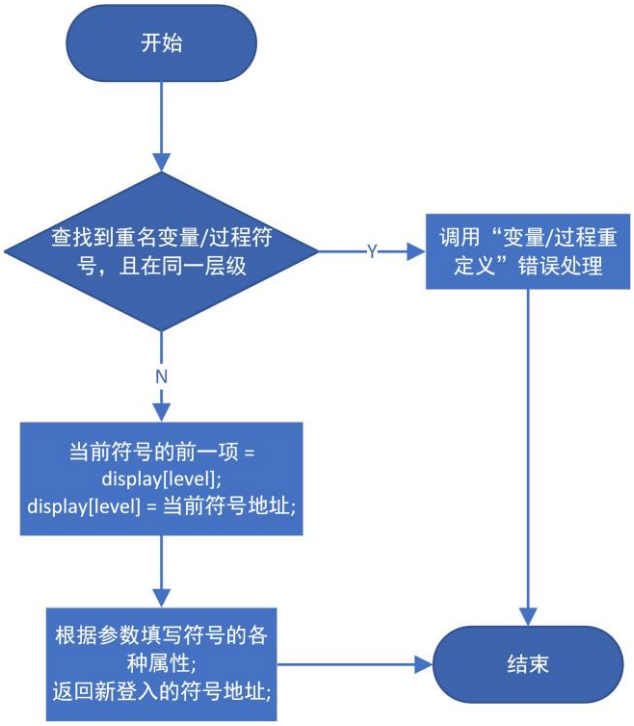
4.3 程序流程图

4.3.1 词法分析

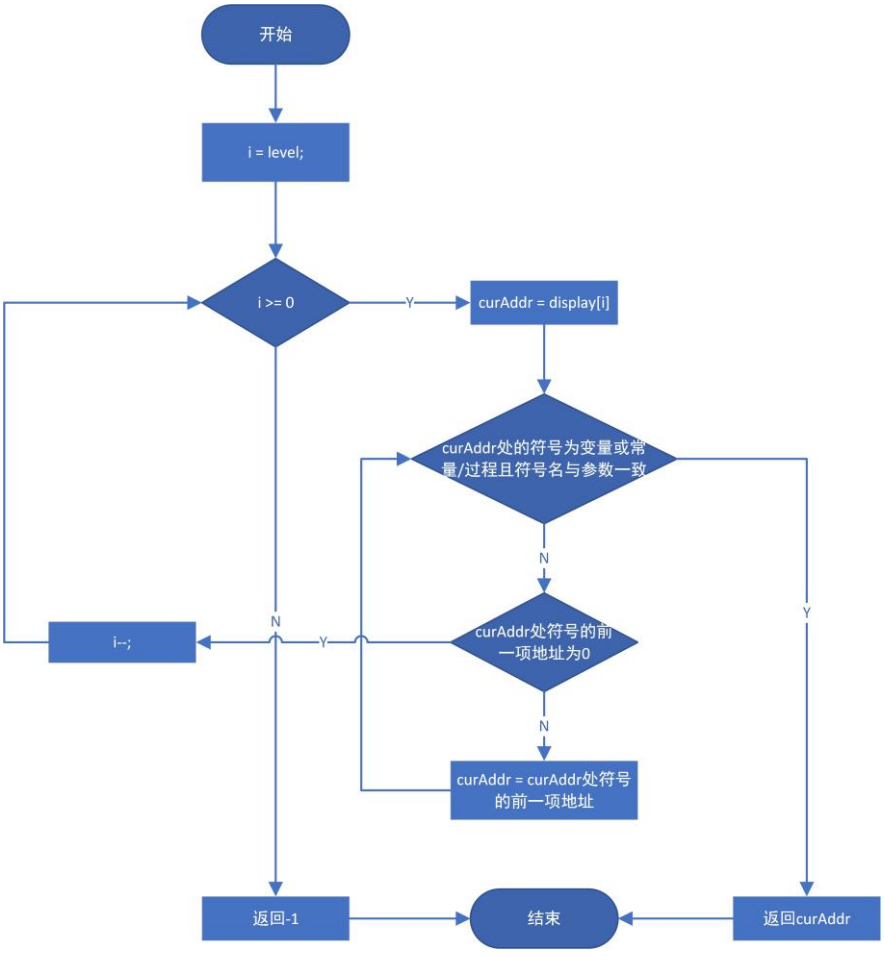


4.3.2 符号表

enter/enterProc：

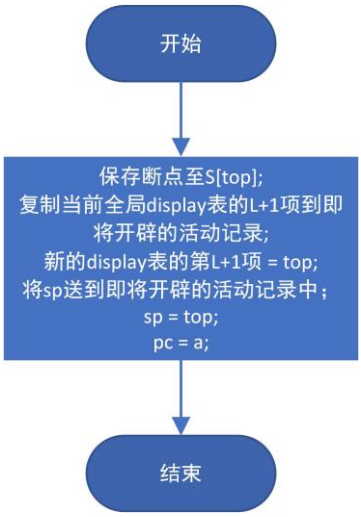


lookUpVar/lookUpProc：

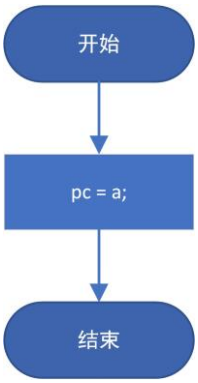


4.3.3 P 代码

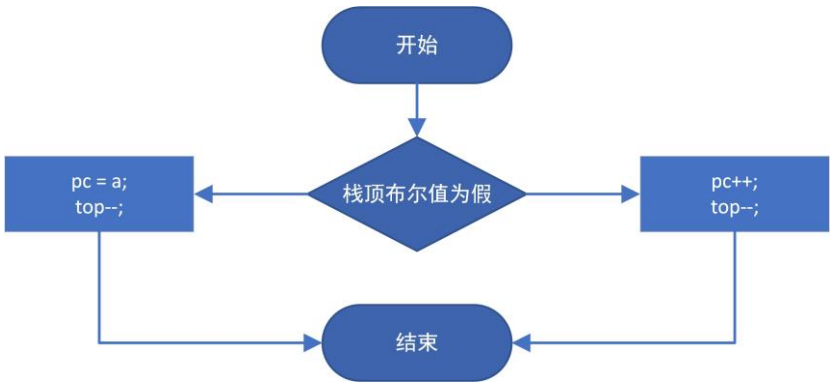
CAL，L，a：



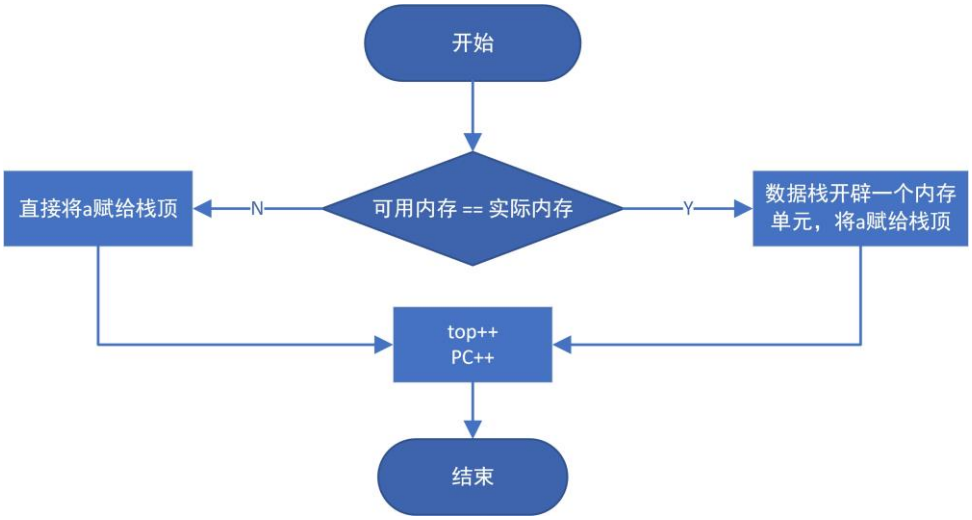
JMP，0，a：



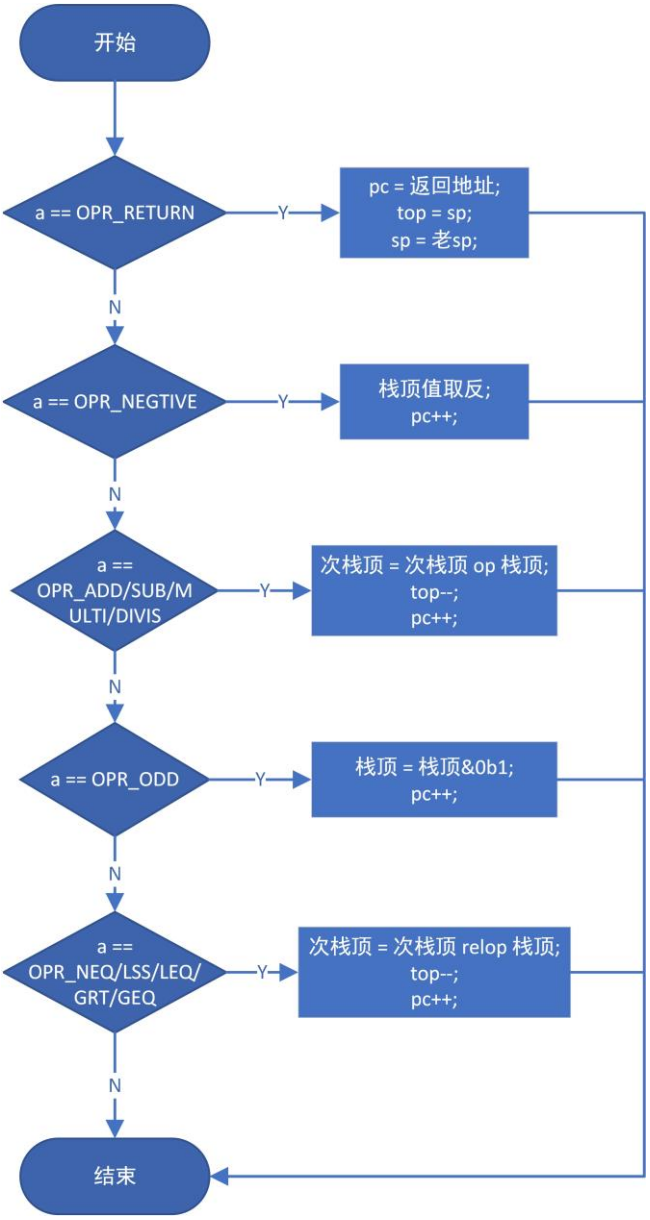
JPC，0，a：



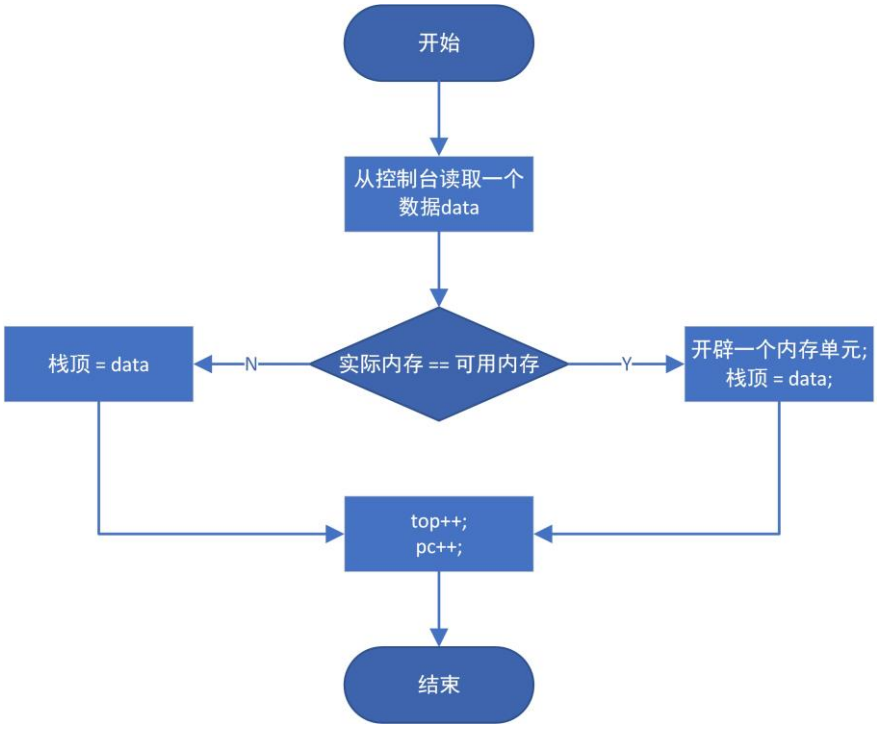
LIT，0，a：


开始
可用内存 == 实际内存
N
Y
直接将a赋给栈顶
数据栈开辟一个内存单元，将a赋给栈顶
top++
PC++
结束

OPR，0，a：

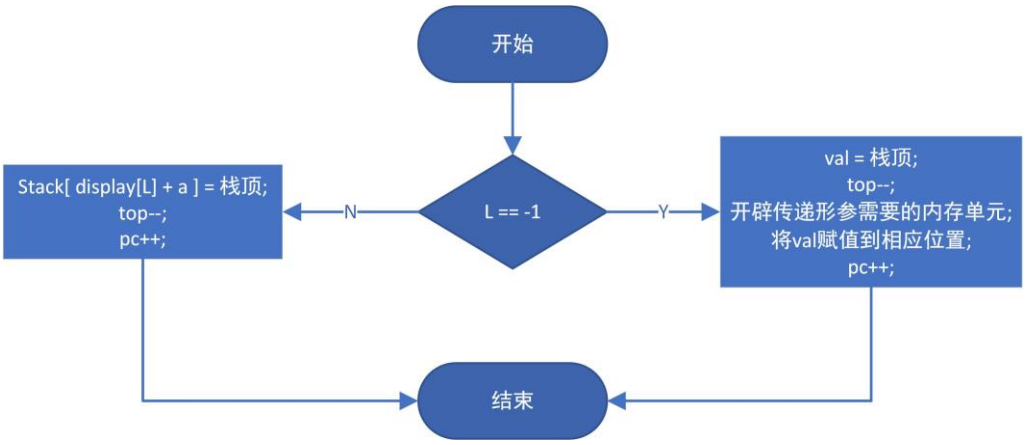

开始
a == OPR_RETURN
Y
pc = 返回地址;
top = sp;
sp = 老sp;
N
a == OPR_NEGTIVE
Y
栈顶值取反;
pc++;
N
a == OPR_ADD/SUB/MULTI/DIVIS
Y
次栈顶 = 次栈顶 op 栈顶;
top--;
pc++;
N
a == OPR_ODD
Y
栈顶 = 栈顶&0b1;
pc++;
N
a == OPR_NEQ/LSS/LEQ/GRT/GEQ
Y
次栈顶 = 次栈顶 relop 栈顶;
top--;
pc++;
N
结束

RED，0，0：

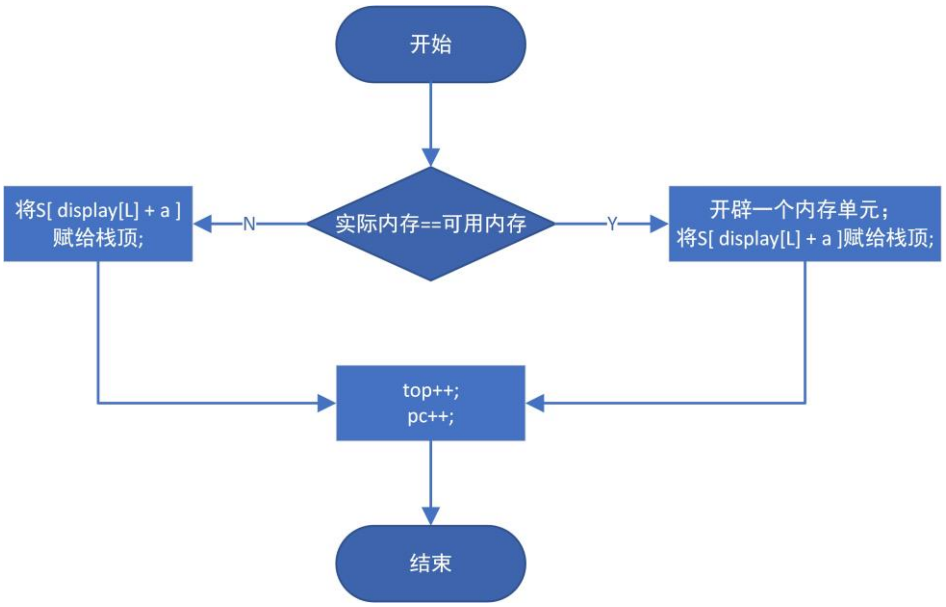


WRT，0，0：

STO，L，a：



LOD，L，a：



5. 参考资料

[1] 《程序设计语言编译原理（第 3 版）》 陈火旺等

[2] 编译原理课设尝试（一）——PL/0 编译器分析 | Chenfan Blog (jcf94.com)

[3] The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) OCTOBER 8, 2003 by JOEL SPOLSKY

[4] StackOverflow：Printing unicode characters in PowerShell via a C++ program

[5] C++ 读取 UTF8 文件_stevenldj 的博客-CSDN 博客_c++ utf8

[6] C++ 字符串 string、字符 char、宽字符数组 wstring、宽字符 wchar_t 互相转换（2021.4.20）_jing_zhong 的博客-CSDN 博客_string 怎么转换 wchar_t

[7] 【转】utf-8 的中文是一个汉字占三个字节长度_weixin_34117211 的博客-CSDN 博客

[8] 实战中遇到的 C++流文件重置的一个大陷阱： 为什么 ifstream 的 seekg 函数无效？ 涛歌依旧的博客-CSDN 博客_c++ seekg 不起效

[9] 下载安装 MinGW-w64 详细步骤（c/c++的编译器 gcc 的 windows 版，win10 真实可用）_jjxcsdn 的博客-CSDN 博客_mingw-w64