

Document ID: FFAL2000D001

## 控制器中的解析器方案

项目编号 #		项目名称	
标题			
保密等级			
作者		日期	

评审记录:

姓名	职位	签名/日期

## 目录

1. 文档基本信息.....	4
1.1. 文档状态.....	4
1.2. 修改记录.....	4
2. 简单介绍.....	5
2.1. 涵盖范围.....	5
2.2. 对谁可见.....	5
2.3. 项目信息.....	5
2.4. 缩写和简称.....	5
2.5. 相关文档.....	5
3. TP、控制器程序下发及解析框架.....	6
4. 程序文件格式转换流程.....	6
5. 如何跳行.....	8
6. XML 文件翻译 .....	10
7. TP 的行号信息.....	12
8. 基于共享内存机制的解析器与控制器的交互.....	13
命令区.....	13
状态区.....	14
寄存器区.....	15
IO 区.....	15
9. 解析器状态机.....	17
10. 解析程序结构.....	18
11. 表达式分析器流程图.....	18
12. 表达式分析器对于数组的支持.....	20
13. 表达式分析器对于数学函数的支持.....	21
14. token 取词器流程图.....	22
15. token 和 tok .....	24
16. 符号表.....	24
17. 双运算符的实现.....	25
18. 程序总体流程图.....	25
19. 扫描文件中的标号和代码行信息流程图.....	27
20. 程序命令 IF 和 ELSEIF 解析流程图.....	27
21. 程序命令 while 解析流程图.....	29
22. 程序命令 wend 解析流程图.....	30
23. 程序命令 break 解析流程图.....	32
24. 程序命令 continue 解析流程图.....	33
25. 程序命令 call/gosub 解析流程图 .....	34
26. 程序命令 return 解析流程图.....	34
27. 程序命令 end 解析流程图.....	35
28. 程序命令 select 解析流程图 .....	36
29. 程序命令 case 解析流程图.....	36

30.	赋值语句流程图.....	37
31.	MOV 相关命令解析流程图 .....	39
32.	MOV 相关命令扩展参数解析流程图.....	40
33.	Timer 命令解析流程图 .....	40
34.	UserAlarm 命令解析流程图.....	41
35.	Wait 命令解析流程图 .....	42
36.	Pause 命令解析流程图 .....	43
37.	Abort 命令解析流程图 .....	43

# 1. 文档基本信息

## 1.1. 文档状态

状态	描述
目前状态	制定中，不可用
完成	
通过评审	

## 1.2 修改记录

文档修改记录			
版本号	日期 yyyy-mm-dd	作者	修改描述
FFAL2000D001	2018-02-02	卢佳明	初稿
FFAL2000D001	2018-02-07	卢佳明	添加共享内存机制，补充一些细节。
FFAL2000D001	2018-02-14	卢佳明	根据评审记录修改

## 2. 简单介绍

### 2.1. 涵盖范围

本文描述了控制器中的解析器的方案。

### 2.2. 对谁可见

系统软件组工程师

### 2.3. 项目信息

#	条目	信息	描述
1			
2			
3			

### 2.4. 缩写和简称

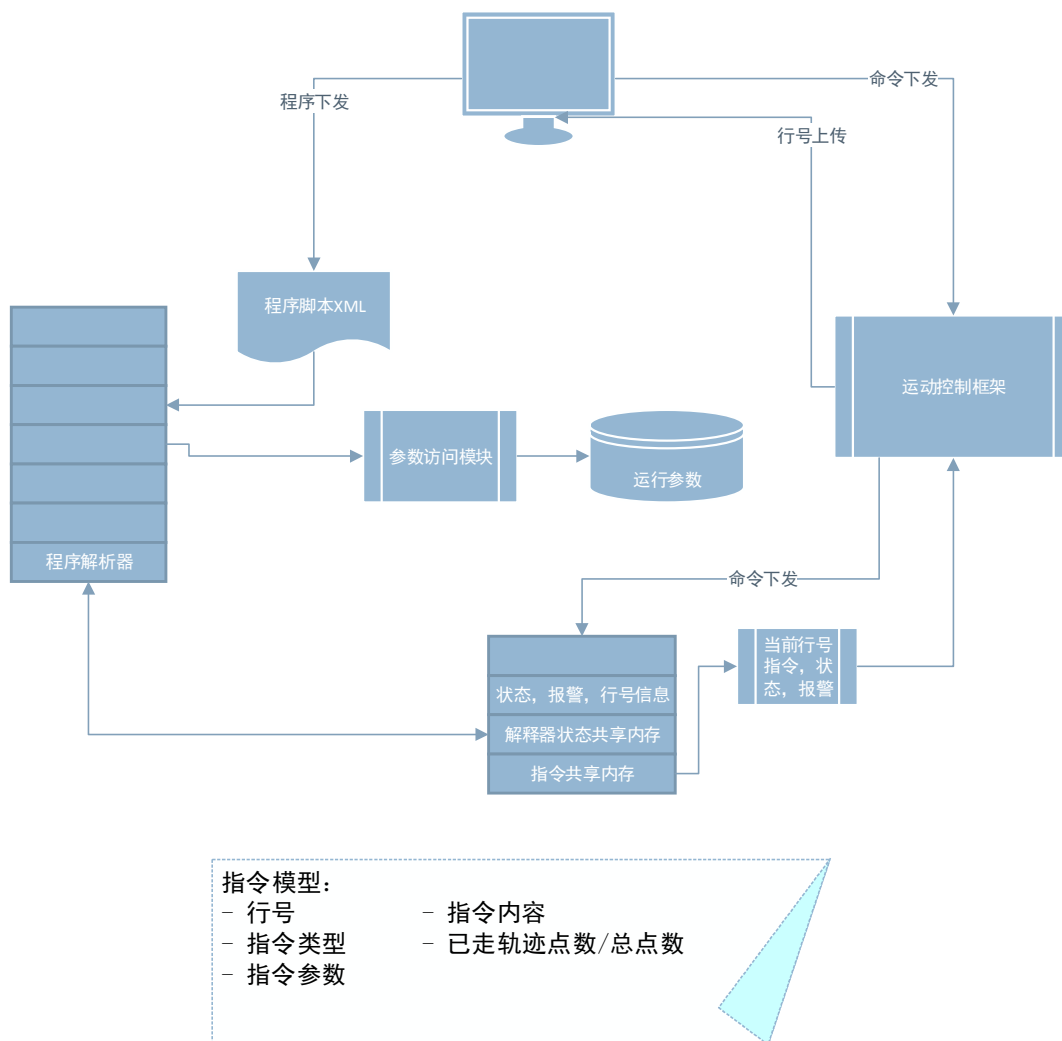
缩写和简称	
token	指的是解析器中的一个单词。是解析器解析的最小单元。

### 2.5. 相关文档

#	文档名称	版本	文档存储地址	描述

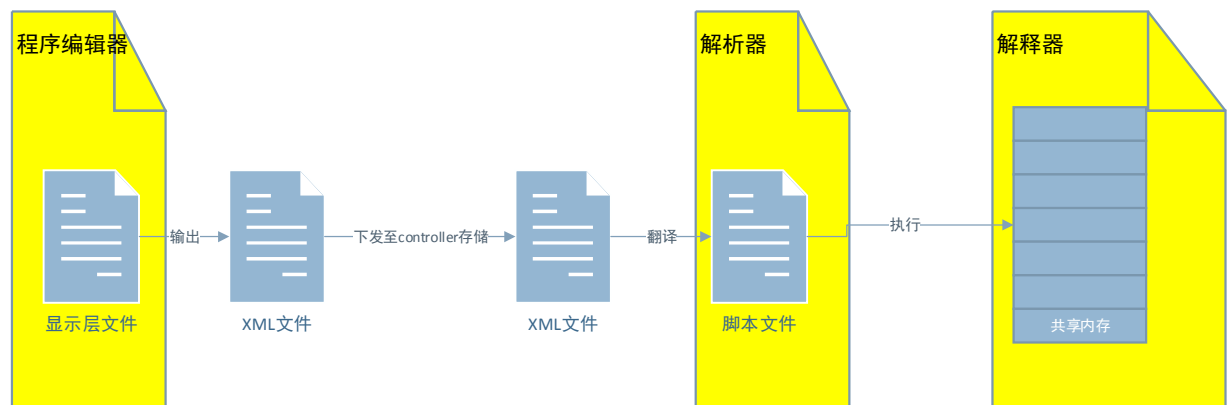
### 3. TP、控制器程序下发及解析框架

控制器主控在收到 TP 的程序执行相关指令的时候，把指令发给解析器。  
解析器根据指令执行保存在控制器上的程序，对于运动相关指令插入共享内存执行。  
对于寄存器操作，操作寄存器共享内存读写。  
在执行过程中，更新行号。和解释器的状态。

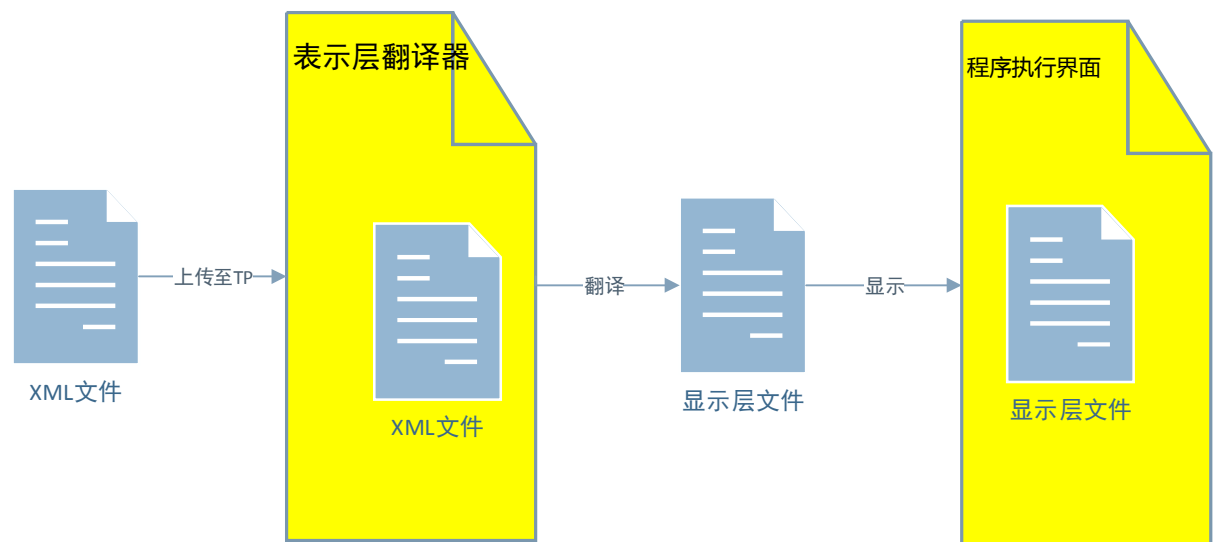


### 4. 程序文件格式转换流程

## 程序下发解析流程：



## 程序上传显示流程：



## 5. 如何跳行

首先脚本代码样例如下：

```

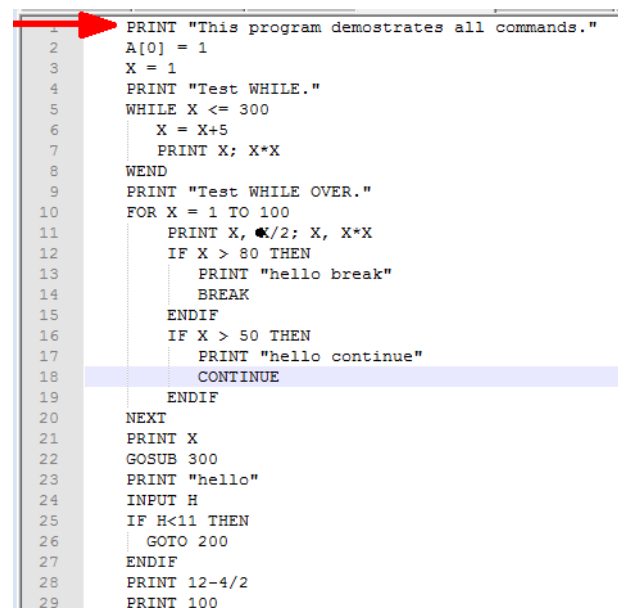
1  PRINT "This program demonstrates all commands."
2  A[0] = 1
3  X = 1
4  PRINT "Test WHILE."
5  WHILE X <= 300
6      X = X+5
7      PRINT X; X*X
8  WEND
9  PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN
13         PRINT "hello break"
14         BREAK
15     ENDIF
16     IF X > 50 THEN
17         PRINT "hello continue"
18         CONTINUE
19     ENDIF
20 NEXT X
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26     GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100

```

可以看到这个脚本代码是一行一行的字符串。

跳行的方法如下：

1. 读入脚本代码。使用一个指针指向代码字符串。表示从哪里开始解析。



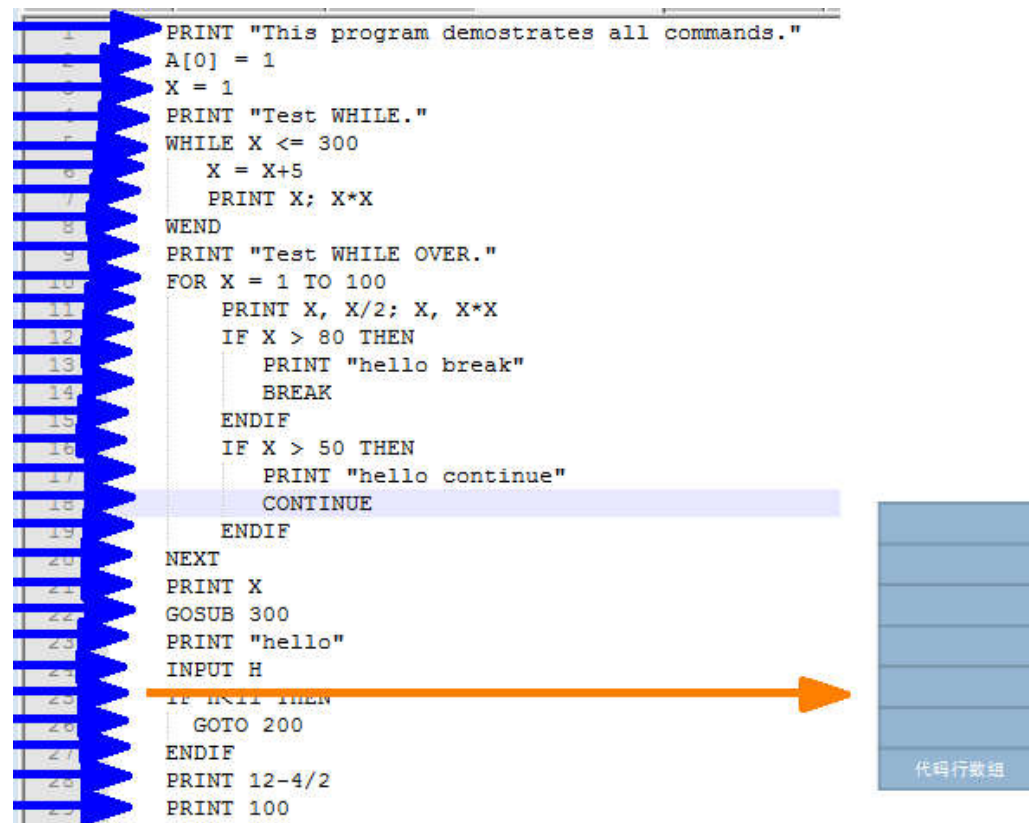
```

1  PRINT "This program demonstrates all commands."
2  A[0] = 1
3  X = 1
4  PRINT "Test WHILE."
5  WHILE X <= 300
6      X = X+5
7      PRINT X; X*X
8  WEND
9  PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN
13         PRINT "hello break"
14         BREAK
15     ENDIF
16     IF X > 50 THEN
17         PRINT "hello continue"
18         CONTINUE
19     ENDIF
20 NEXT X
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26     GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100

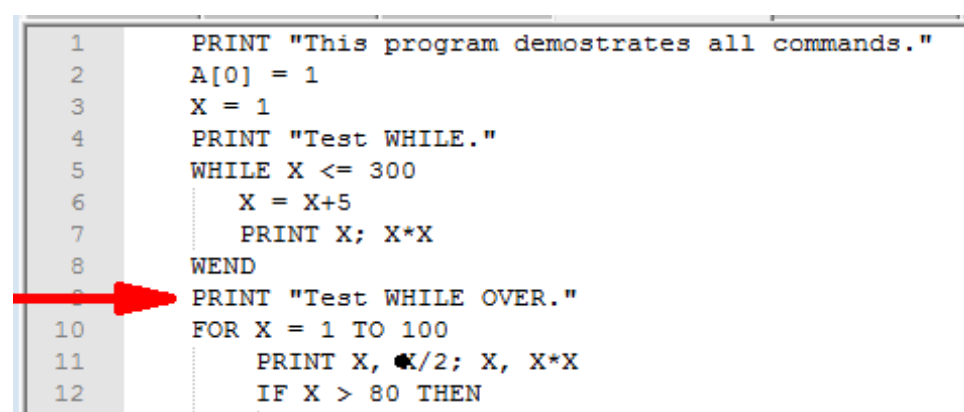
```



2. 之后遍历一遍这个代码字符串。记下每一行的起始位置。



3. 在单步模式下，解释器会每次解释代码字符串指针指向的一行。



4. 解释一行完成之后，等待用户输入行号。

5. 根据行号，取出每一行的起始位置。这里假设输入 4。

6. 代码字符串指针指向这个位置。再次调用解释器。

```

1  PRINT "This program demonstrates all commands."
2  A[0] = 1
3  X = 1
4  PRINT "Test WHILE."
5  WHILE X <= 300
6      X = X+5
7      PRINT X; X*X
8  WEND
9  PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN
13         PRINT "hello break"
14         BREAK
15     ENDIF
16     IF X > 50 THEN
17         PRINT "hello continue"
18         CONTINUE
19     ENDIF
20 NEXT
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26     GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100

```

## 6. XML 文件翻译

翻译程序使用 libxml2 库使用 DOM 遍历 XML 的每一个节点，生成 BASIC 格式的程序文件。

XML 文件格式参见《#1454-用户程序存储格式设计方案》

根据需求，每一个文件都是一个子程序。在设计上，实现为定义在文件中的一个名字叫 main 的函数。也就是文件支持包含多个函数。默认执行 main 函数。因为 libxml2 库使用 DOM 将 XML 解释为一棵树。

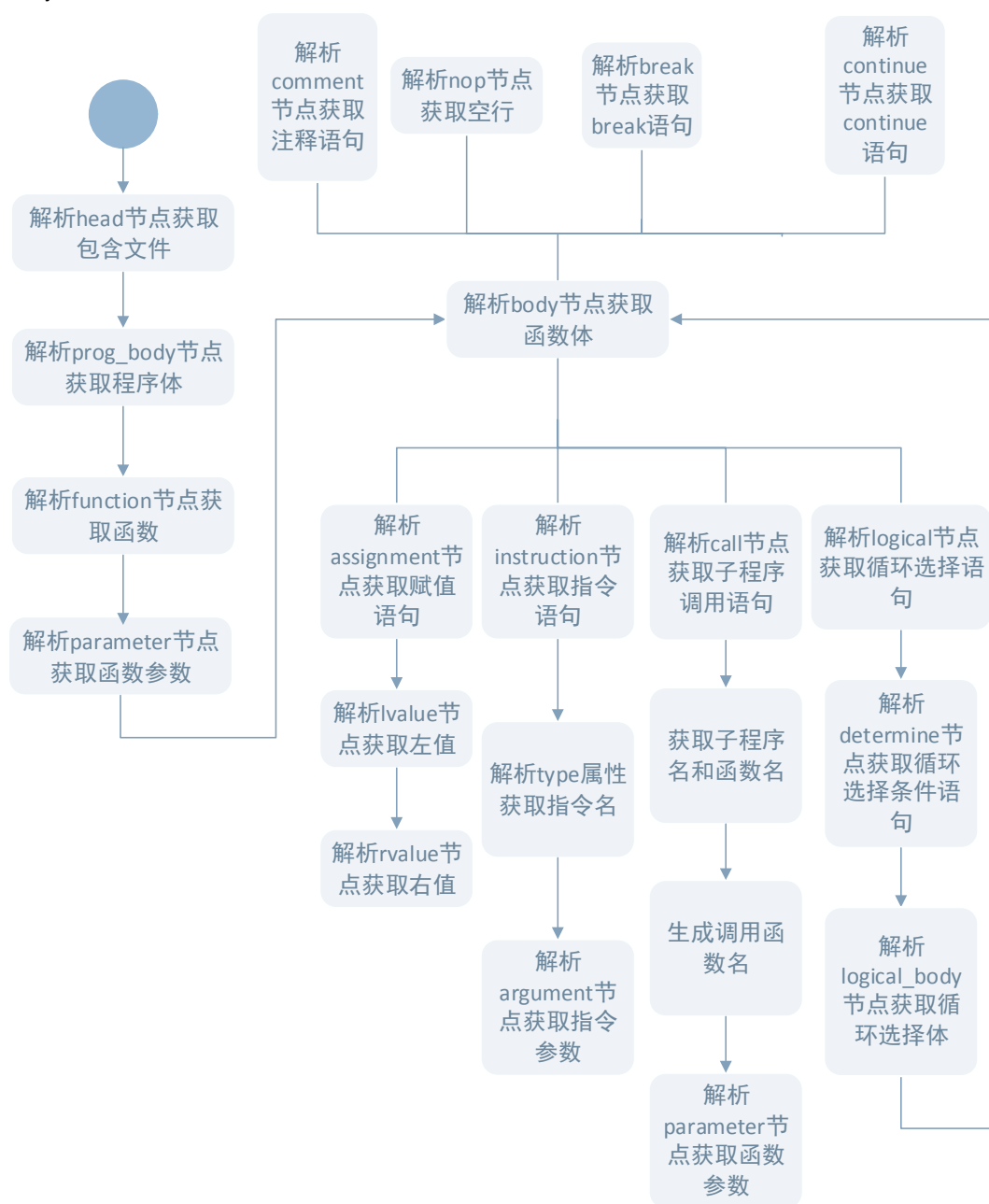
整体流程其实就是这个 DOM 树的深度遍历。流程如下：

1. 调用 xmlDocGetRootElement 获取根节点，返回一个 xmlNodePtr 对象。
2. 遍历根节点的子节点。也就是根节点 xmlNodePtr 对象的 children 指针。
3. 对于得到的子节点，也是一个 xmlNodePtr 对象，再次遍历这个节点 xmlNodePtr 对象的子节点。



4. 每一个节点都有 **name 成员**。并可以根据**属性名**调用 xmlGetProp 获得属性值。调用 xmlNodeGetContent 获得**值**。

```
<argument name="velocity" type="num" unit="mm/s">30</argument>
```

```
xmlNodePtr rootProg = xmlDocGetRootElement(doc);
for(xmlNodePtr nodeHead = rootProg->children;
    nodeHead; nodeHead = nodeHead->next){
    ....
}
```



对于指令节点，参数节点顺序，和《Rtimeman Robot\_核心系统\_用户指令（PartI 语法格式）\_需求文档\_V1.0》中保持一致。这一点由《#1454-用户程序存储格式设计方案》保证。

XML 文件格式如下：	翻译后的程序文件如下：
 prog_demo_dec.xml	 prog_demo_dec.bas.txt

## 7. TP 的行号信息

在生成每一行语句的时候，调用 `xmlGetNodePath` 获取这个节点对应的 XPath 作为行号信息。

代码如下：

```

21  R[1] = 5
22  IF R[1] + sin( 2 ) == 100  THEN
23      MOVEL 1.1 2.2 3.3 4.4 5.5 6.6 100 SV 50 ;ACC 40
24  ELSEIF R[1] == 1  THEN
25      WAIT COND DI[0] = ON 10 skip
26  ENDIF
    
```

代码对应的行号信息如下：

```

021:/prog/prog_body
022:/prog/prog_body/function[1]/body/logical[1]
023:/prog/prog_body/function[1]/body/logical[1]/logical_body/instruction
024:/prog/prog_body/function[1]/body/logical[1]/vice_logical
025:/prog/prog_body/function[1]/body/logical[1]/vice_logical/vice_logical_body/instruction
026:/prog/prog_body/function[1]/body/logical[1]
    
```

可以看到，这里的 026 行是 `endif`。不是运动指令。在实际执行过程中是不会上报的。因为 TP 过来的 XML 是一个类似于这样的结构。

```

<logical type="if">
    <determine> . . . </determine>
    <logical_body> . . . . </logical_body>
</logical>
    
```

026行的endif是我自己生成的。

## 8. 基于共享内存机制的解析器与控制器的交互

解析器和控制器通过共享内存进行交互。共享内存分四个部分：命令区、状态区、寄存器区和 IO 区。

命令区是命令交换的媒介；状态区是发送给对方的、并需要对方知道的、自己的状态；寄存器区是存放寄存器的区域；IO区是存放当前IO的区域。



图 1 解析器和控制器之间的共享内存

### 命令区

命令区是解析器（Interpreter）和控制器（Controller）交换命令的区域。分两个部分：一个部分是 ITC，解析器给控制器的命令；另外一个部分是 CTI，控制器给解析器的命令。

这两个部分有共同的属性：1、读和写是互斥的。读完才能写，写完才能读；2、传输是单向的。一方只能读，另一方只能写；3、不能连续读，不能连续写。一方读完一次后必须等待另一方写后才能再读第二次，一方写一次后必须等另一方读后才能写第二次；4、一次只能传输一条命令。

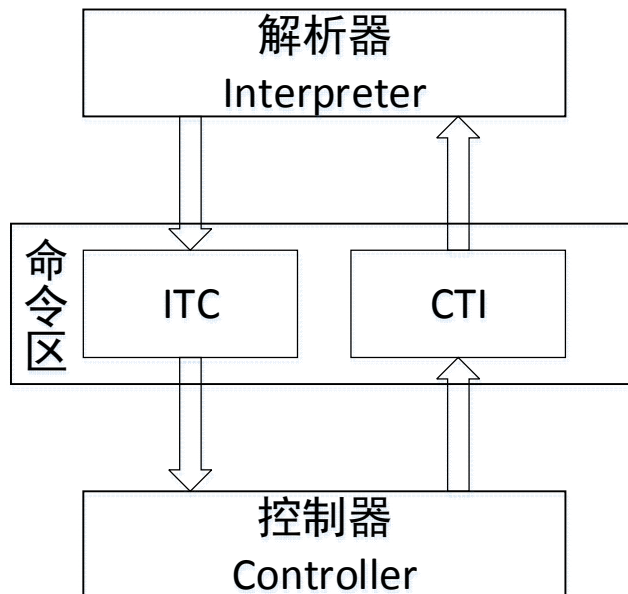


图 2 命令区

## 状态区

状态区是存放各自状态的区域，这些状态需要对方知道。分两部分：一部分是控制器的状态 CStat，这个状态是控制器写入，解析器之能读；另外一部分是解析器的状态 IStat，这个状态是解析器写入，控制器只能读。

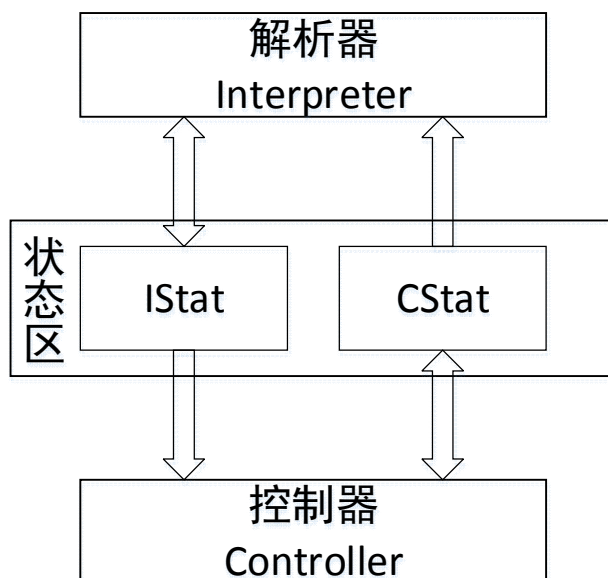


图 3 状态区

## 寄存器区

寄存器区存放解析器使用的各类寄存器。该区域只能由解析器写入，解析器和控制器都可以读。寄存器类型有：位姿寄存器 PR、字符串寄存器 SR、数值寄存器 R、运动寄存器 MR、码垛寄存器 PL

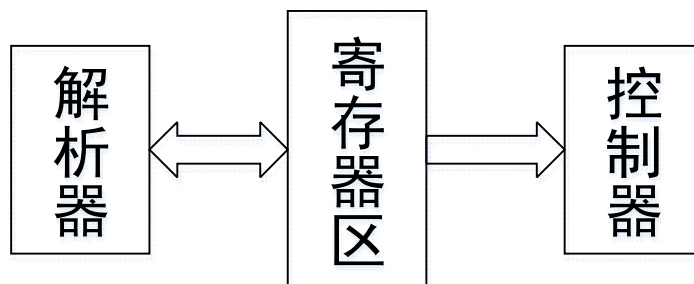
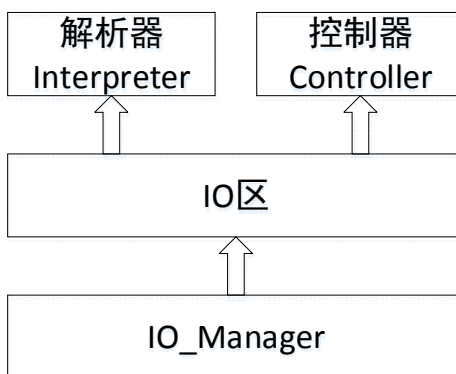


图 4 寄存器区

## IO 区

IO 区存放的是 IO 的当前状态。解析器和控制器只能读，IO\_Manager 实时更新。



因为，解释器和控制器都需要命令区和状态区。因此，控制器启动的时候，会创建六块共享内存。也就是：解释器命令区、解释器状态区、控制器命令区、控制器状态区、寄存器区和IO区。

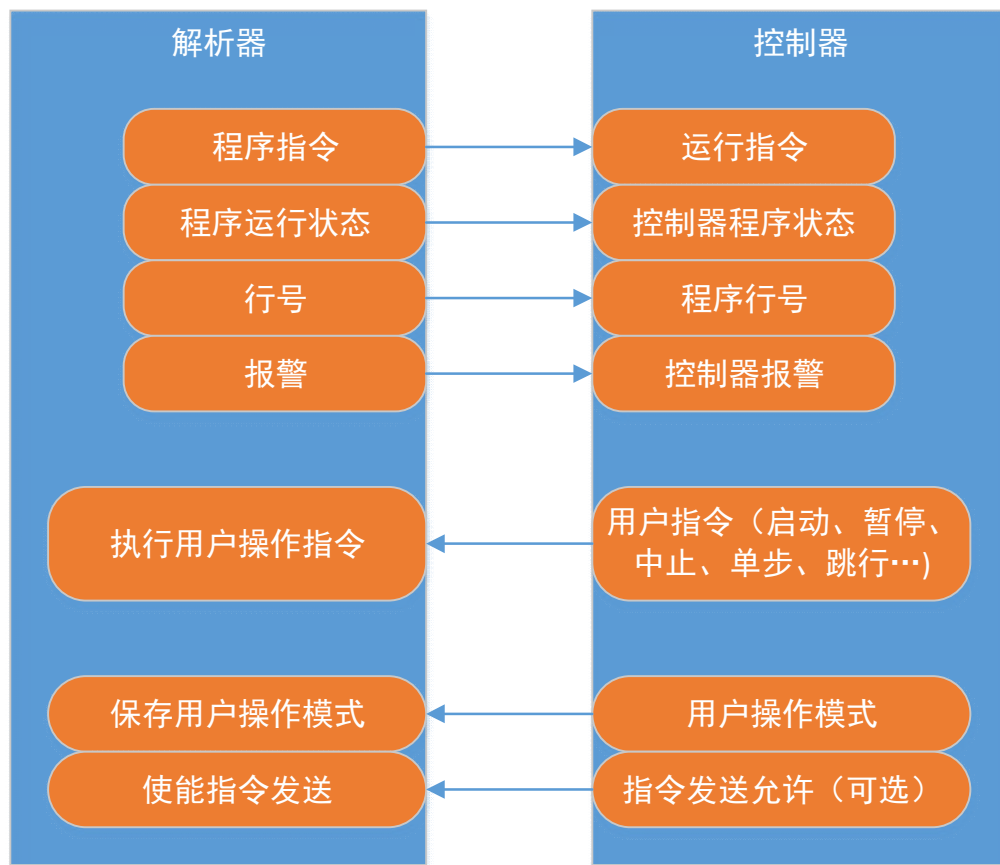
针对解析器与控制器的数据结构体定义如下：

- 1 解释器命令。定义为一个Instruction对象。
  - 1.1 行号。
  - 1.2 指令类型。包括MOTION和END。
  - 1.3 MOTION信息。
  - 1.4 是否包含附加指令。

- 1.5 附加指令个数。
- 1.6 附加指令缓冲区。
- 2 解释器状态。定义为IntprtStatus信息。如下：
  - 2.1 行号。
  - 2.2 执行状态。包括空闲，执行，暂停。
  - 2.3 程序警告号。
- 3 控制器命令。定义为InterpreterControl对象。定义了下面六个指令。
  - 3.1 程序加载LOAD
  - 3.2 程序跳转JUMP
  - 3.3 程序启动START
  - 3.4 程序正序执行FORWARD
  - 3.5 程序逆序执行BACKWARD
  - 3.6 程序继续执行CONTINUE
  - 3.7 写寄存器MOD\_REG
- 4 控制器状态。
  - 4.1 是否可以下发指令。
- 5 寄存器区
- 6 IO区。

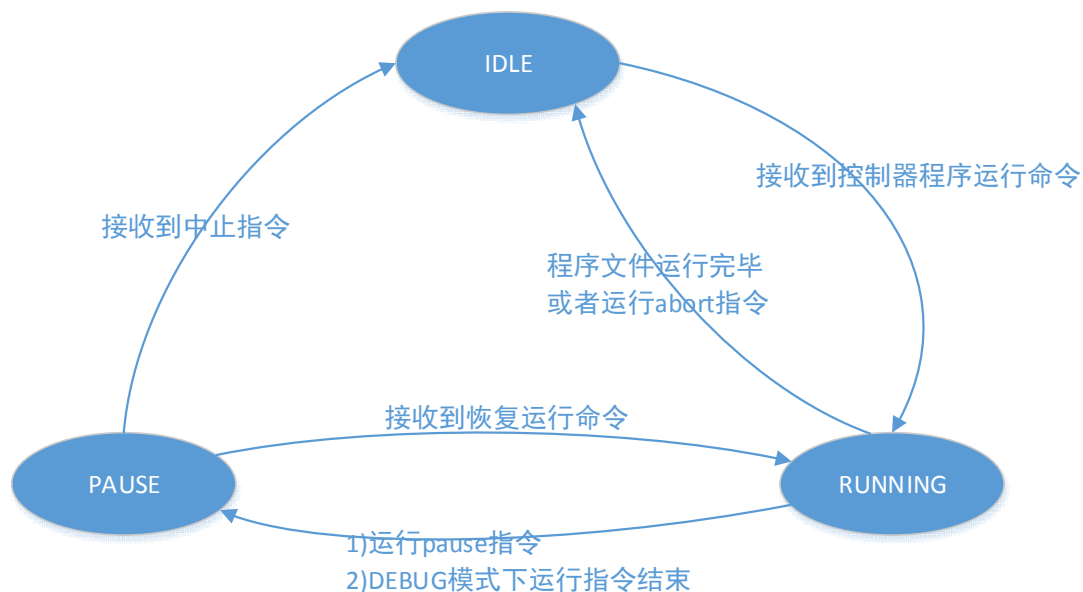
解析器启动的时候，打开这六块共享内存。通过读写共享内存的方式进行交互。交互如下：





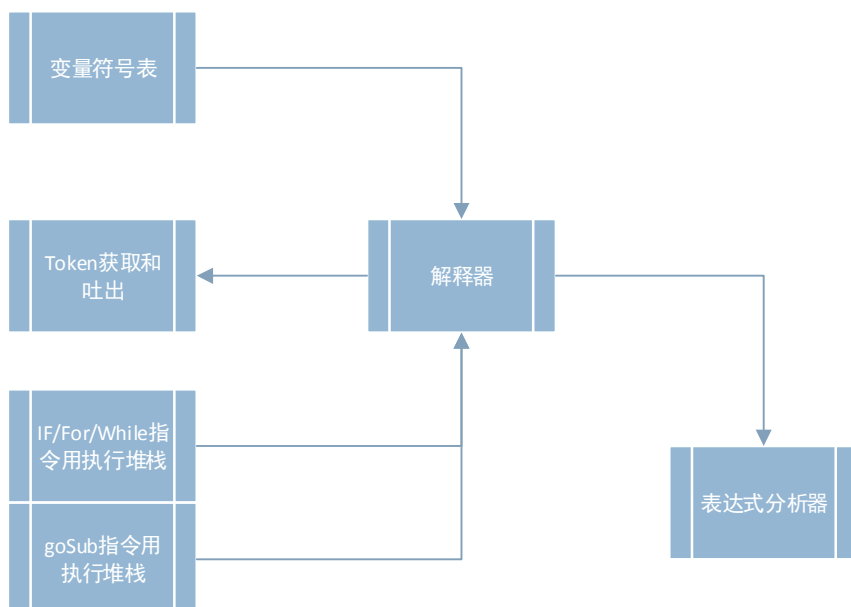
## 9. 解析器状态机

解析器解析用户程序并向控制器发送指令，而控制器也控制解析器的运动过程，状态机如下：



## 10. 解析程序结构

BAS-INT 是网上的一份开源代码。梁肇新的《编程高手箴言》中也引用过这套代码。

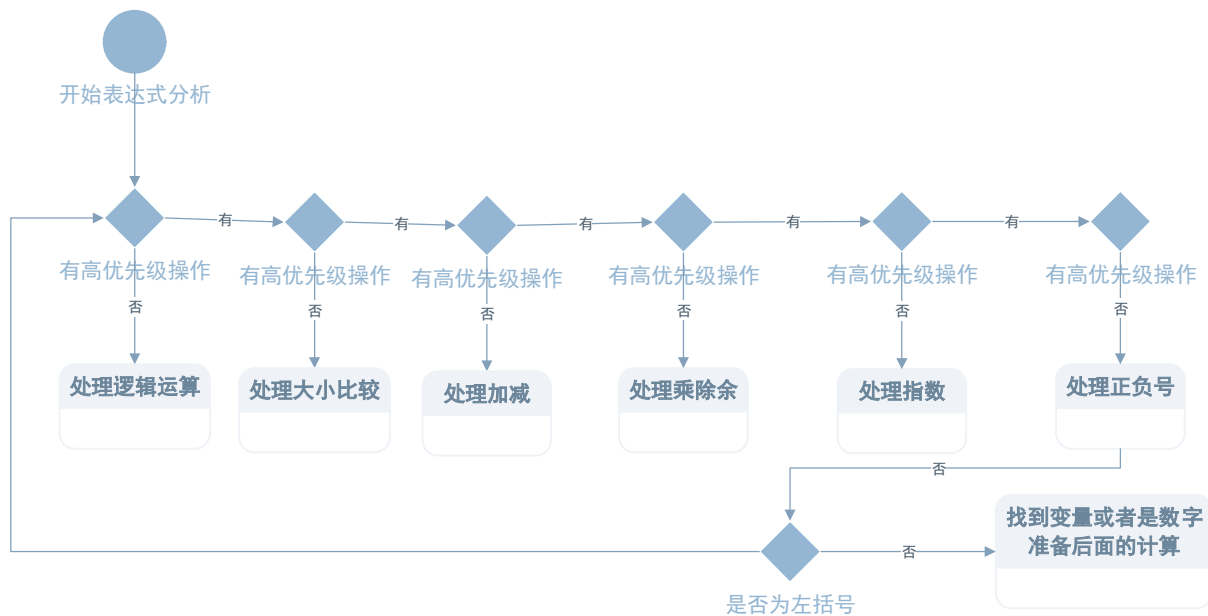


## 11. 表达式分析器流程图

根据《Rtimeman Robot\_核心系统\_用户指令（PartI 语法格式）\_需求文档\_V1.0》

中的要求。表达式分析器需要支持下面的运算操作符：

1. 条件运算如 AND 和 OR。
2. 比较运算如，大于，等于，小于以及组合。
3. 加减乘除。

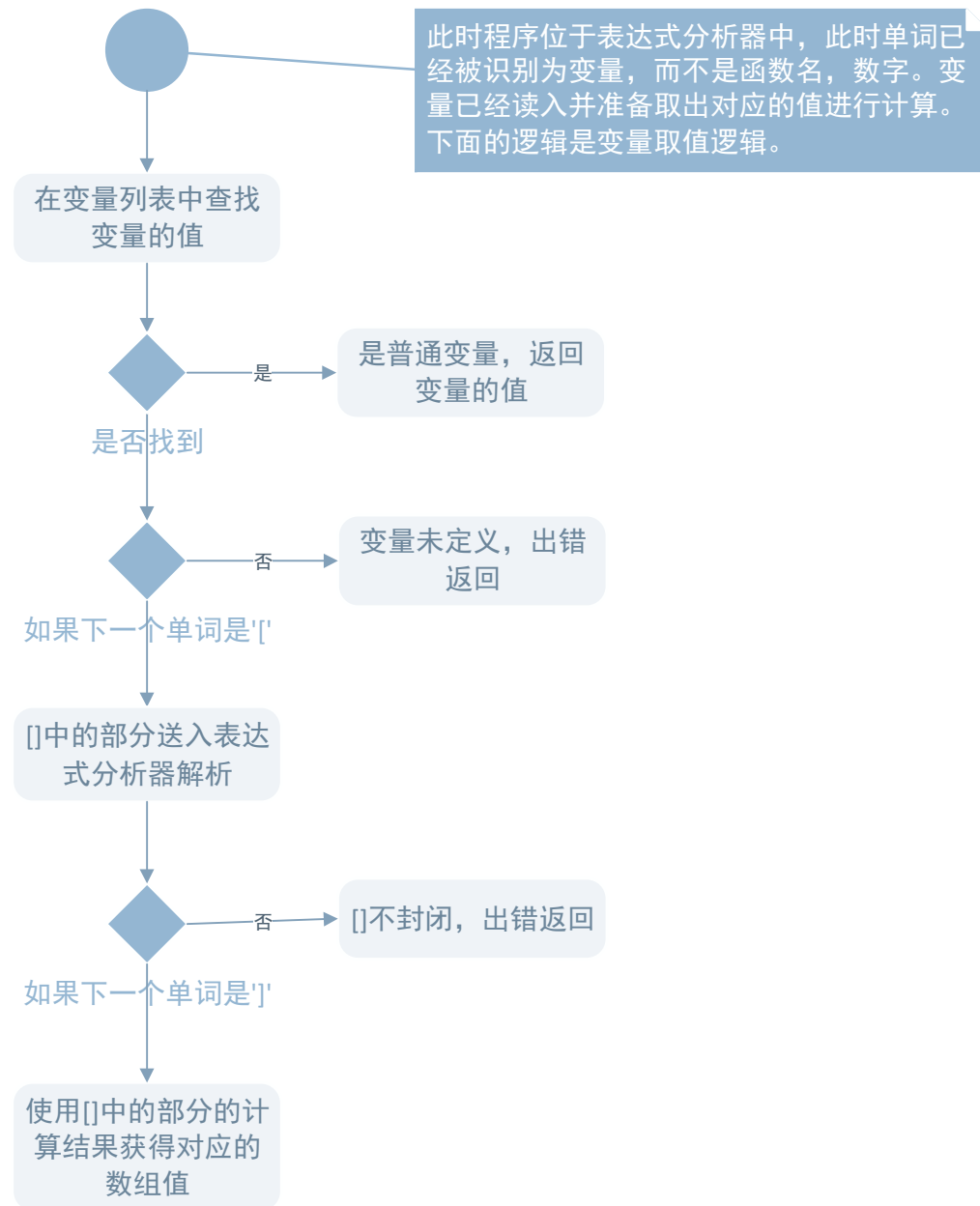


## 12. 表达式分析器对于数组的支持

因为所有的寄存器都是数组格式的。因此上表达式分析器需要支持数组格式的变量。

这里利用了一个规定：

也就是对于形如 `PR[x]` 这样的寄存器。`PR` 是一个关键字，不能作为普通变量名。



## 13. 表达式分析器对于数学函数的支持

表达式支持 C 语言的大部分数学函数。如下。

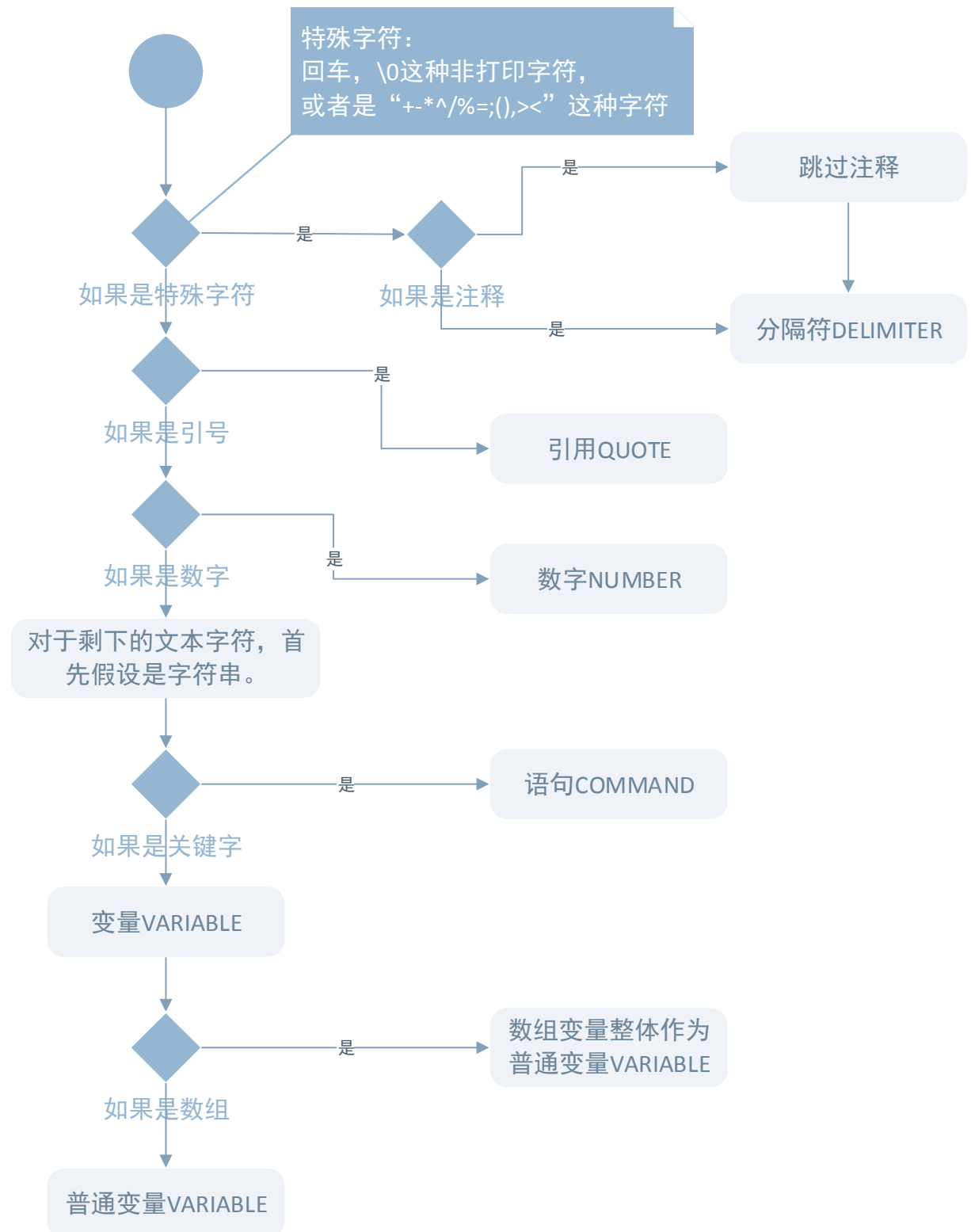
函数	解释	函数	解释
<b>fabs</b>	求整型 x 的绝对值，返回计算结果。	<b>fmod</b>	求整除 x/y 的余数，返回该余数的双精度。
<b>acos</b>	计算 $\cos^{-1}(x)$ 的值，返回计算结果，x 应在-1 到 1 范围内。	<del><b>frexp</b></del>	<del>把双精度数 val 分解为数字部分(尾数)和以 2 为底的指数 n，即 <math>val = x * 2^n</math>，n 存放在 eptr 指向的变量中。返回数字部分 <math>0.5 \leq x &lt; 1</math>。</del>
<b>asin</b>	计算 $\sin^{-1}(x)$ 的值，返回计算结果，x 应在-1 到 1 范围内。	<b>log</b>	$\log_e x$ ， $\ln x$ 。返回计算结果。
<b>atan</b>	计算 $\tan^{-1}(x)$ 的值，返回计算结果。	<b>log10</b>	求 $\log_{10} x$ 。返回计算结果。
<b>atan2</b>	计算 $\tan^{-1}(x/y)$ 的值，返回计算结果。	<b>pow</b>	计算 $x^y$ 的值，返回计算结果。
<b>cos</b>	计算 $\cos(x)$ 的值，返回计算结果，x 的单位为弧度。	<b>rand</b>	产生-90 到 32767 间的随机整数。返回随机整数。
<b>cosh</b>	计算 x 的双曲余弦 $\cosh(x)$ 的值，返回计算结果。	<b>sin</b>	计算 $\sin x$ 的值。返回计算结果。x 单位为弧度。
<b>exp</b>	求 $E^x$ 的值，返回计算结果。	<b>sinh</b>	计算 x 的双曲正弦函数 $\sinh(x)$ 的值，返回计算结果。
<b>fabs</b>	求 x 的绝对值，返回计算结果。	<b>sqrt</b>	计算根号 x。返回计算结果。x 应 $\geq 0$ 。
<b>floor</b>	求出不大于 x 的最大整数，返回该整数的双精度实数。	<b>tan</b>	计算 $\tan(x)$ 的值，返回计算结果。x 单位为弧度。
		<b>tanh</b>	计算 x 的双曲正切函数 $\tanh(x)$ 的值。返回计算结果。

## 14. token 取词器流程图

首先是获取 token 类型的 `get_token` 函数返回的 `token_type` 如下。

<code>#define DELIMITER</code>	1	分隔符
<code>#define VARIABLE</code>	2	变量
<code>#define NUMBER</code>	3	数字
<code>#define COMMAND</code>	4	命令，也就是语句
<code>#define STRING</code>	5	字符串，中间类型。 最后要么是命令，要么是变量。
<code>#define QUOTE</code>	6	引用，其实就是用双引号包起来的字符串。

为了处理这六种情况。处理的顺序非常重要。因此上逻辑如下：



读取的单词被保存在一个全局变量中，因为存在读取过后，需要再次吐出来的情况。

## 15. token 和 tok

这里还有个 tok 的概念。

#define PRINT	1	#define INPUT	2
#define IF	3	#define THEN	4
#define ELSE	5	#define FOR	6
#define NEXT	7	#define TO	8
#define GOTO	9	#define EOL	10
<b>#define FINISHED</b>	<b>11</b>	#define GOSUB	12
#define RETURN	13	#define BREAK	14
#define CONTINUE	15	#define SELECT	16
#define CASE	17	#define WHILE	18
#define WEND	19	#define ENDIF	20
#define ENDIF	21	#define LOOP	22
#define ENDLOOP	23	#define SUB	24
#define CALL	25	#define END	26
#define IMPORT	27	#define DEFAULT	28
#define WAIT	29		

这里大部分都是命令。但是有一个特殊的命令 **FINISHED**。该命令在取词器读到\0时，被设置。表明代码解析结束。当然一般来说，主程序的最后一句正常来说是 **END**。

## 16. 符号表

符号表定义如下：

```
// This structure encapsulates the info
// associated with variables.
struct var_type {
    char var_name[80]; // name
    float fValue ;
};
```

之后这里定义的是一个 vector 变量。

全局变量：      vector<var\_type> global\_vars;

局部变量：      vector<var\_type> local\_var\_stack;



## 17. 双运算符的实现

获取 token 类型的 `get_token` 函数。会设置下面三个全局变量：

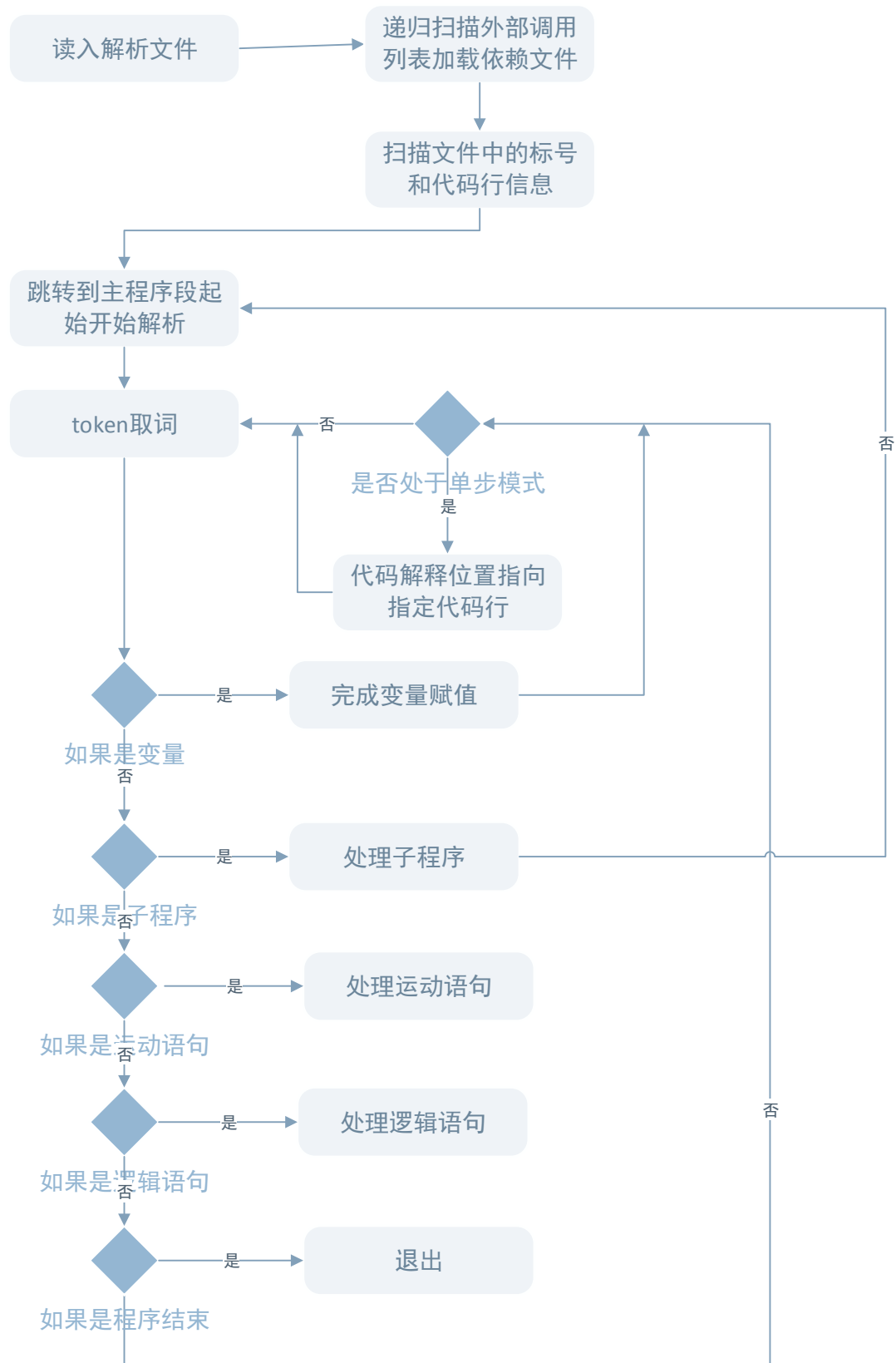
- `char token[80];`
- `char token_type;`
- `char tok;`

函数 `get_token` 执行过后，读取的单词作为字符串保存在 `token` 中。  
这里如果需要实现双运算符，需要在读词的时候，进行翻译，变成特殊字符。

```
enum double_ops {  
    LT=1,    // value < partial_value  
    LE,      // value <= partial_value  
    GT,      // value > partial_value  
    GE,      // value >= partial_value  
    EQ,      // value == partial_value  
    NE,      // value <> partial_value  
    AND,     // value AND partial_value  
    OR,      // value OR partial_value  
};
```

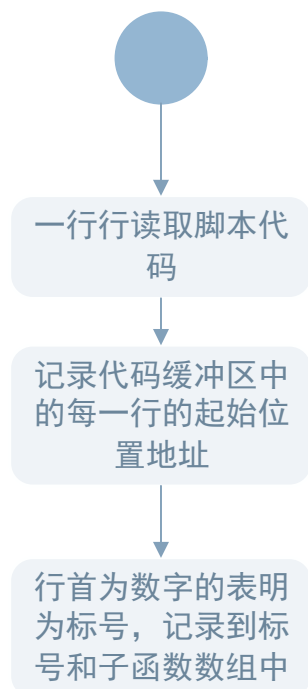
## 18. 程序总体流程图

该流程支持多线程。



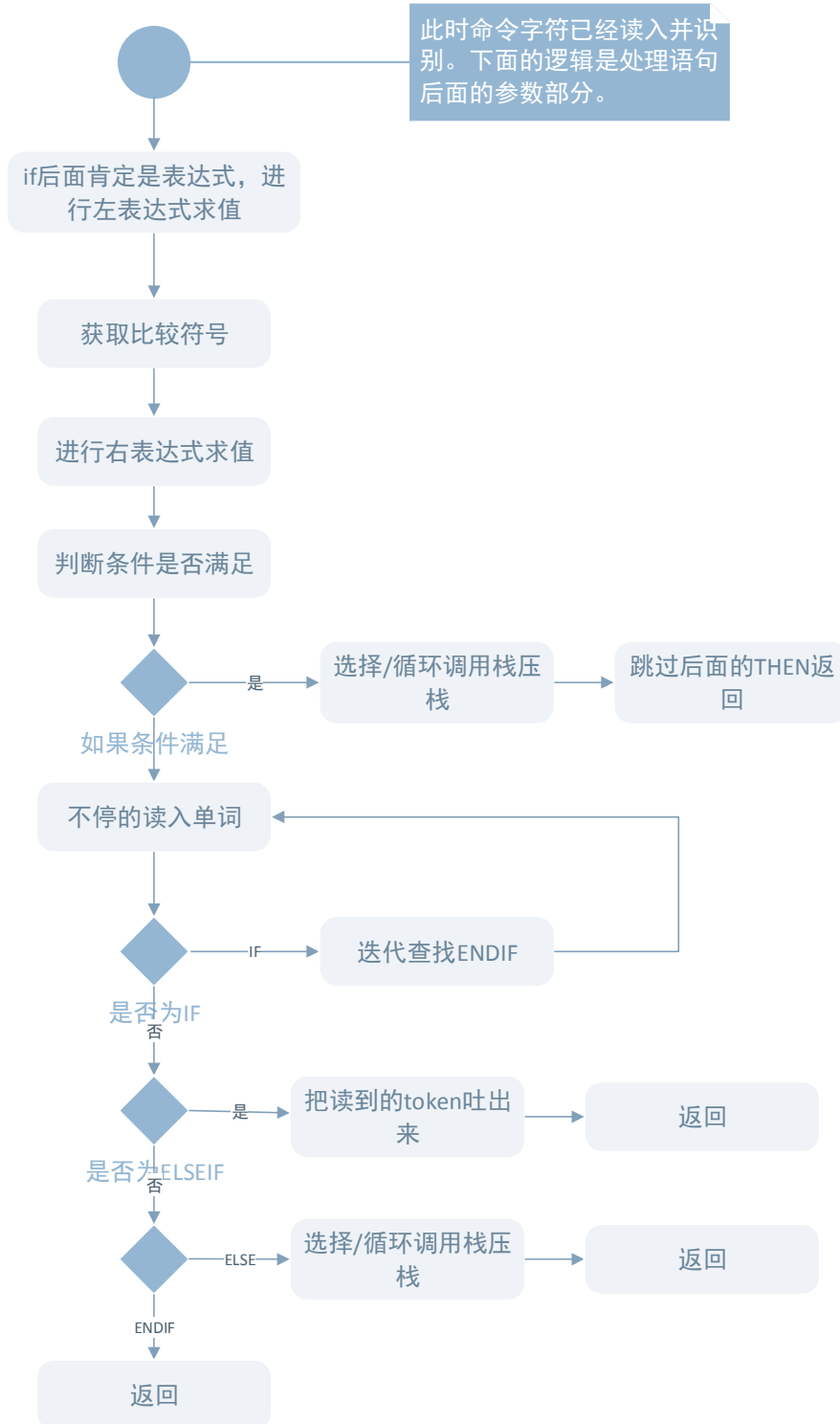
## 19. 扫描文件中的标号和代码行信息流程图

在 BASIC 语法中，函数的开头单词为 SUB。



## 20. 程序命令 IF 和 ELSEIF 解析流程图

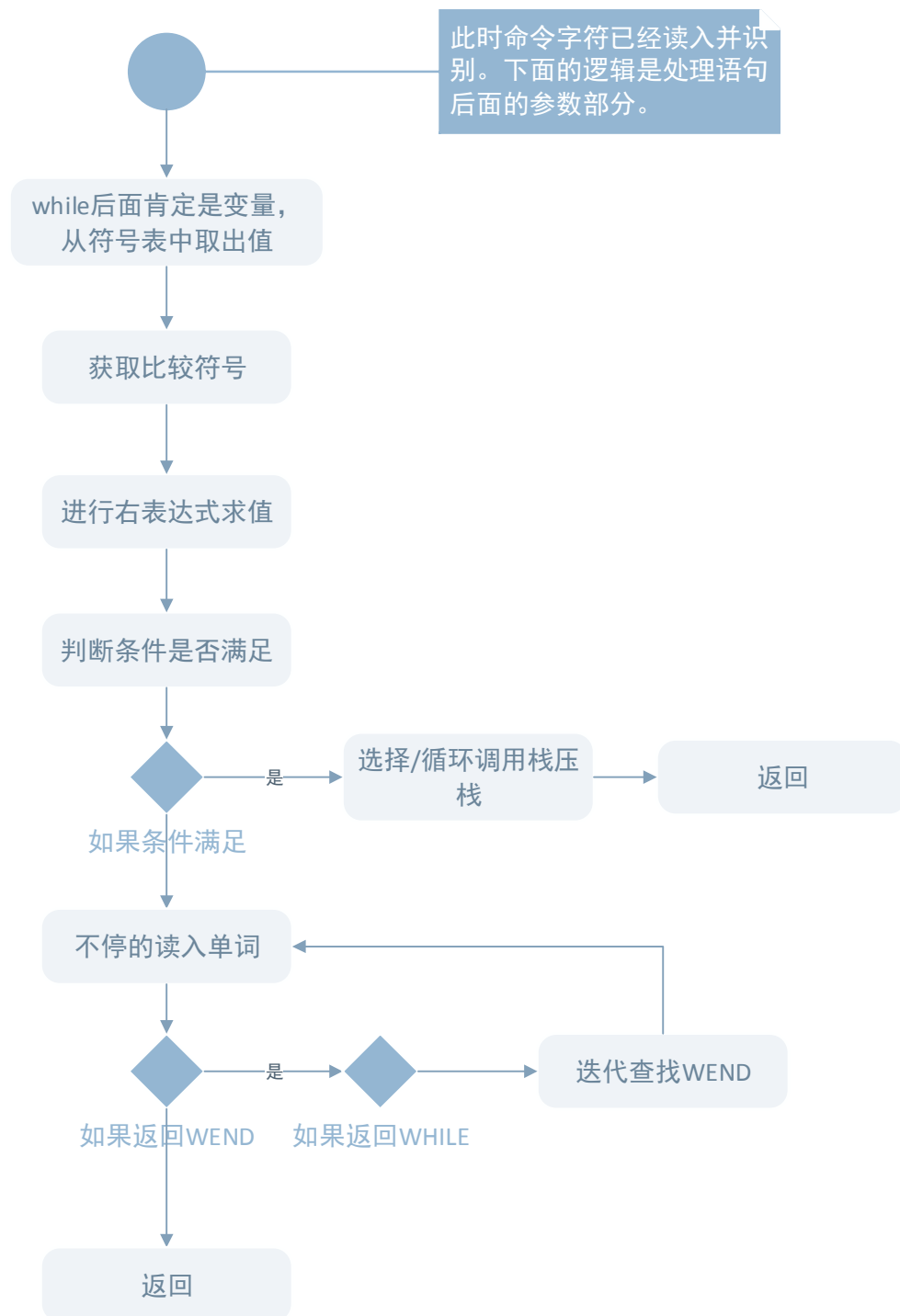
判断条件后，满足条件就继续解析。不满足条件，就选择跳过这个代码块。



## 21. 程序命令 while 解析流程图

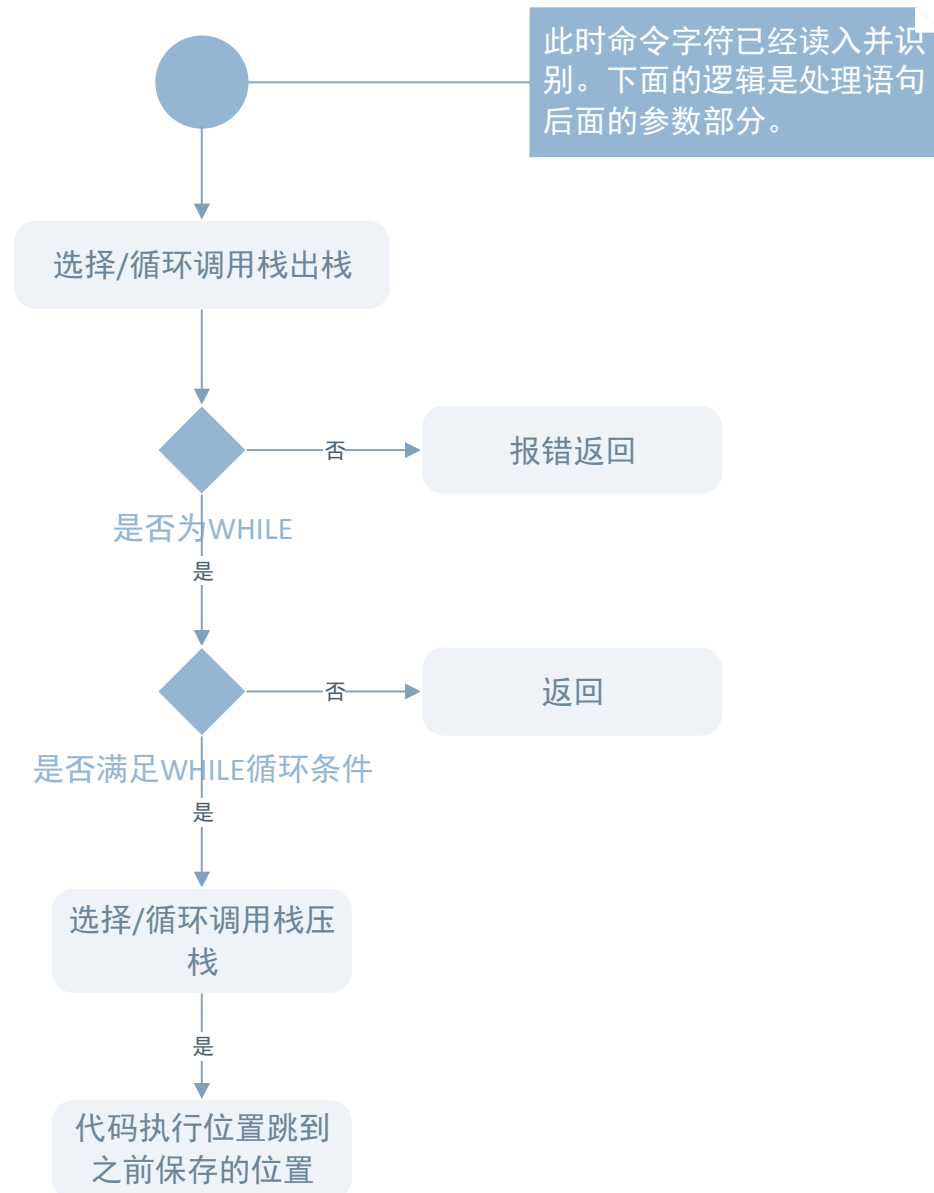
首先 WHILE 语句是需要和 WEND 语句配对的。因此上需要定义一个用于配对栈的 fstack 栈变量。定义如下：

```
typedef struct select_and_cycle_stack {
    int itokentype ;                配对类型： IF/ FOR/ WHILE
    // For and While with now
    float target; /* target value */  FOR 和 WHILE 用目标值
    // For And While
    int var; /* counter or selector variable */  循环变量名。
    char *loc;  代码循环起始位置。
} select_and_cycle_stack_struct ;
```



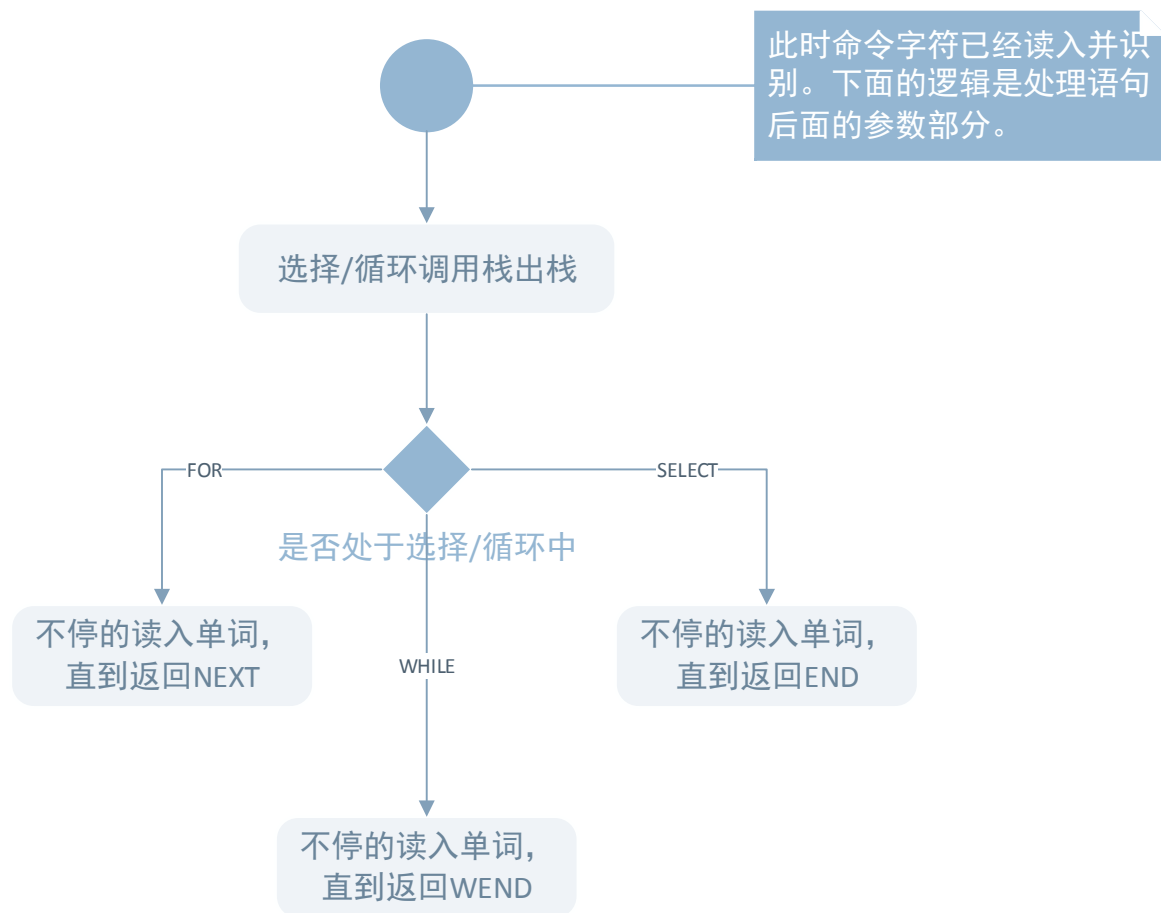
## 22. 程序命令 wend 解析流程图

wend 的功能是判断循环变量是否满足循环条件。



## 23. 程序命令 **break** 解析流程图

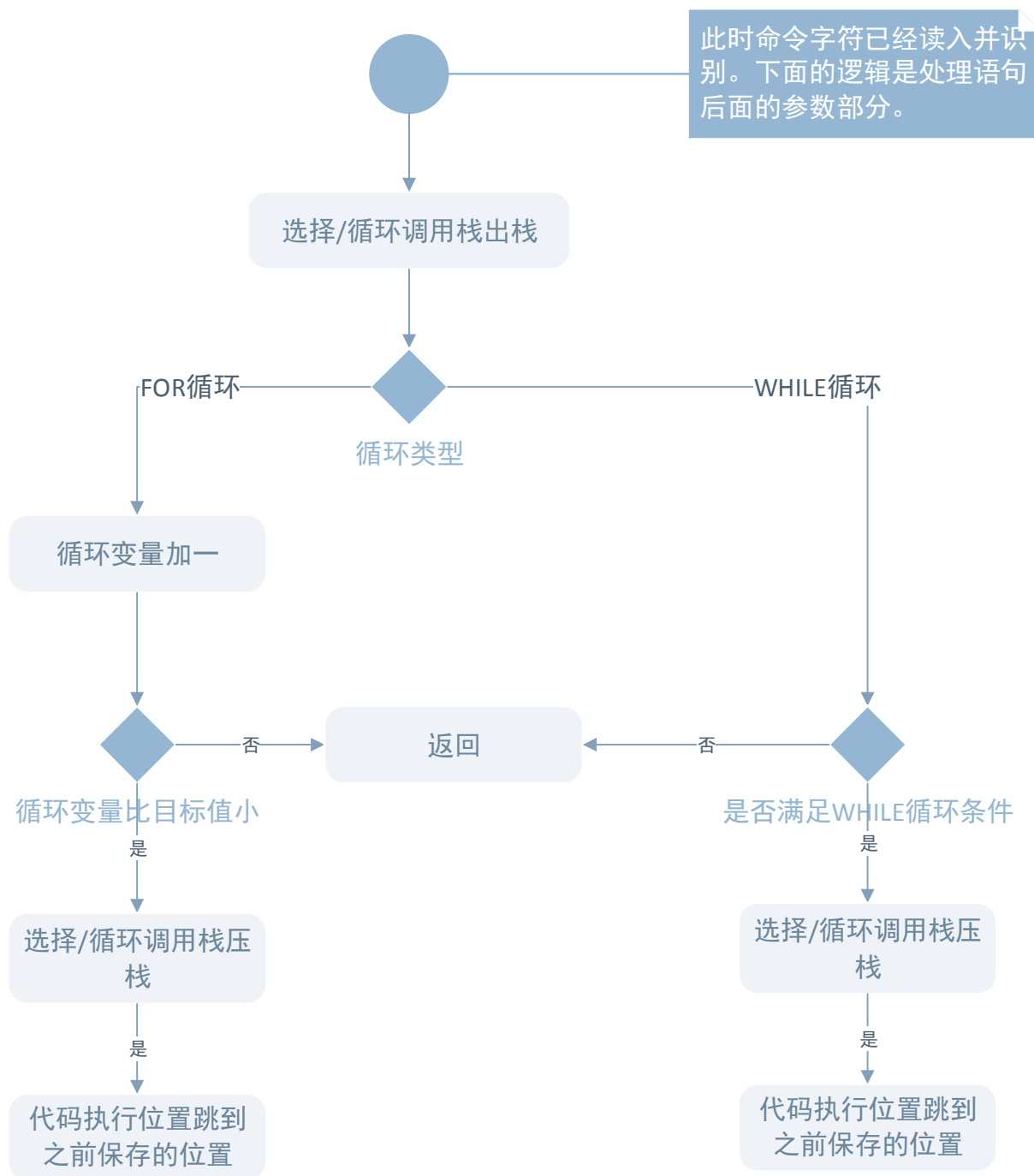
Break 的功能是跳出循环。





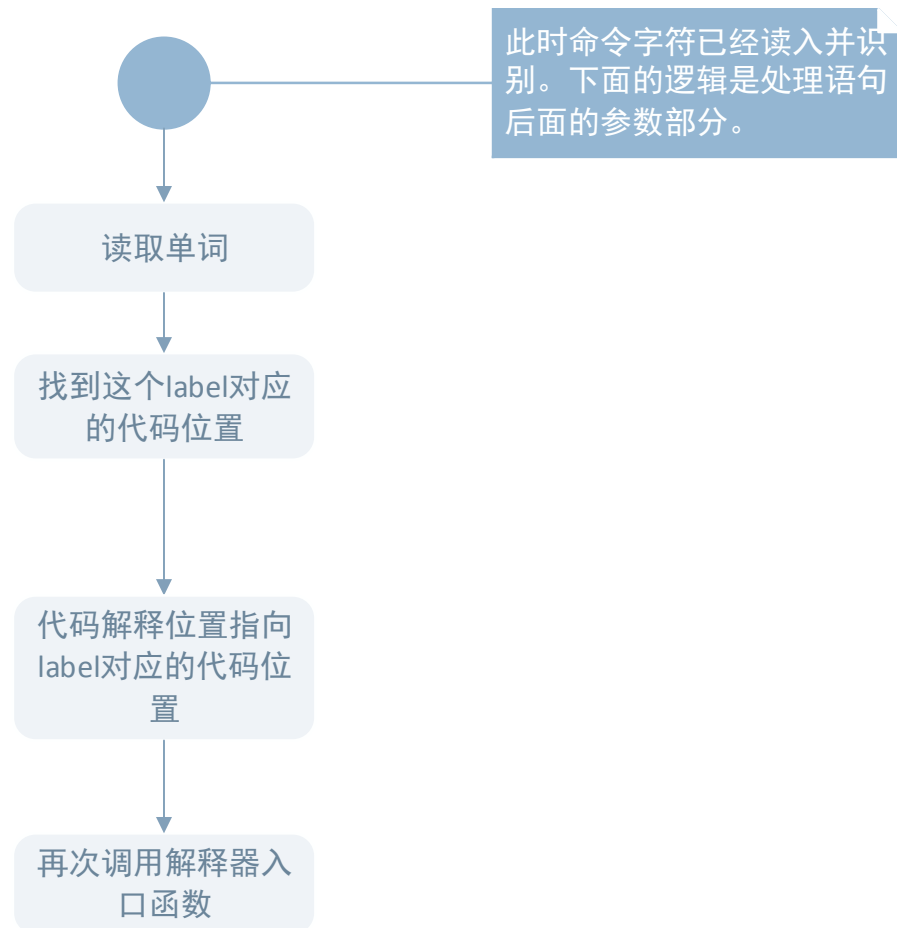
## 24. 程序命令 **continue** 解析流程图

Continue 的功能同 Wend 和 NEXT 的功能相同。



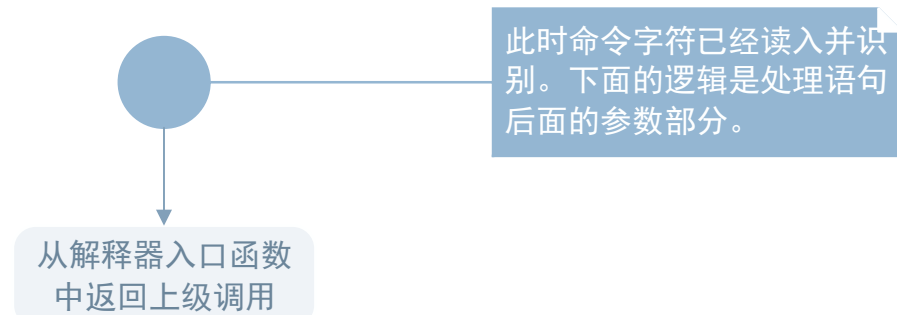
## 25. 程序命令 `call/gosub` 解析流程图

直接调用解释器入口函数实现该功能。



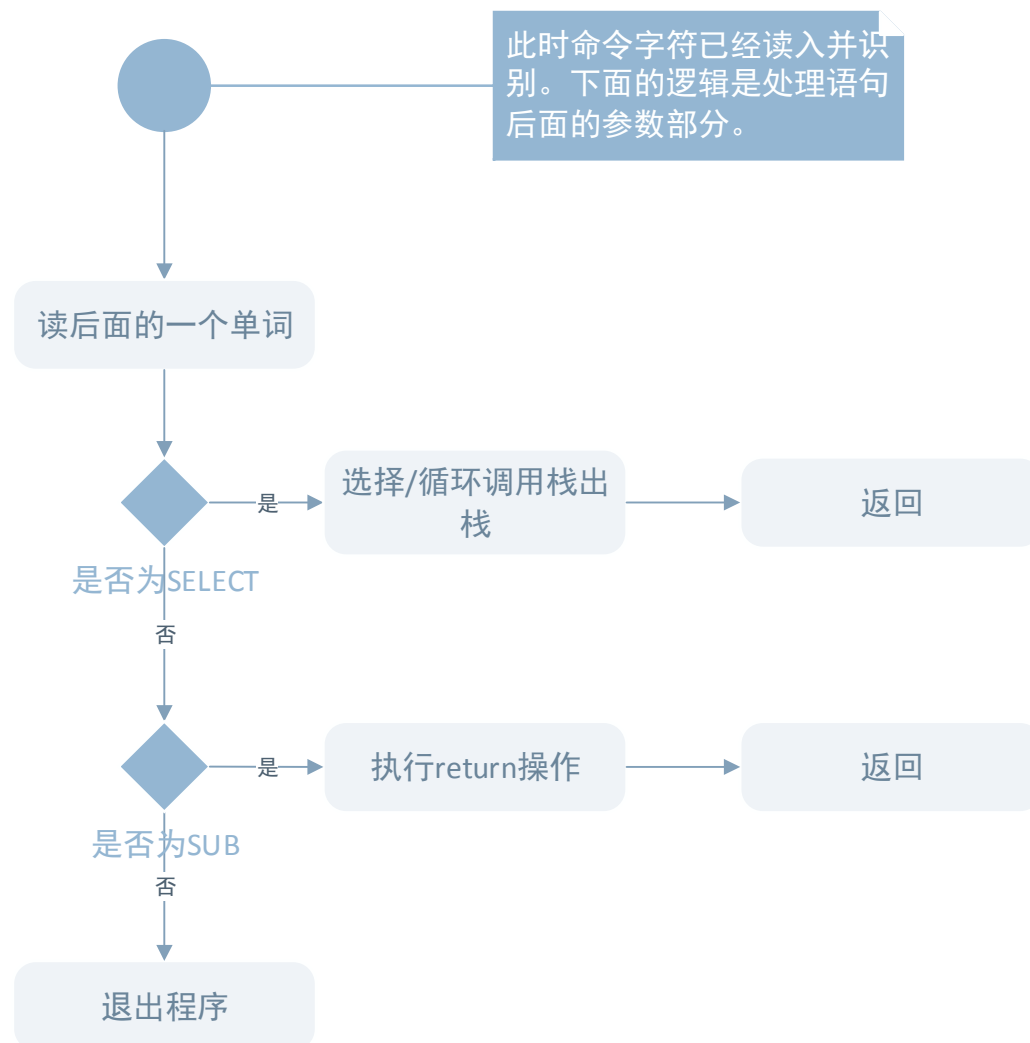
## 26. 程序命令 `return` 解析流程图

直接使用 C 语言的函数功能返回实现。



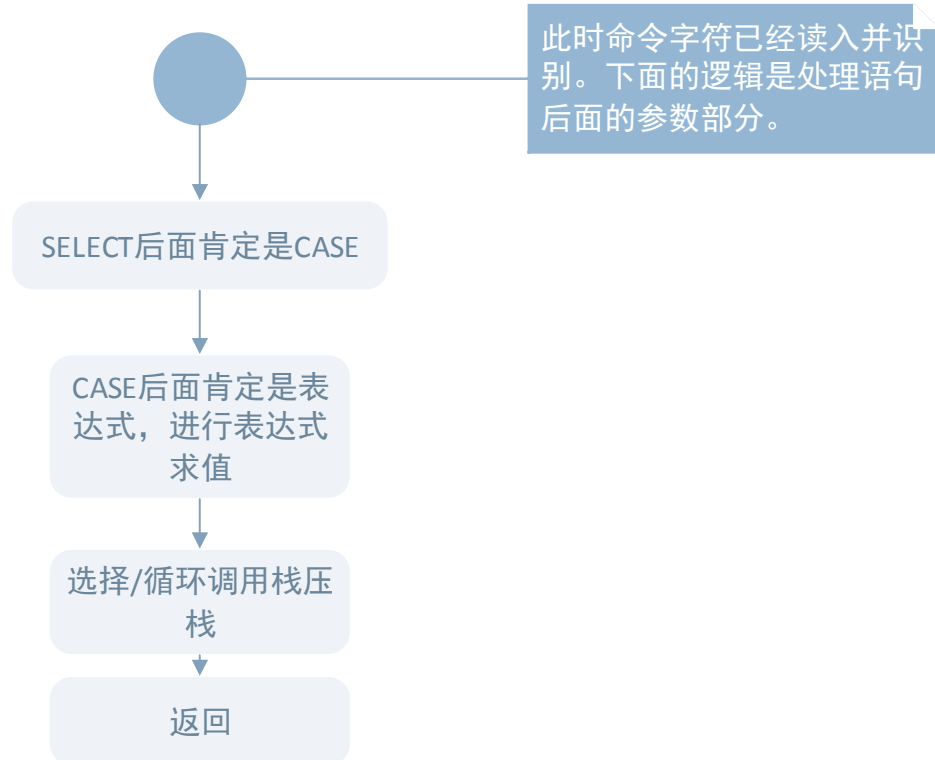
## 27. 程序命令 end 解析流程图

这里处理了三种情况，分别是 END SELECT 语句，END SUB 语句和程序结束 END 语句。



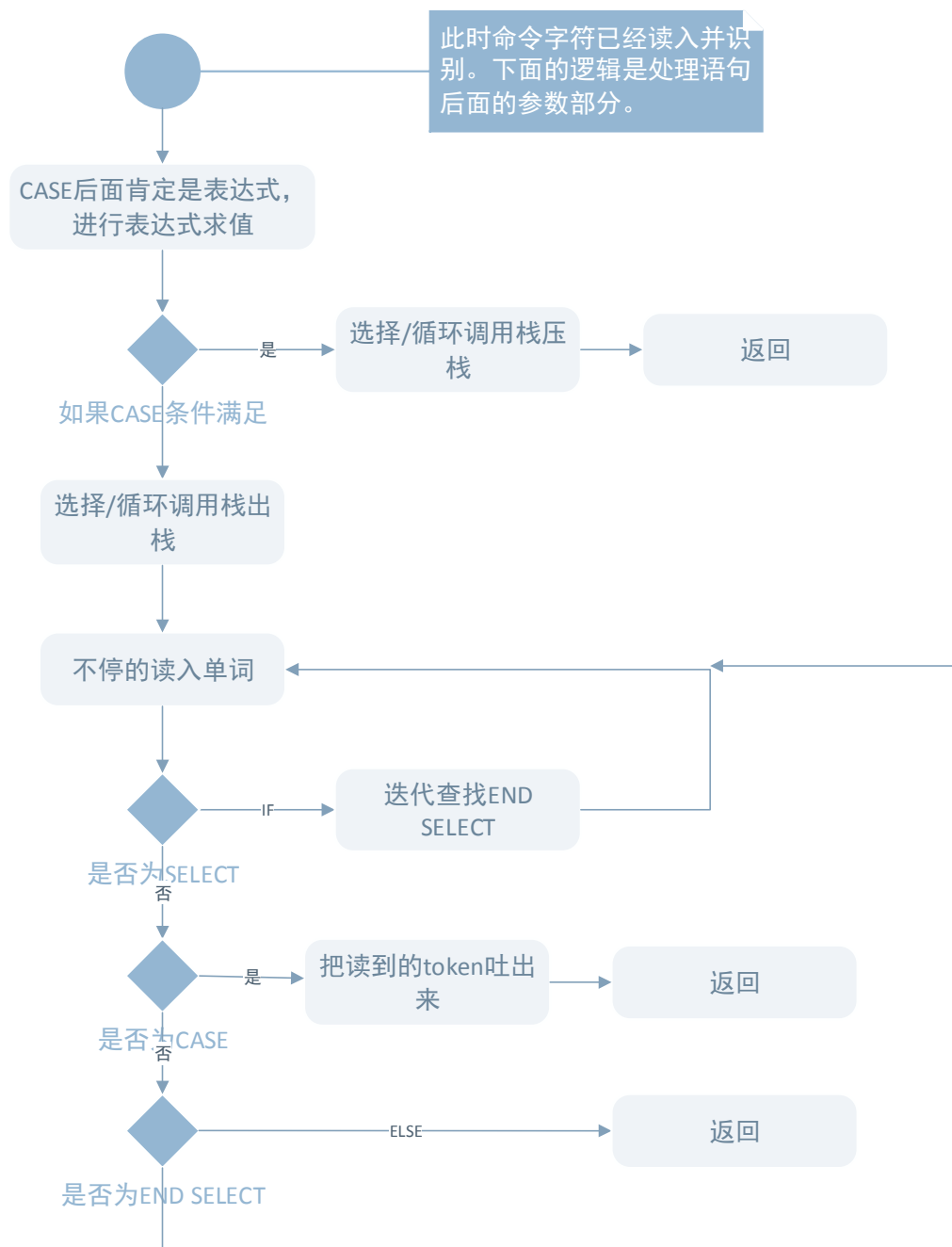
## 28. 程序命令 **select** 解析流程图

Select 语句的作用是记录用于 case 语句比较的变量。



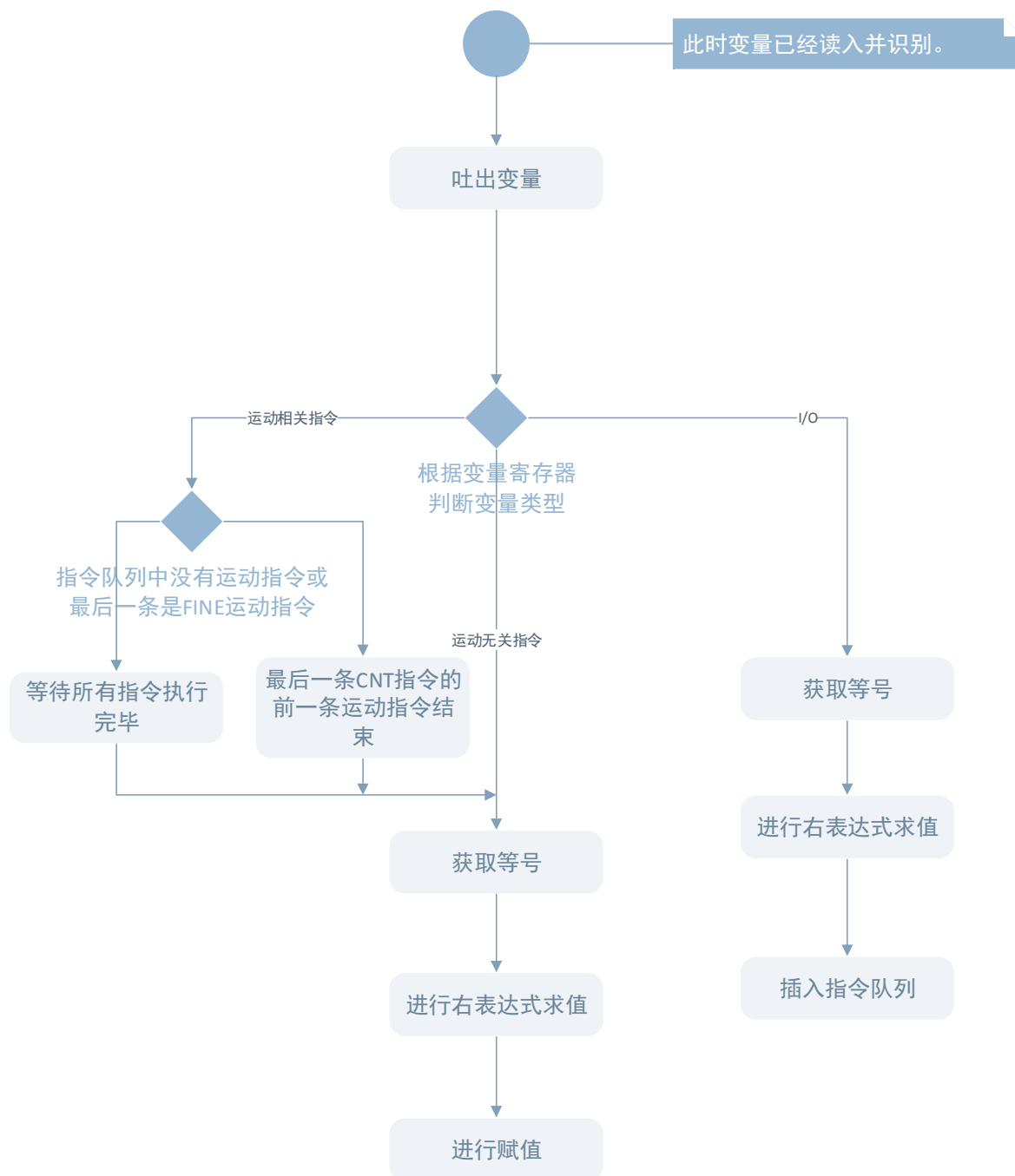
## 29. 程序命令 **case** 解析流程图

case 语句的作用是和 Select 语句变量进行比较。



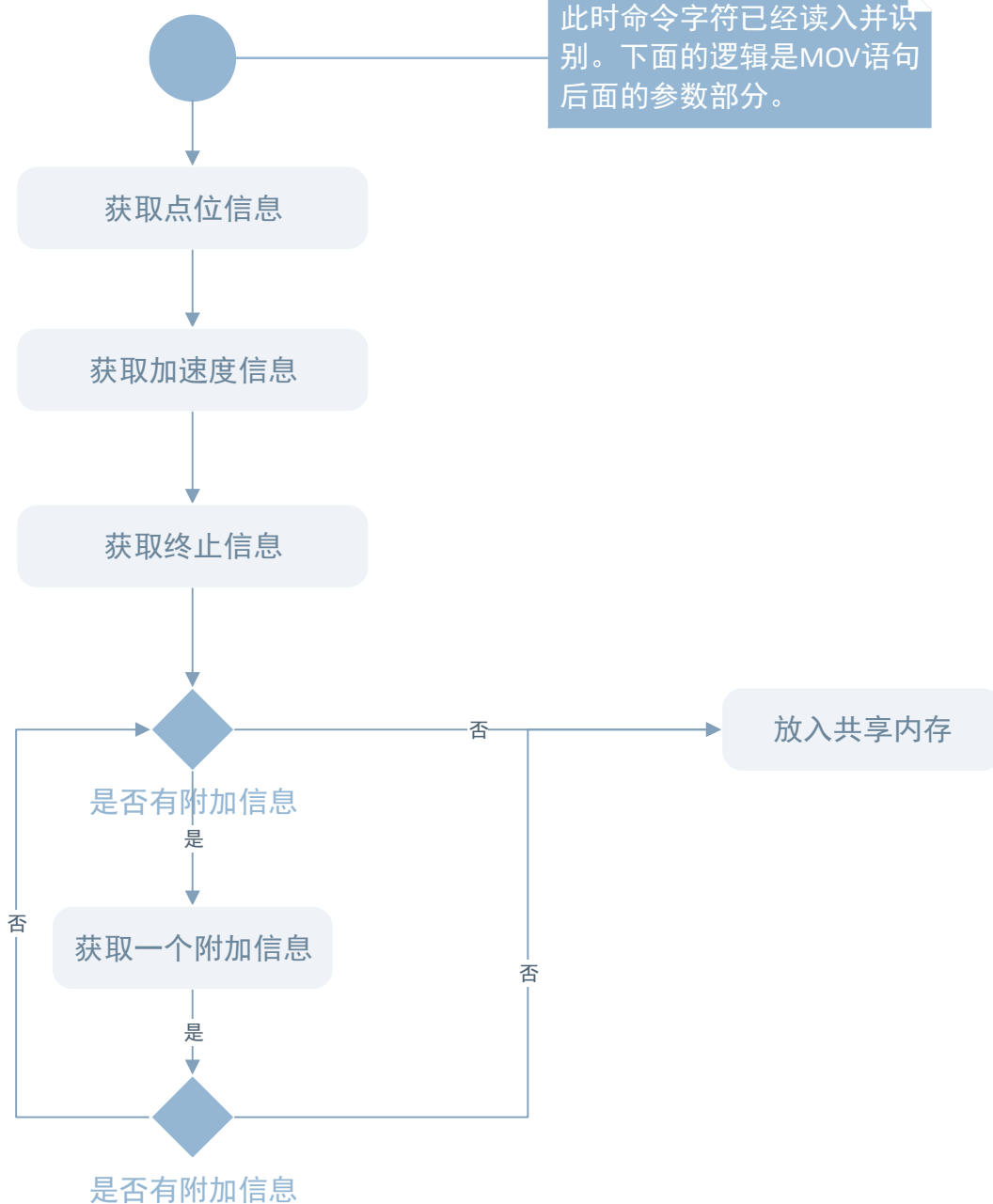
## 30. 赋值语句流程图

脚本解析器在读到变量的时候，会进入赋值流程。



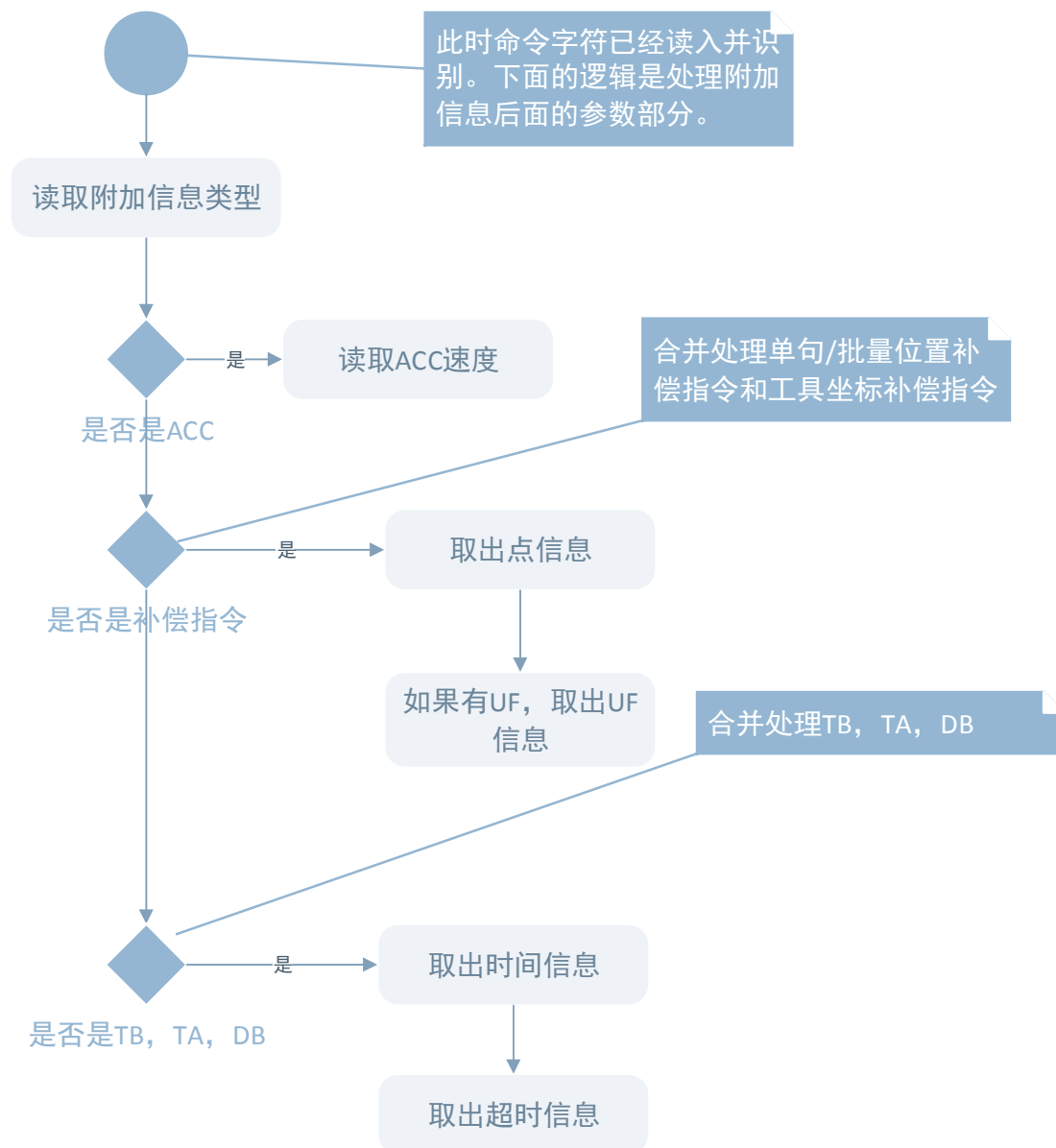
## 31. MOV 相关命令解析流程图

MOV 命令的参数依次为：点位，加速度，终止信息和扩展参数。



## 32. MOV 相关命令扩展参数解析流程图

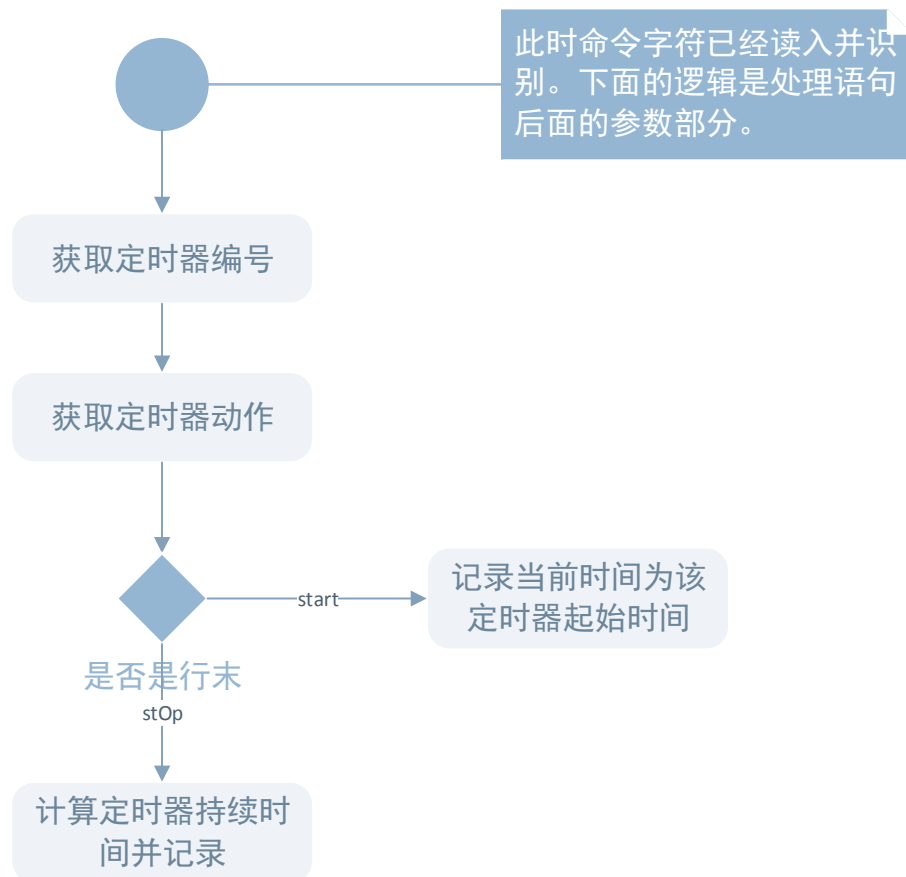
支持 ACC，位置补偿，TB/TA/DB 三种扩展参数。



## 33. Timer 命令解析流程图

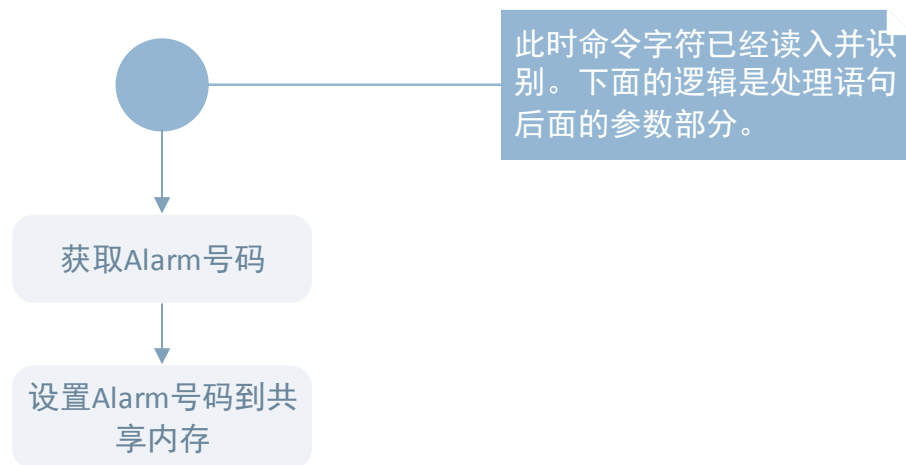
实现方式就是实现一个秒表的功能。记录开始时间和结束时间的的时间差。





## 34. UserAlarm 命令解析流程图

实现方式就是写入用户警告到控制器。



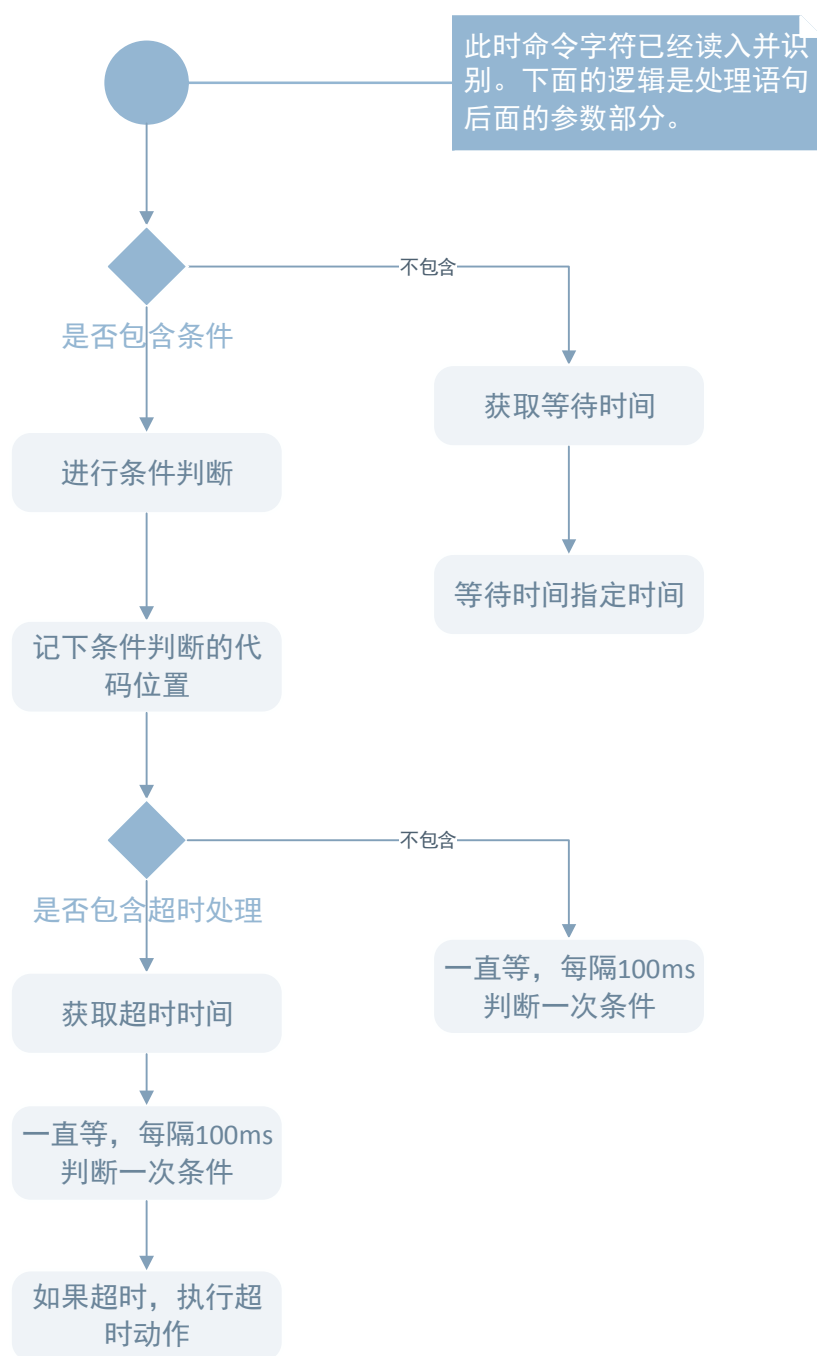
## 35. Wait 命令解析流程图

在之前进行 XML 翻译的时候，会针对是否包含条件，决定是否添加关键字 COND。

WAIT 3

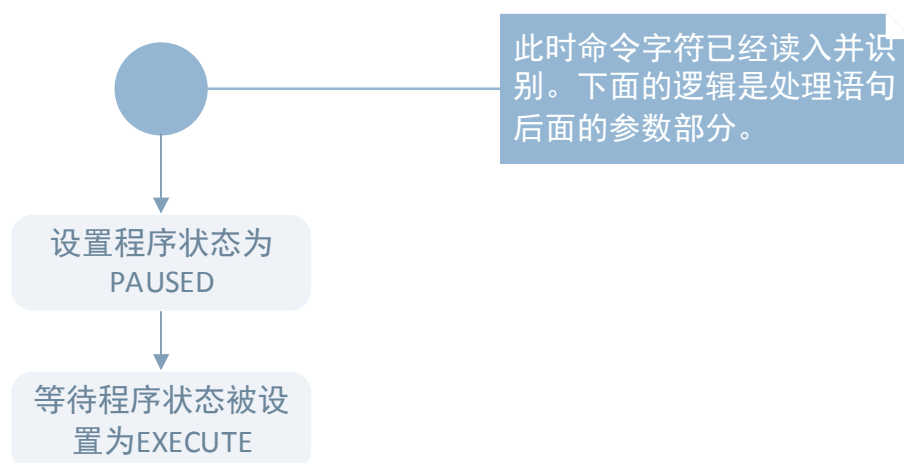
WAIT COND DI[1] = PE

WAIT COND R[1] > 10 3 skip



## 36. Pause 命令解析流程图

实现方式就是设置程序状态为 PAUSED，暂停解析，等待控制器设置为 EXECUTE 后继续执行。



## 37. Abort 命令解析流程图

实现方式就是停止解析。设置程序状态为 IDLE。

