一、问题说明

指针变量在声明过程中，初始化指向目标`target`，如下测试用例所示：

```
program test
    integer,target::targ = 123
    integer,pointer::ptr => targ
    print*,ptr                      // 输出123
end program test
```

以上测试用例，输出结果为目标`targ`变量的存储结果，如果变量`targ`的存储内容发生变化，则`ptr`的输出值也随之变化。

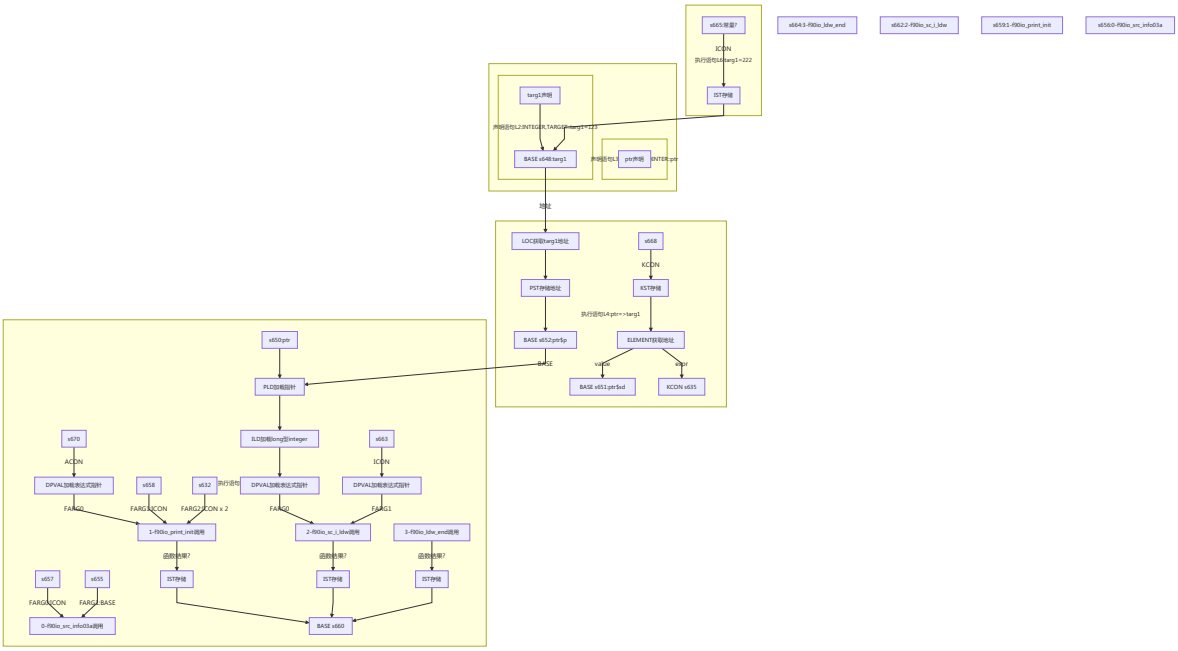
二、语法解析过程中对指针的处理说明

本例通过生成的`ilm`文件，解析指针的处理过程。

测试用例代码如下：

```
1 program a
2     INTEGER,TARGET::targ1=123
3     INTEGER,POINTER::ptr
4     ptr=>targ1
5     print*,ptr
6     targ1=222
7     print*,ptr
8 end program a
```

指针`ILM`处理流程如下：

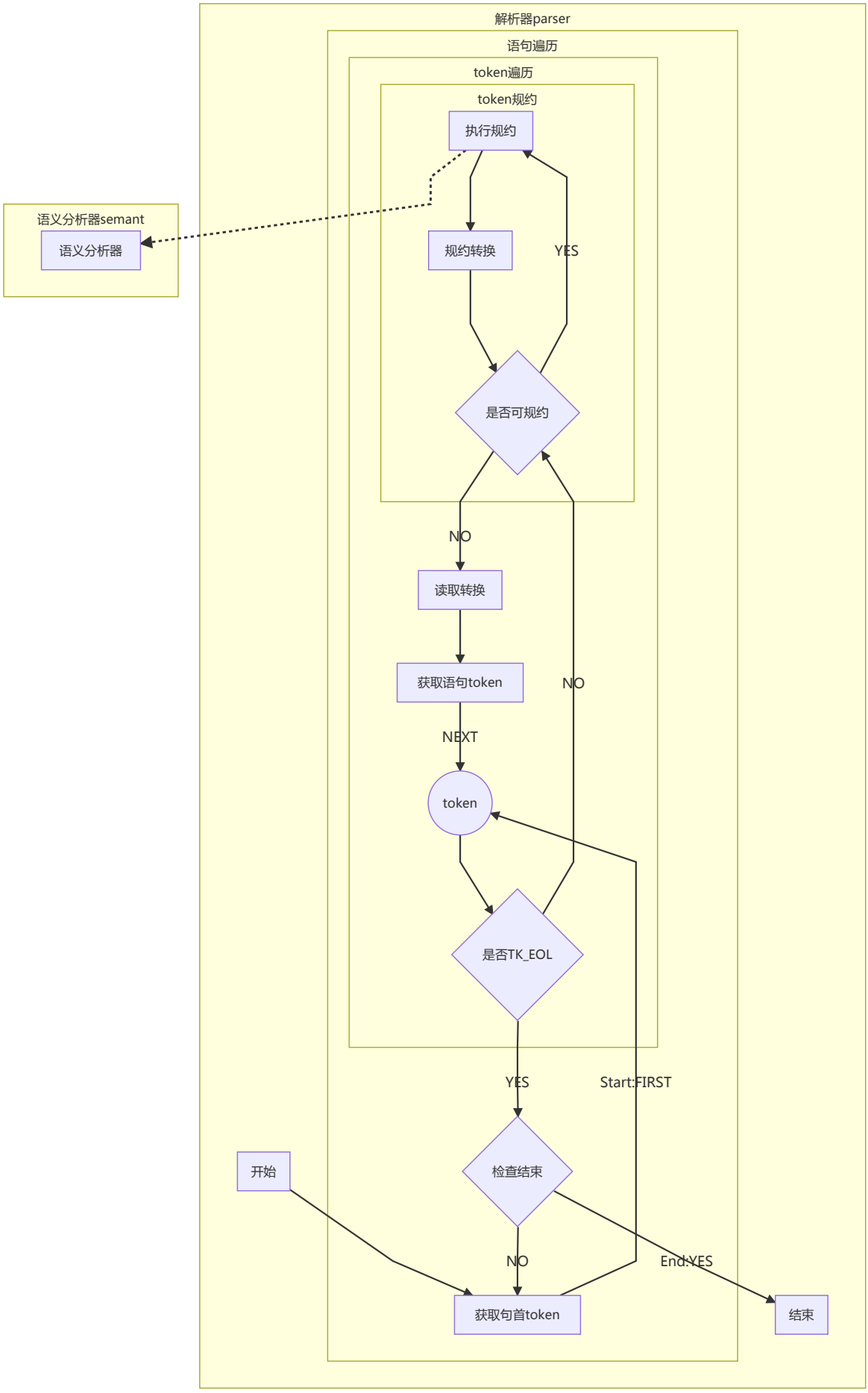


三、语法解析过程和语义处理说明

3.1 语法解析流程

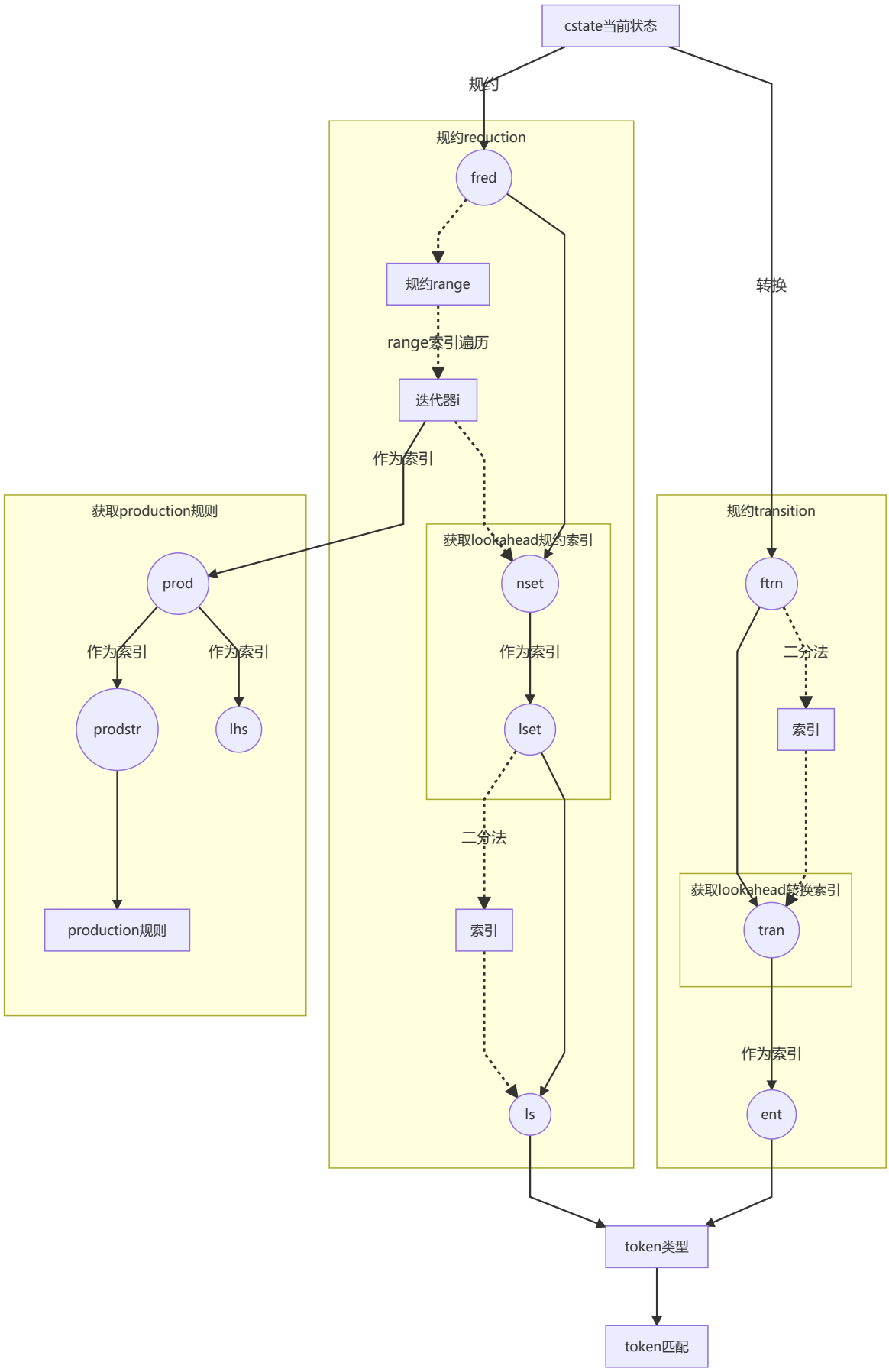
3.1.1 语法流程图

以下流程图，用以说明接口`_parser`中的规约处理流程：



3.1.2 语法解析器的结构关系

以下结构关系图，用以说明接口`_parser`中的规约处理操作：



3.2 语义动作处理说明

3.2.1 语义动作解析

词法解析后，解析器将按照顺序遍历`token`，并查找响应的规约规则，如果有规约规则，则可进行相关规约，规约过程执行语义分析器`semant`相关接口：`semant1`，`semant2`，`semant3`，`semantio`等，其中：

- `semant1`处理声明；
- `semant2`处理表达式和简单语句；
- `semant3`处理分配语句、条件语句、分支和函数/函数调用语句；
- `semantio`处理输入输出io语句；

3.2.2 规约规则的处理

每个语句的规约顶部，都是一个`<stmt>`，该`<stmt>`在规约时，将会被处理解析成其他规约规则，如下图所示，`<stmt>`被解析为`<stmt>::=<stbeg> <statement> <stend>`规则，该规则下也有三个子规则：`<stbeg>`，`<statement>`，和`<stend>`，三个子规则也将会被依次进行规约。

3.2.3 `SST`栈结构处理

规约过程中，每个规则都将占用一个`SST`栈，对当前语法的解析结果（如生成的`ast`，符号表`sptr`，语义解析类型，源码解析位置，形参实参相关信息，`acl`信息等等）都会被存入当前`SST`中，随着当前栈处理完成，相关信息将会被处理到对应结构中，如`ast`抽象语法树，符号表等当中，部分信息将被承接到上一级规约的对应`sst`栈结构中。

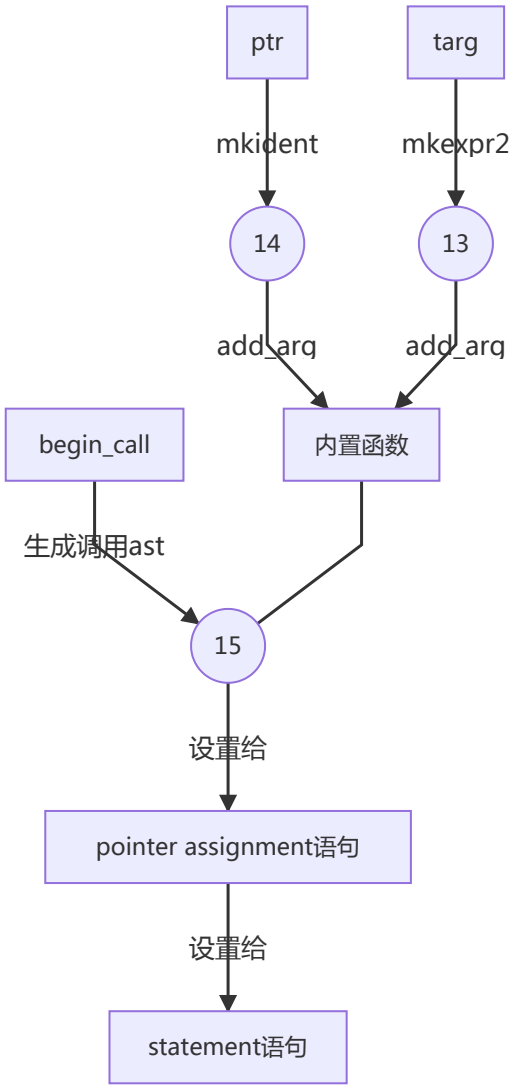
3.2.4 规约规则与语义栈关系

语句`ptr=>targ`规约规则与`SST`语义栈关系图：

对应$token$	语法规则	$sst=2$	3	4	5	6	7	8	9
		$<stmt>$							
	$<stmt>::=$	$<stbeg>$	$<statement>$	$<stend>$					
	$<statement>::=$		$<nii>$	$<nim>$	$<simple\ stmt>$				
	$<simple\ stmt>::=$				$<pointer\ assignment>$				
	$<pointerassignment>::=$				$<psfunc>$	$<var\ ref>$	$<psfunc>$	$'=>'$	$<expression>$
	$<expression>::=$								$<primary>$
	$<primary>::=$								$<var\ ref>$
	$<var\ ref>::=$								$<ident>$
	$<ident>::=$								$<id>$
$targ\uparrow$	$<id>::=$								$<id\ name>$
	$<var\ ref>::=$					$<ident>$			
	$<ident>::=$					$<id>$			
$ptr\uparrow$	$<id>::=$					$<id\ name>$			

四、`ast`抽象语法树处理分析

==`ptr=>targ`语句中，对`ast`抽象语法树的操作==:



语句 `ptr=>targ` 将被处理成对内置函数 `PTR2_ASSIGN` 的调用，并将 `ptr` 和 `targ` 处理成该内置函数的两个参数，通过调用 `begin_call` 生成该内置函数的 `ast`，该 `ast` 最终被当前语句的规约顶部栈接收，并由 `add_stmt` 接口，将当前语句的 `ast` 添加到 `std` 中，以便在 `tranform` 过程中的 `rewrite_call` 中对该内置函数进行处理调用。

五、新增处理代码记录说明

1. 添加指针的初始化语法
    - `tools/flang1/utils/prstab/gram.txt`：添加对 `<init beg>` 的修改；
    - `tools/flang1/flang1exe/semant.c`：添加对新增语法的处理；
2. 处理相关规约过程
    1. 规则 `<entity decl> ::= <entity id> <init beg> <expression>` 的处理
        - `tools/flang1/flang1exe/semant.c`：添加对指针初始化目标的处理，调用 `assign_pointer` 接口，同时将生成的内置函数 `ast`，传递给当前 `SST` 栈，即 `<entity decl>` 的语义栈。
    2. 规则 `<declaration> ::= <data type> <opt attr list> :: <pgm> <entity decl list>` 的处理
        - 通过 `sem.funcval` 字段，判断是否为声明阶段的指针初始化，并将 `<entity decl list>` 的 `ast` 传递给 `<declaration>`。

3. 规则`<statement> ::= <declaration>`的处理
    - 将`<declaration>`的`ast`传递给当前语句`<statement>`，当本语句具备`ast`信息时，语句处理流程`statement_shared`会将语句`ast`添加到`std`中，以便后续的内置函数调用处理。