编译原理课程实验报告(2025 春)

姓名	王圣伟	班级	2211101	学号	2022211821	
成绩	出勤得分		实验报告 得分		实验总分 (共 20 分)	
	实验代码得分					

一、实验目的

实现一个基本的高级语言文法的 SLR(1) parser。即实现读入一个 CFG；解析 CFG，生成 SLR(1)分析表；基于分析表，解析一段代码，输出解析树

综上，需要实现 CFG 文法的解析，SLR(1)分析表的生成（包括 FIRST 集，FOLLOW 集的计算，活前缀识别 DFA 的构建，以及分析表的生成），以及基于分析表给出语法树。对于 CFG 文法文件的解析，使用递归下降方法解析。然后根据算法完成上述过程，以 json 格式导出语法树，并可视化。

此外，本报告还完成了整个编译器前端（以及目标代码生成），即后续的语法制导翻译与目标代码生成工作。因为编译的目标语言 WebAssembly 是一种比较简单的汇编，因此直接生成目标代码，不使用中间表示。项目创造的语言能在任意支持 WebAssembly 的虚拟机或解释器中运行。因为这部分不在实验要求内，这部分将不会在报告中体现，仅做简单的展示。不过，在语法设计部分，会带上语义动作与其解释，解析文法时也会一并解析语义动作与 AST 树生成信息。

Parser 部分使用 C++实现。语义动作，语法制导翻译等不在实验的要求范围内，使用了 JS 语言，以便直接动态解释执行语义动作。

二、实验内容

1. 项目设计

整个项目分为 C++实现的词法分析器，语法分析器（同时会执行进行抽象语法树生成的语义动作），JS 实现的语法制导翻译与运行时。下文称与文法解析直接匹配的解析树为 CST（具体语法树），作为输出结果，带有语义动作信息的树为 AST。

项目使用一个统一的文法文件描述目标语言。它包含若干带有语义动作的 CFG。格式如下，

[AST 动作] 变量->产生式右侧 `制导翻译语义动作`

除了制导翻译语义动作外，其它内容不允许换行。忽略空行和以#开头的行（作为注释）。

AST 动作的格式如下，

展开标记;保留节点标记

其中展开标记为*或者空。为*时代表，被规约出的节点在作为子节点插入其他 AST 节点的孩子列表时，将其展平，即将这个节点的孩子全部插入父节点的孩子列表，而忽略这个节点。如果为空，代表这个节点承载语义动作，如果这个节点是另一个节点的孩子，直接插入。

保留节点标记可以为空，代表将当前节点在对应的 AST 节点中，保持 CST 节点所有孩子对应的 AST 节点。如果为用逗号分隔的数字列表，代表只保留列表编号中的几个 CST 节点，将其转为 AST 节点后插入当前 AST 节点的列表。如果是-，代表不产生任何 AST 孩子节点。

AST 动作会在语法分析后执行。
对于 CFG 语法部分，在所有的文法变量可以任意命名，但要被双引号包裹。终结符有两种，可以是被单引号包裹的变量，其代表单引号中的字面量作为终结符，也可以是被尖括号包裹的特殊终结符，其内部字符会转义，例如换行符表示为<n>。
CFG 文法表示为变量->产生式右侧，产生式右侧可以是被竖线分隔的多个句型串。写在一起的多个产生式共享 AST 动作与制导翻译动作。
制导翻译动作就是 JS 代码，但是可以使用翻译时提供的$开头的部分函数执行属性计算，获取 AST 子节点等其它动作。

函数	参数	功能
$gather	Node, AttrName	将 Node 节点所有子节点数据字段的的所有 AttrNode 收集为一个数组
$gather_terminal	Node	将 Node 节点的所有子节点当作终结符合并成一个字符串
$str_to_uint	Digits	字符串转整数
$str_to_float	Digits	字符串转浮点数
$mktmp	TypeName	创建某类型的临时变量
$cltmp		从临时变量栈中删除一个
$clear_tmp		清空临时变量
$type_of_ret	Name	返回名为 name 函数的返回类型
$type_of_var	Name	返回名为 name 的变量的类型
$decl_var	Name, TypeName	在变量表中创建一个变量
$decl_func	Name, TypeName	在函数表中创建一个函数
$get_var_table		获取当前变量表
$mk_var_table		在当前变量表下，建立一个新的子变量表
$exit_var_table		从当前变量表退出到父变量表
$		获取当前产生式左侧规约出的节点
$	Idx	获取当前 AST 节点的第 Idx 个子节点
$$		获取当前节点的父节点（用于传递继承属性）

2. 文法设计

报告设计的文法能解析类似下面的这种高级语言，

```
func fib int {}

func fib(n int, cache int) int {
    if(*(cache + (n * 4)) != 0) {
        return *(cache + (n * 4));
    };
    if(n < 2) {
        cache + (n * 4) <- n;
        return n;
    } else {
        result := fib(n - 1, cache) + fib(n - 2, cache);
        cache + (n * 4) <- result;
        return result;
```

```
    };
    unreachable;
}

func main() void {
    cache malloc 1024 * 64;
    echo fib(40, cache);
    free cache;
    a := 1;
    while(a < 10) {
        a = a + 1;
    };
    echo a;
    return nil;
}
```

首先是一些基础的变量，例如类型名，字面量，变量名等。这一部分原则上应该在词法分析阶段完成。但为了方便，这里推后到语法分析阶段。

```
[*;] "letter" -> 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
| 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' |
'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' |
'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

[*;] "digit" -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
| '9'

[*;] "char_char" -> "letter" | "digit" | "escaped" | ',' | '.' | ';'
| ' ' | '(' | ')' | '+' | '-' | '!' | '~' | '#' | '{' | '}' | '='

[*;] "escaped" -> '\\' | '\"' | '\n' | '\t' | '\a'

[;] "uint_literal" -> "digits" `
const place = $mktmp("int");
$().d.code = {
    "op": "put",
    "lhs": $str_to_uint($(0).d.literal),
    "o": place,
}
$().d.val = {
    "v": place,
    "type_name": "int",
};
`

[;] "digits" -> "digits_wrapper" `
$().d.literal = $gather_terminal($())
`

[*;] "digits_wrapper" -> "digit" "digits_wrapper"
[*;] "digits_wrapper" -> "digit"

[;0,2] "float_literal" -> "digits" '.' "digits" `
const place = $mktmp("float");
$().d.code = {
    "op": "put",
    "lhs": $str_to_float($(0).d.literal+"."+$(1).d.literal),
    "o": place,
```

```
}
$().d.val = {
    "v": place,
    "type_name": "float",
};
`

[;] "bool_literal" -> 'true' `
const place = $mktmp("bool");
$().d.code = {
    "op": "put",
    "lhs": true,
    "o": place,
}
$().d.val = {
    "v": place,
    "type_name": "bool",
};
`
[;] "bool_literal" -> 'false' `
const place = $mktmp("bool");
$().d.code = {
    "op": "put",
    "lhs": false,
    "o": place,
}
$().d.val = {
    "v": place,
    "type_name": "bool",
};
`

[;1] "char_literal" -> <squot> "char_literal_body" <squot> `
const place = $mktmp("char");
$().d.code = {
    "op": "put",
    "lhs": $gather_terminal($()),
    "o": place,
}
$().d.val = {
    "v": place,
    "type_name": "char",
};
`

[*;] "char_literal_body" -> "char_char"

[;] "void_literal" -> 'nil' `
const place = $mktmp("void");
$().d.code = {
    "op": "put",
    "lhs": null,
    "o": place,
}
$().d.val = {
    "v": place,
    "type_name": "void",
};
```

```
`

[*;] "literal" -> "uint_literal" | "float_literal" | "bool_literal"
| "void_literal" | "char_literal"

[;] "id" -> "id_wrapper" `
$().d.name = "c_" + $gather_terminal($());
`
[*;] "id_wrapper" -> '_' | "letter" | "id_wrapper" '_' |
"id_wrapper" "digit" | "id_wrapper" "letter"

[;] "type" -> 'int' `
$().d.type_name = "int"
`
[;] "type" -> 'char' `
$().d.type_name = "char"
`
[;] "type" -> 'void' `
$().d.type_name = 'void'
`
[;] "type" -> 'float' `
$().d.type_name = 'float'
`
```

凡是类型名都带有 type_name 属性，凡是表达式都有 val 属性，这个表达式的结果类型和对应变量（即地址）。

put 操作将被翻译为常量创建与赋值。

所有用户创建的名称自动加上 c_前缀，以避免与编译过程产生的其它名称发生冲突。

整个程序分为一个或多个函数声明，

```
[;] "program" -> "func_decl_list" `
$().d.functions = $gather($(), "func_decl");
`

[*;] "func_decl_list" -> "func_decl_list" "func_decl" | "func_decl"

[;1,2] "func_decl" -> 'func' "id" "type" '{}'`
$decl_func($(0).d.name, $(1).d.type_name);
$().d.skip = true;
`

[;1,3] "func_decl" -> 'func' "id" '(' "func_decl_post" `
$decl_func($(0).d.name, $(1).d.return_type);
$().d.func_decl = {
    name: $(0).d.name,
    param: $(1).d.param,
    return_type: $(1).d.return_type,
};
$().d.var_table = $get_var_table();
$exit_var_table();
$clear_tmp()
`

[*;] "func_decl_post" -> "func_decl_post_with_param" |
"func_decl_post_no_param"
[;0,2] "func_decl_post_with_param" -> "param_list" ')'
"ret_type_and_func_body" `
$().d.param = $(0).d.param;
```

```
$().d.return_type = $(1).d.type_name;
`
[;1] "func_decl_post_no_param" -> ')' "ret_type_and_func_body" `
$().d.param = [];
$().d.return_type = $(0).d.type_name;
`
[;] "param_list" -> "param_list_wrapper" `
$().d.param = $gather($(), "one_param")
const param_table = $().d.param;
$mk_var_table();
if(param_table != undefined && param_table != null) {
    for(const param of param_table) {
        $decl_var(param.name, param.type_name);
    }
}
`
[*;0,2] "param_list_wrapper" -> "param_list_wrapper" ',' "one_param"
[*;] "param_list_wrapper" -> "one_param"

[;] "one_param" -> "id" "type" `
$().d.one_param = {
    "name": $(0).d.name,
    "type_name": $(1).d.type_name
};
`

[*;] "ret_type_and_func_body" -> "type" "block"
```

block 是一个被花括号包裹的语句块。程序设计的语句有以下几种，

名称	例子	说明
逗号语句	a=1,b=1	逐个语句翻译
声明语句	var a int	
赋值语句	a = 1	
While 语句	while(a>1){…}	
If 语句	if(a>1){…}else if(a<0){…}	可以有多个分支，有可选的 else
Echo 语句	echo a	打印某个表达式的值
Return 语句	return a	
表达式	test()	因为将函数调用作为了表达式的一种，因此将单个表达式也作为语句，以支持过程类函数调用
声明且赋值语句	a:=1	声明并赋值变量。变量类型自动推导为右侧表达式的类型
Unreachable 语句	unreachable	因为没有实现编译器后端，手动加入 unreachable 以对应 WASM 汇编器的检查
Pass 语句	pass	即空语句
Break 语句	break	
Continue 语句	continue	
Malloc 语句	x malloc 1024	分配动态内存
Free 语句	free x	释放动态内存

Store 语句	ptr <- a	给 ptr 地址所在的动态内存赋值

为了避免冲突，需要将逗号语句其它语句分级，降低它的优先级。

下面是语句的定义，

```
[;1] "block" -> '{' "stmt_list" '}'

[*;0,1] "stmt_list" -> "stmt_list" "stmt" ';'
[*;0] "stmt_list" -> "stmt" ';'

[*;] "stmt" -> "single_stmt" | "single_stmt" ',' "comma_stmt"

[*;] "comma_stmt" -> "single_stmt" | "single_stmt" ',' "comma_stmt"

[*;] "single_stmt" -> "decl_stmt" | "assign_stmt" | "while_stmt" |
"if_else_stmt" | "echo_stmt" | "return_stmt" | "expr_stmt" |
"decl_and_assign_stmt" | "unreachable_stmt"
[*;] "single_stmt" -> "pass_stmt" | "break_stmt" | "continue_stmt" |
"malloc_stmt" | "free_stmt" | "store_stmt"

[;0,2] "malloc_stmt" -> "id" 'malloc' "expr" `
$decl_var($(0).d.name, "int")
$().d.code = {
    "op": "malloc",
    "o": $(0).d.name,
    "val": $(1).d.val,
};
`

[;1] "free_stmt" -> 'free' "expr" `
$().d.code = {
    "op": "free",
    "val": $(0).d.val,
}
`

[;] "unreachable_stmt" -> 'unreachable' `
$().d.code = {
    "op": "unreachable",
};
`

[;] "expr_stmt" -> "expr" `
$().d.code = {
    "op": "expr"
}
`

[;] "pass_stmt" -> 'pass' `
$().d.code = {
    "op": "pass",
};
`
[;] "break_stmt" -> 'break' `
$().d.code = {
    "op": "break",
};
`
```

```
[;] "continue_stmt" -> 'continue' `
$().d.code = {
    "op": "continue",
};
`

[;1,2] "decl_stmt" -> 'var' "id" "type" `
$decl_var($(0).d.name, $(1).d.type_name);
$().d.code = {
    "op": "pass",
};
`

[;0,2] "assign_stmt" -> "id" '=' "expr" `
$().d.code = {
    "op": "=",
    "val": $(1).d.val,
    "o": $(0).d.name,
};
$cltmp();
`

[;0,2] "store_stmt" -> "expr" '<-' "expr" `
$().d.code = {
    "op": "store",
    "val": $(1).d.val,
    "o": $(0).d.val,
}
`

[;0,2] "decl_and_assign_stmt" -> "id" ':=' "expr" `
$decl_var($(0).d.name, $(1).d.val.type_name);
$().d.code = {
    "op": "=",
    "o": $(0).d.name,
    "val": $(1).d.val,
};
$cltmp();
`

[;1] "echo_stmt" -> 'echo' "expr" `
$().d.code = {
    "op": "echo",
    "val": $(0).d.val,
};
$cltmp();
`
[;1] "return_stmt" -> 'return' "expr" `
$().d.code = {
    "op": "return",
    "val": $(0).d.val
};
$cltmp();
`

[;2,4] "while_stmt" -> 'while' '(' "expr" ')' "block" `
$().d.code = {
    "op": "while",
```

```
    "cond": $(0).d.val,
    "code": 1,
};
$cltmp();
`

[;2,4] "if_else_stmt" -> 'if' '(' "expr" ')' "if_else_stmt_post" `
$().d.code = {
    "op": "if",
    "cond": $(0).d.val,
    "code_t": $(1).d.code_t,
    "code_f": $(1).d.code_f,
    "f_type": $(1).d.f_type,
};
$cltmp();
`

[;] "if_else_stmt_post" -> "block" `
$().d.code_t = 0;
$().d.code_f = -1;
$().d.f_type = "no";
`

[;0,2] "if_else_stmt_post" -> "block" 'else'
"if_else_stmt_post_else" `
$().d.code_t = 0;
$().d.code_f = 1;
$().d.f_type = $(1).d.f_type;
`

[;] "if_else_stmt_post_else" -> "block" `
$().d.f_type = "else";
`

[;] "if_else_stmt_post_else" -> "if_else_stmt" `
$().d.f_type = "nest";
`
```

下面要设计表达式，表达式有以下几种，按优先级从低到高，

表达式类型	说明
或表达式	
与表达式	
比较表达式	
移位表达式	
加法表达式	包含加，减，按位与，或
乘法表达式	包含乘除，取模
一元表达式	包括逻辑非，按位取反，正数符号，负数符号以及从地址取值
原始表达式	包含括号表达式，变量名自身作为一个表达式，字面量与函数调用

每个表达式都会创建一个临时变量存储自己的值，并且会释放其依赖的表达式的临时变量。

描述表达式的文法如下，

```
# -------- 表达式 --------
[*;] "expr" -> "or_expr"
```

```
[;] "or_expr" ->  "and_expr" '||' "or_expr" `
const place = $mktmp("bool");
$().d.code = {
    "op": "||",
    "lhs": 0,
    "rhs": 1,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": "bool",
};
$cltmp();
$cltmp();
`
[*;] "or_expr" -> "and_expr"

[;] "and_expr" -> "comp_expr" '&&' "and_expr" `
const place = $mktmp("bool");
$().d.code = {
    "op": "&&",
    "lhs": 0,
    "rhs": 1,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": "bool",
};
$cltmp();
$cltmp();
`
[*;] "and_expr" -> "comp_expr"

[;] "comp_expr" -> "add_expr" "comp_op" "comp_expr" `
const place = $mktmp("bool");
$().d.code = {
    "op": $(1).d.op_name,
    "lhs": 0,
    "rhs": 2,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": "bool",
};
$cltmp();
$cltmp();
`
[*;] "comp_expr" -> "shift_expr"

[;] "comp_op" -> '<' | '<=' | '>' | '>=' | '==' | '!=' `
$().d.op_name = $gather_terminal($());
`

[;] "shift_expr" -> "add_expr" "shift_op" "shift_expr" `
const place = $mktmp($(0).d.val.type_name);
$().d.code = {
```

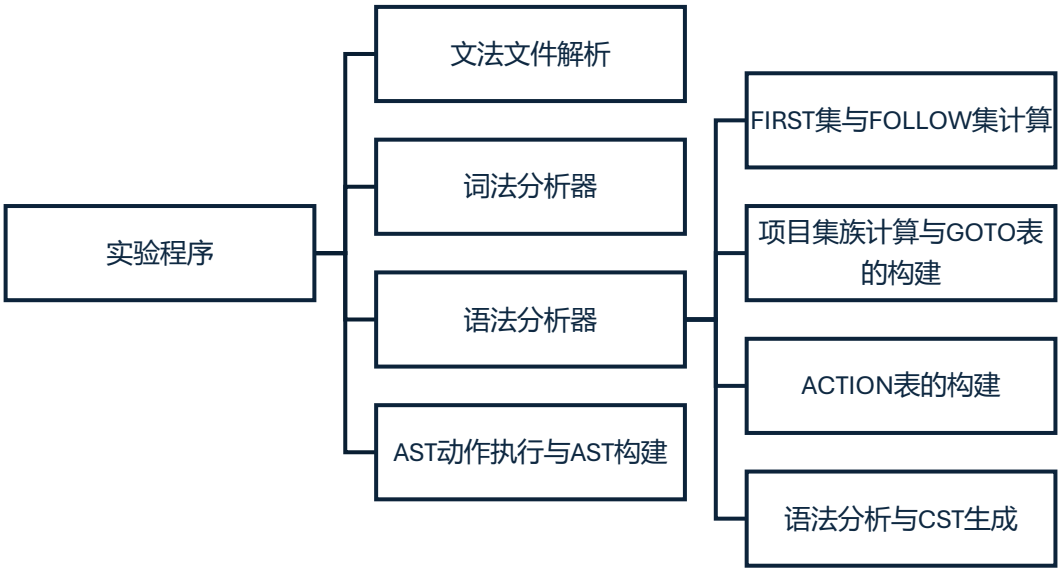
```
    "op": $(1).d.op_name,
    "lhs": 0,
    "rhs": 2,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(0).d.val.type_name,
};
$cltmp();
$cltmp();
`

[;] "shift_op" -> '<<' | '>>' `
$().d.op_name = $gather_terminal($());
`

[*;] "shift_expr" -> "add_expr"

[;] "add_expr" -> "mul_expr" "plus_or_minus_op" "add_expr" `
const place = $mktmp($(0).d.val.type_name);
$().d.code = {
    "op": $(1).d.op_name,
    "lhs": 0,
    "rhs": 2,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(0).d.val.type_name,
};
$cltmp();
$cltmp();
`
[;] "add_expr" -> "mul_expr" "bit_and_or_op" "add_expr" `
const place = $mktmp($(0).d.val.type_name);
$().d.code = {
    "op": $(1).d.op_name,
    "lhs": 0,
    "rhs": 2,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(0).d.val.type_name,
};
$cltmp();
$cltmp();
`

[*;] "add_expr" -> "mul_expr"

[;] "plus_or_minus_op" -> '+' | '-' `
$().d.op_name = $gather_terminal($());
`
[;] "bit_and_or_op" -> '&' | '|' `
$().d.op_name = $gather_terminal($());
`
```

```
[;] "mul_expr" -> "unary_expr" "mul_div_or_mod_op" "mul_expr" `
const place = $mktmp($(0).d.val.type_name);
$().d.code = {
    "op": $(1).d.op_name,
    "lhs": 0,
    "rhs": 2,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(0).d.val.type_name,
};
$cltmp();
$cltmp();
`
[*;] "mul_expr" -> "unary_expr"

[;] "mul_div_or_mod_op" -> '*' | '/' | '%' `
$().d.op_name = $gather_terminal($());
`

[;] "unary_expr" -> "unary_op" "primary_expr" `
const place = $mktmp($(1).d.val.type_name);
$().d.code = {
    "op": $(0).d.op_name,
    "lhs": 1,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(1).d.val.type_name
}
$cltmp();
`
[*;] "unary_expr" -> "primary_expr"

[;] "unary_op" -> '+' | '-' | '!' | '~' | '*' `
$().d.op_name = $gather_terminal($());
`

# -------- primary expression --------
[;1] "primary_expr" -> '(' "expr" ')' `
const place = $mktmp($(0).d.val.type_name);
$().d.code = {
    "op": "=",
    "lhs": 0,
    "o": place,
};
$().d.val = {
    "v": place,
    "type_name": $(0).d.val.type_name
}
$cltmp();
`
[;] "primary_expr" -> "id" `
const place = $mktmp($type_of_var($(0).d.name));
$().d.val = {
```

```
    "v": place,
    "type_name": $type_of_var($(0).d.name)
};
$().d.code = {
    "op": "get",
    "lhs": $(0).d.name,
    "o": place,
}
`

[;] "primary_expr" -> "literal" `
const place = $mktmp($(0).d.val.type_name);
$().d.val = $(0).d.val
$().d.code = {
    "op": "get_literal",
    "lhs": $(0).d.val.v,
    "o": place,
}
$cltmp();
`

[*;] "primary_expr" -> "call_expr"

# -------- 函数调用 --------
[;0,2] "call_expr" -> "id" '(' "call_expr_post" `
const place = $mktmp($type_of_ret($(0).d.name));
$().d.val = {
    "v": place,
    "type_name": $type_of_ret($(0).d.name),
}
$().d.code = {
    "op": "call",
    "o": place,
    "args": $(1).d.args,
    "name": $(0).d.name,
}
`

[*;] "call_expr_post" -> "call_expr_post_with_args" |
"call_expr_post_no_args"
[;0] "call_expr_post_with_args" -> "args" ')' `
$().d.args = $(0).d.args;
`
[;-] "call_expr_post_no_args" -> ')' `
$().d.args = [];
`

[;] "args" -> "args_wrapper" `
$().d.args = $gather($(), "val");
`
[*;0,2] "args_wrapper" -> "expr" ',' "args_wrapper"
[*;] "args_wrapper" -> "expr"
```

3. 实验程序的结构与实现

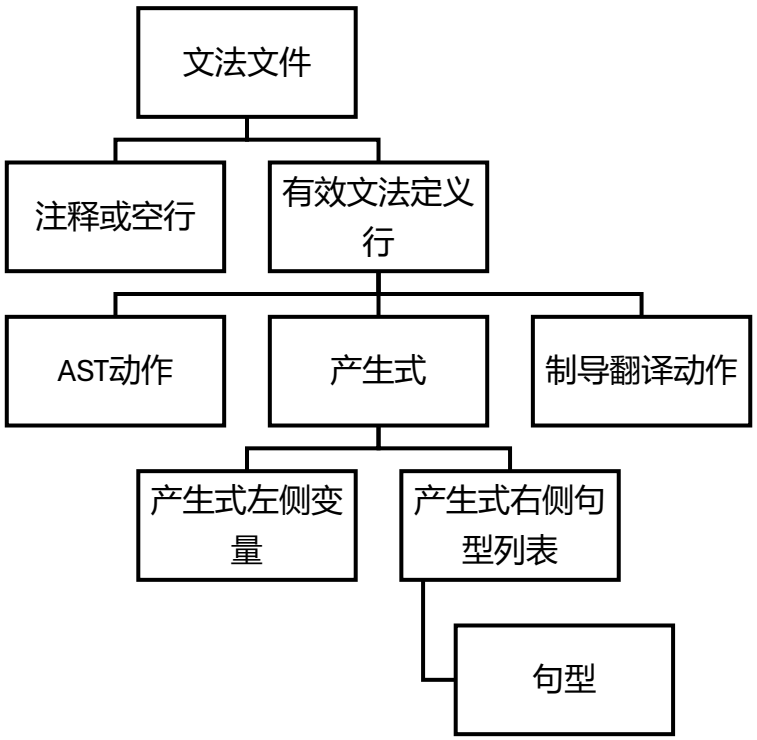
实验程序的整体结构如下（仅展示报告要求的部分），



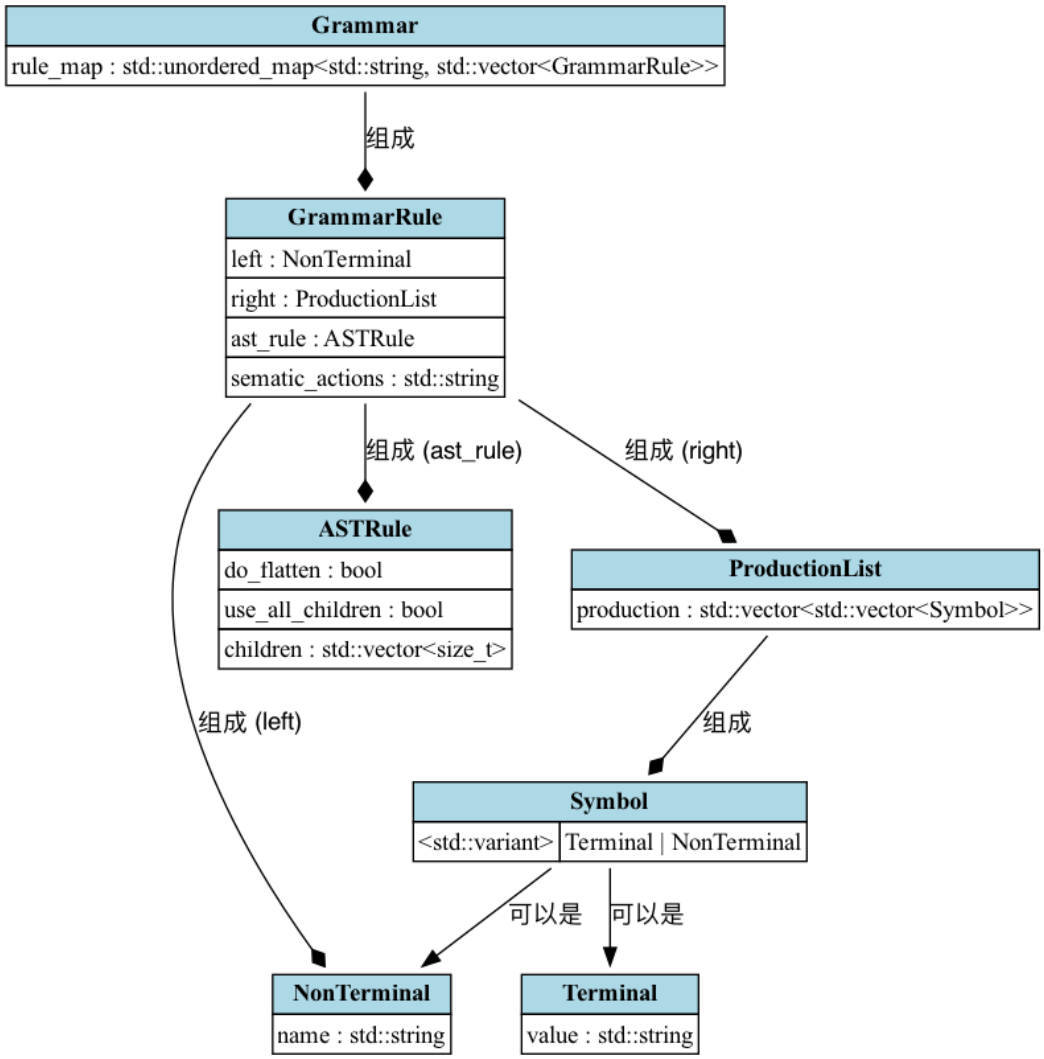
下面将分部介绍程序的实现。

1) 文法文件解析

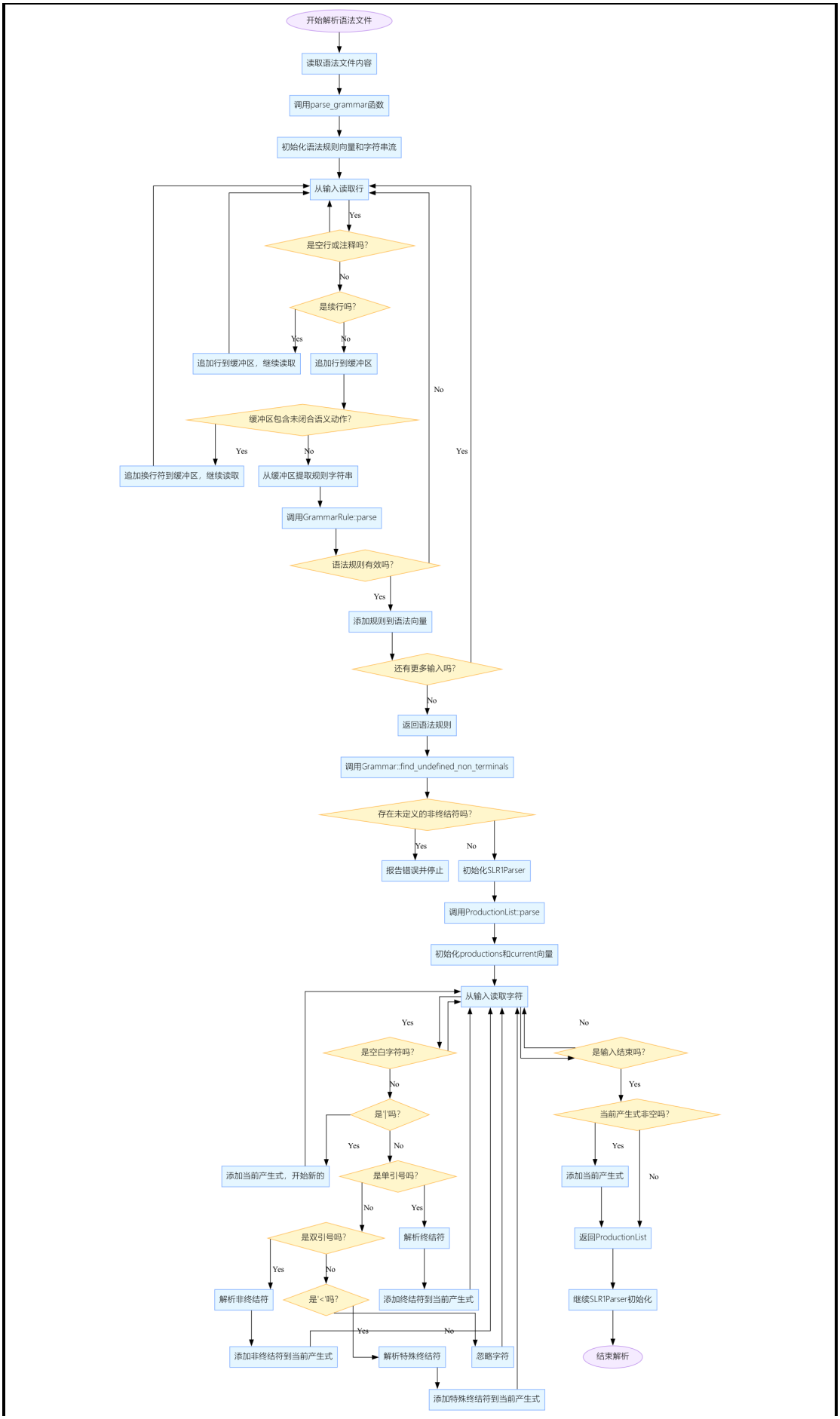
文法文件的设计已经在上文中第一部分说明。显然这是一种正则语言。不过为了方便实现，这里使用确定性递归下降法解析。
文法文件不同成分的推导关系如下，



递归下降方法会为每一个非终结符创建一个函数用于解析。解析的目标 C++结构如下图所示，



逐步使用递归下降方法进行解析，即可实现文法文件的解析。因为文法文件是正则语言，因此不需要回溯，递归下降是确定的。

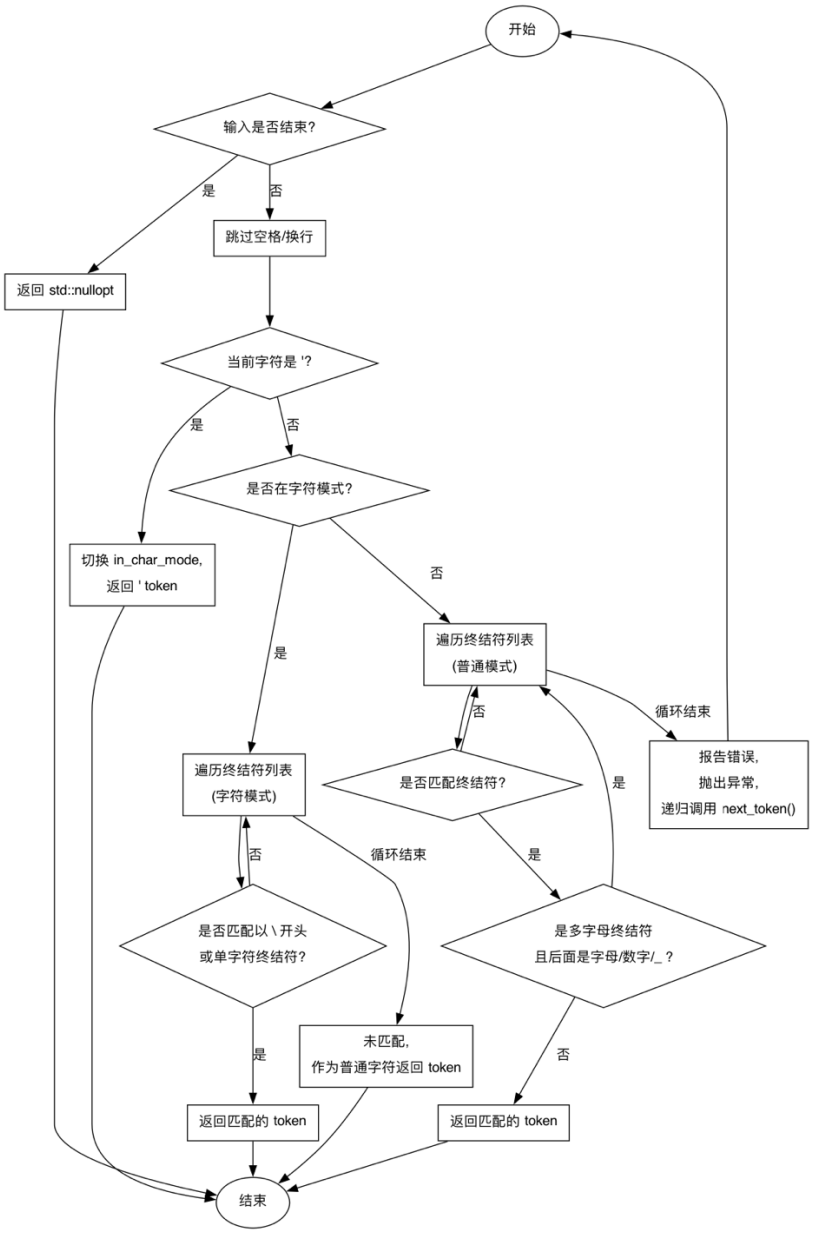

开始解析语法文件
读取语法文件内容
调用parse_grammar函数
初始化语法规则向量和字符串流
从输入读取行
是空行或注释吗?
Yes
No
是续行吗?
Yes
No
追加行到缓冲区，继续读取
追加行到缓冲区
缓冲区包含未闭合语义动作?
Yes
No
追加换行符到缓冲区，继续读取
从缓冲区提取规则字符串
调用GrammarRule::parse
语法规则有效吗?
Yes
No
Yes
添加规则到语法向量
还有更多输入吗?
No
返回语法规则
调用Grammar::find_undefined_non_terminals
存在未定义的非终结符吗?
Yes
No
报告错误并停止
初始化SLR1Parser
调用ProductionList::parse
初始化productions和current向量
从输入读取字符
Yes
No
是空白字符吗?
是输入结束吗?
No
Yes
是'|'吗?
当前产生式非空吗?
Yes
No
Yes
No
添加当前产生式，开始新的
是单引号吗?
Yes
No
添加当前产生式
No
No
Yes
是双引号吗?
解析终结符
返回ProductionList
Yes
No
解析非终结符
是'<'吗?
添加终结符到当前产生式
继续SLR1Parser初始化
Yes
No
添加非终结符到当前产生式
解析特殊终结符
忽略字符
结束解析
添加特殊终结符到当前产生式

这一步中，可以从上文中的文法定义文件得到C++的文法类。

2) 词法分析

词法分析需要将原本的输入文本拆分成非终结符。一般会使用正则语言描述构词方法，然后使用正则表达式处理。另一种更简单的方法是使用贪心匹配的方法，从当前位置开始，找到长度最长且匹配的终结符号。但这种简单的方法有两个问题：它无法区分全字母的关键字和标识符，也无法区分某个符号是全字母的关键字还是字符类型的字面量，这里在词法分析器上为这一个语法开洞处理：当匹配到全字母的终结符时，检查如果进行匹配，匹配后的后一个字符是否允许在标识符中出现。如果是，则认为当前匹配的部分属于标识符的一部分，因而跳过这个终结符的匹配。对于字符类型的匹配，当词法分析器识别到单引号，进入字符模式，只匹配允许的字符或转义字符。综上，经过这两个特殊处理，就能正确地将程序转化为标记。

具体而言，词法分析需要先从文法中获取所有的终结符，然后执行匹配算法。匹配的逻辑如下图所示，

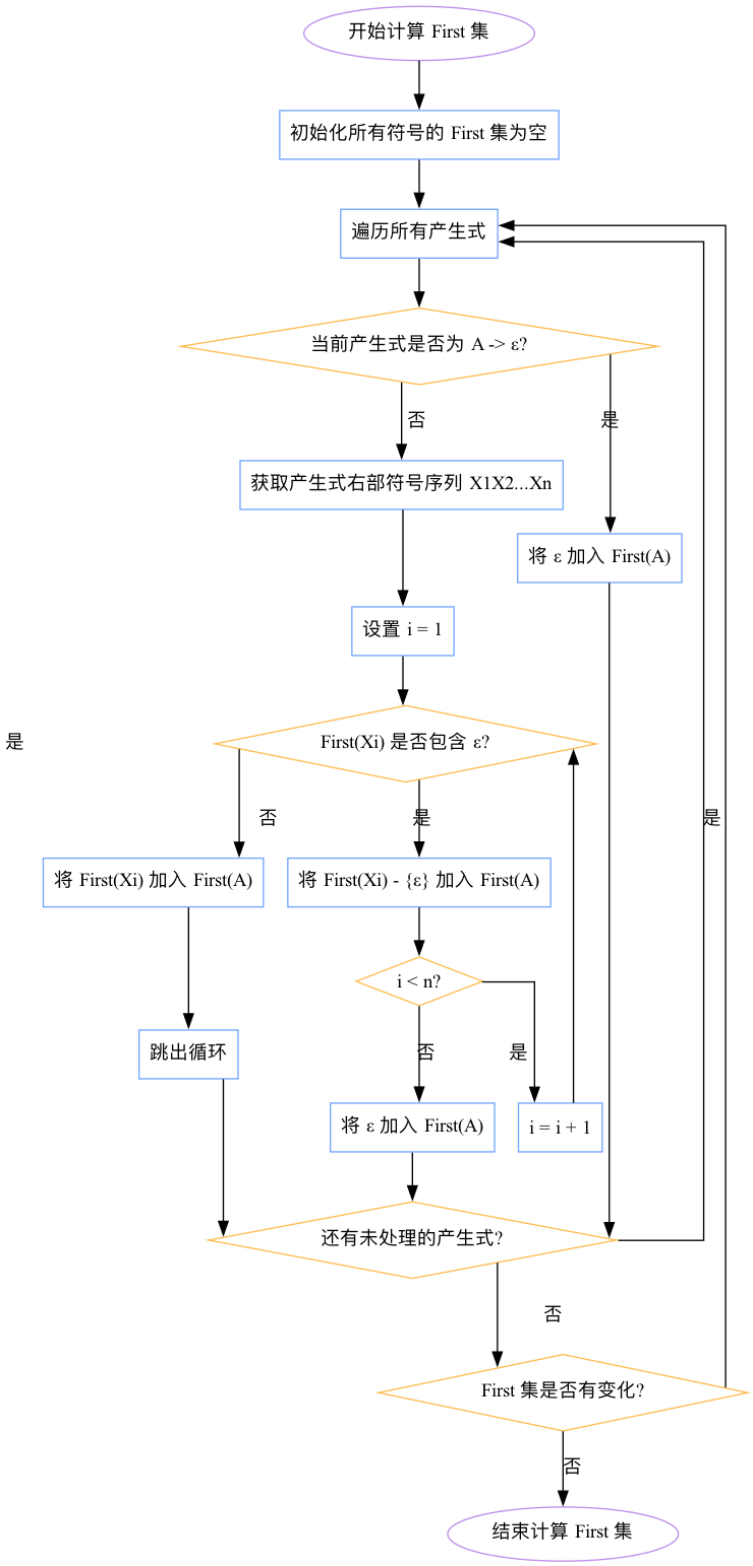


综上，完成了词法分析。

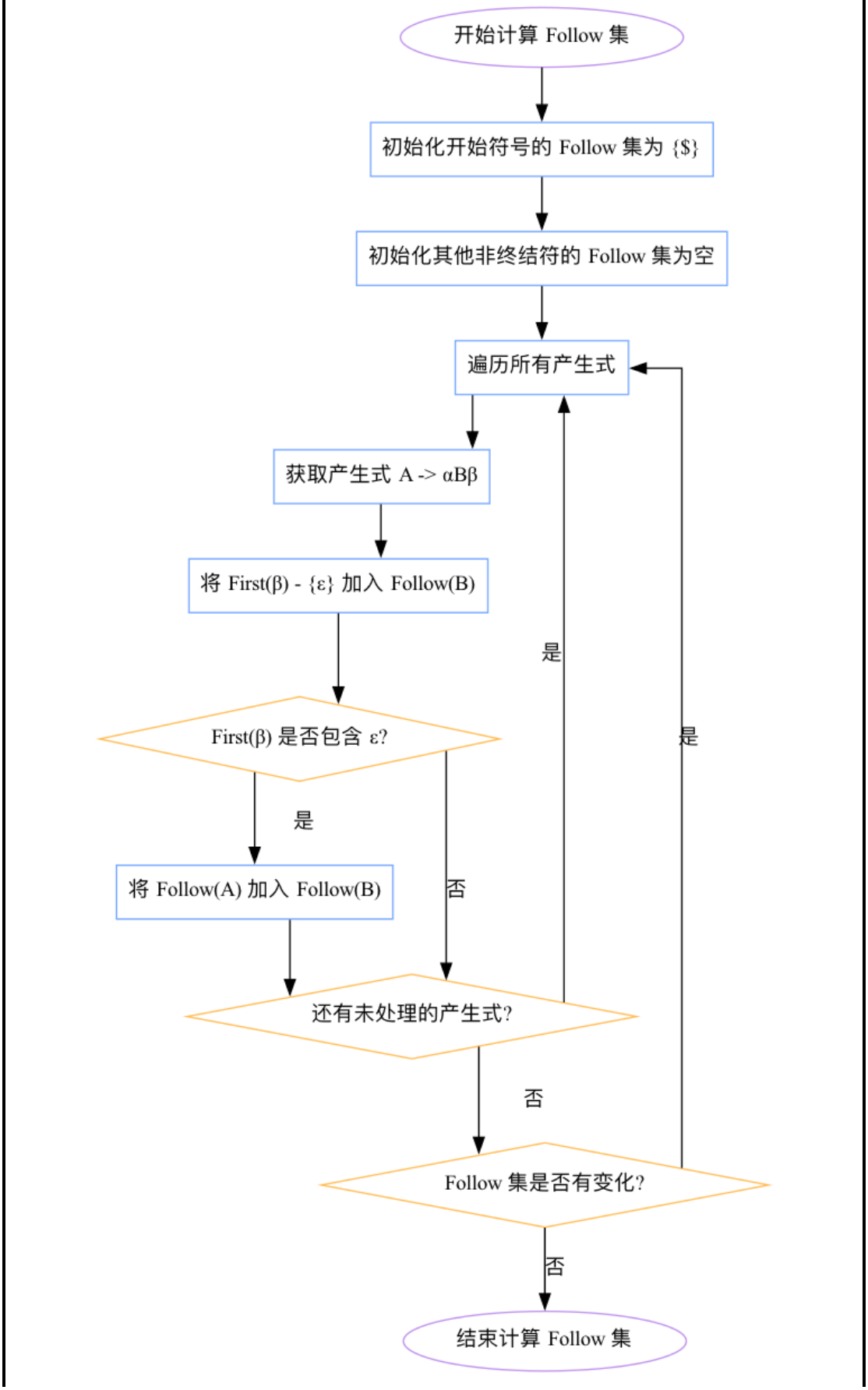
3) FIRST 集与 FOLLOW 集计算

这里使用标准的算法进行。不过在进入这一步时，会把前文的产生式拆成单个，以方便处理。

以下是计算 FIRST 集的流程图。

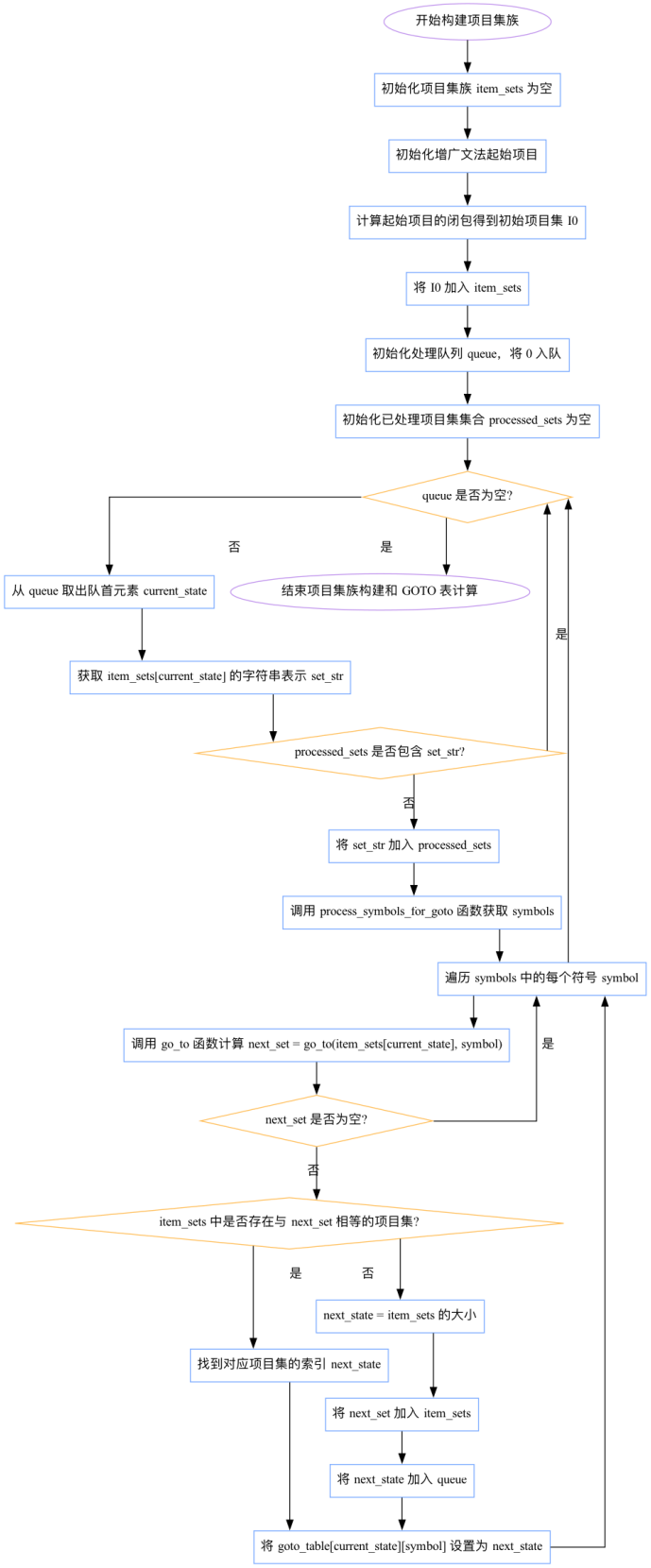


以下是计算 FOLLOW 集的流程图，

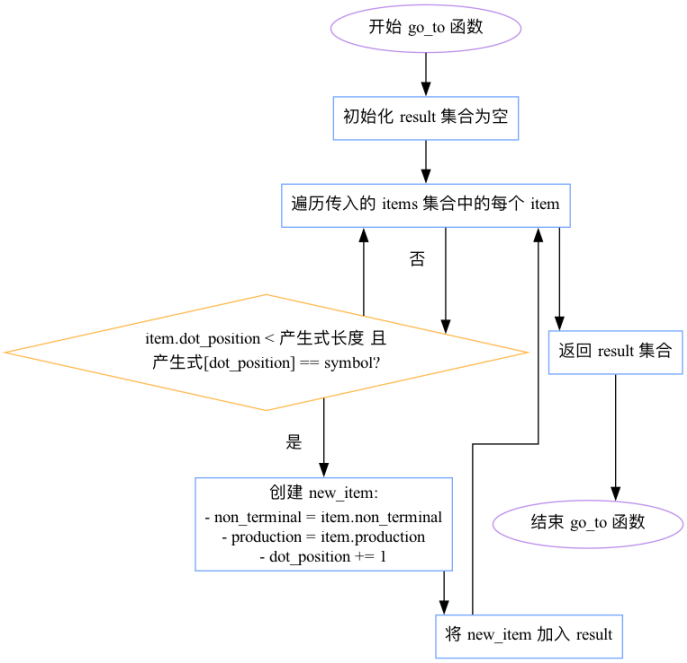


4) 项目集族的计算与 GOTO 表的计算

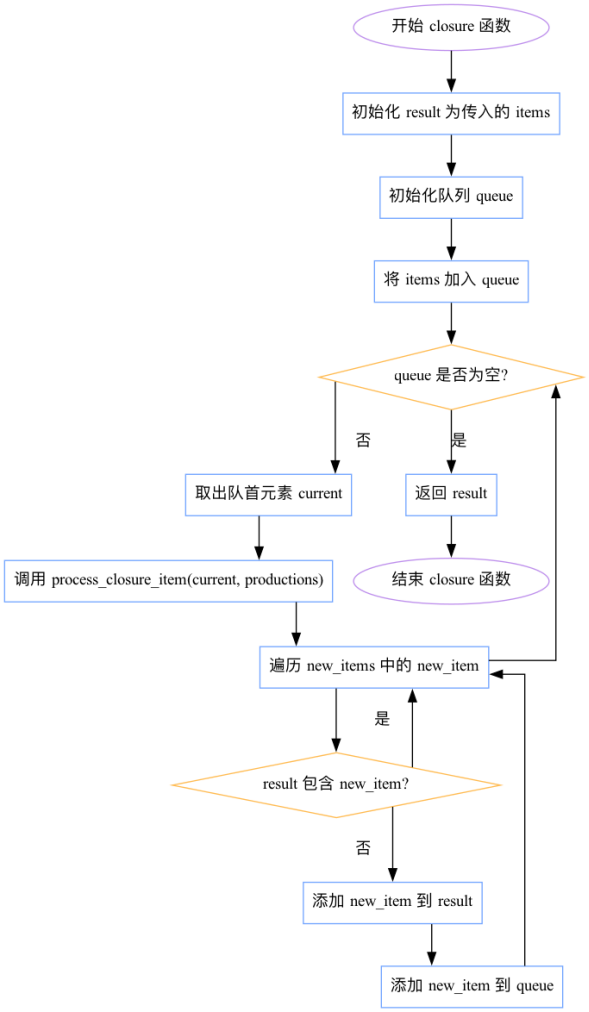
下一步是构建活前缀识别 DFA。这里使用广度优先搜索的方法。首先是计算所有的项目集族，并在这个过程中将 GOTO 表构建出来。流程图如下，



这里需要一个 go_to 函数。这个函数会计算再输入一个符号，会得到的新的项目集闭包。闭包算法和 go_to 函数的流程图如下，

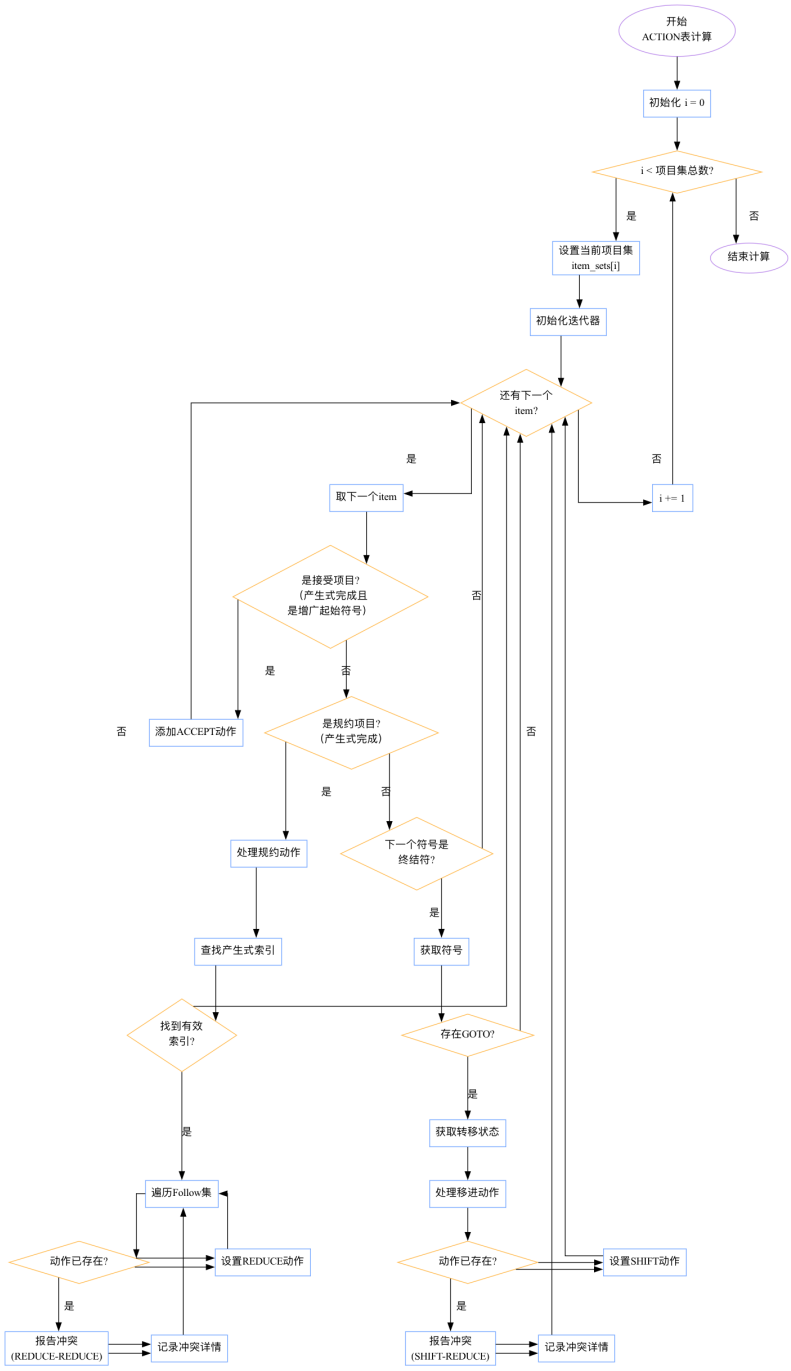


这里需要闭包计算，



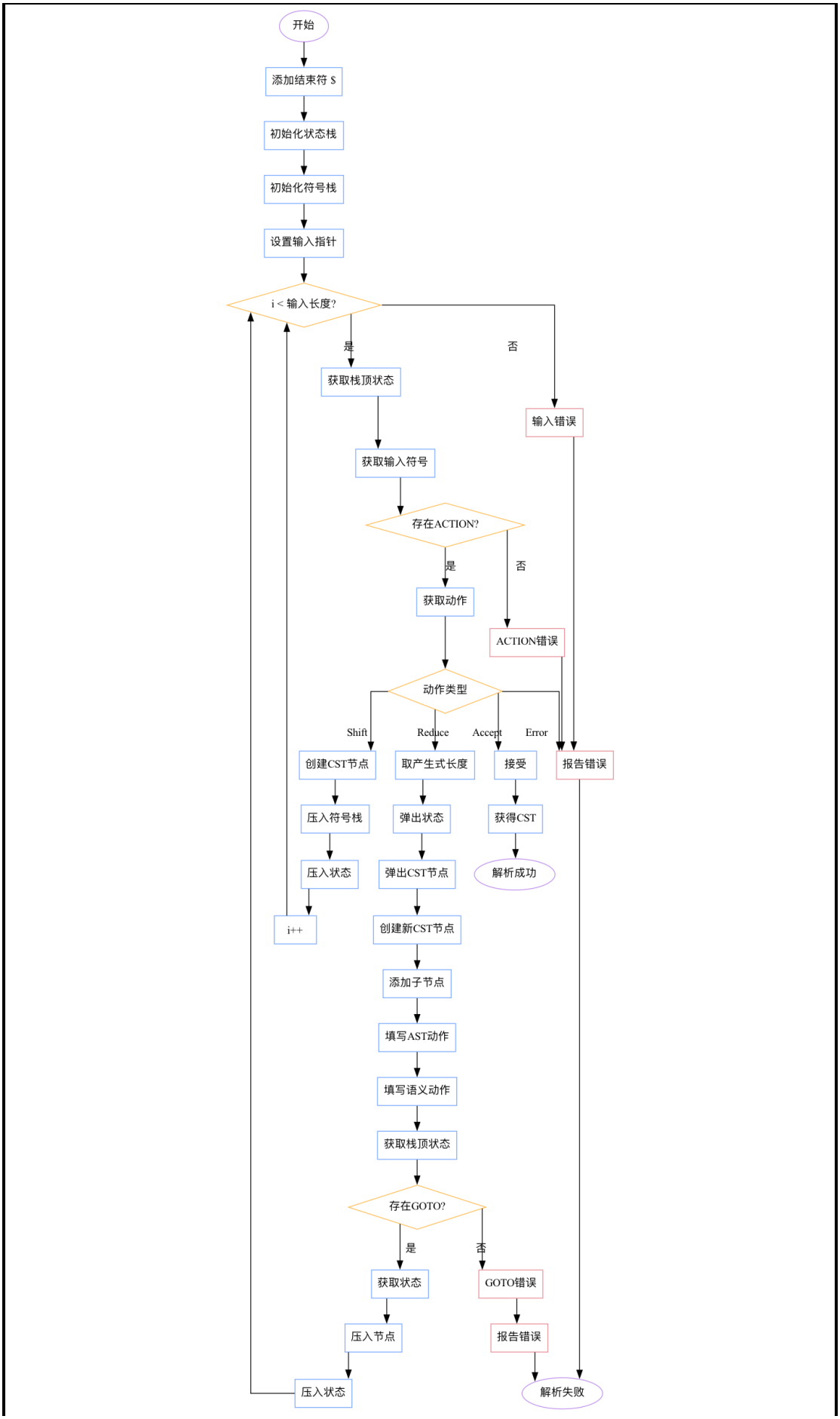
5) ACTION 表的计算

现在遍历每个项目集，填写规约，移进与接受动作。



6) 语法分析与 CST 的生成

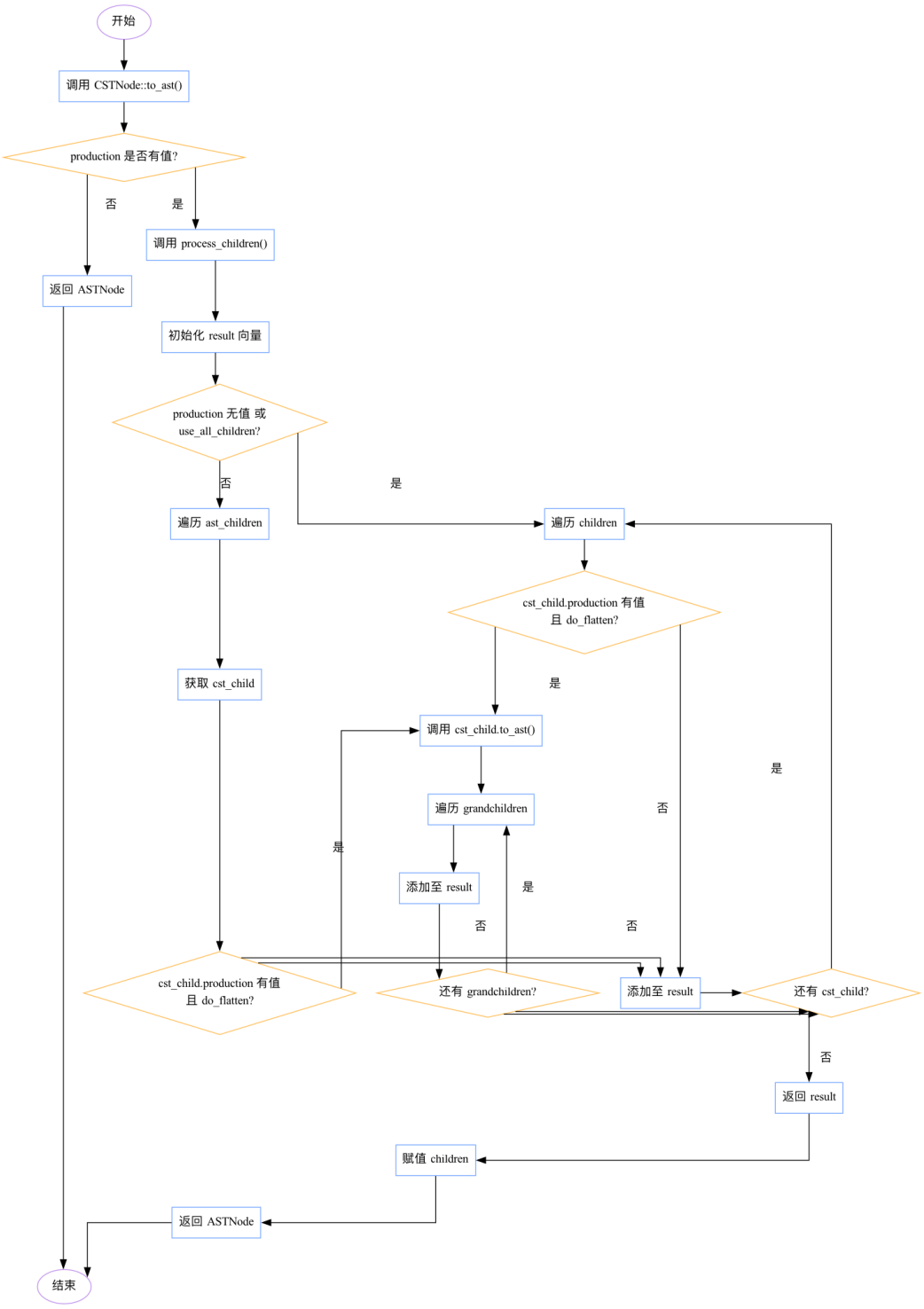
使用符号栈与状态栈法进行解析。因为需要给出 CST，所以需要在符号栈实际存放的是 CST 节点。开始状态压栈时，压入带有非终结符的节点。当执行规约时，弹出被规约的句型，然后创建一个新的 CST 节点作为被弹出的规约节点的父节点，节点的值是规约出的非终结符，然后将这个新节点压回栈中。此外，还要向节点添加 AST 动作与语义动作（基于规约规则）。不断进行规约，最后只剩下开始节点。这个开始节点即是完整的 CST。


开始
添加结束符 $
初始化状态栈
初始化符号栈
设置输入指针
i < 输入长度?
是
否
获取栈顶状态
输入错误
获取输入符号
存在ACTION?
是
否
获取动作
ACTION错误
动作类型
Shift
Reduce
Accept
Error
创建CST节点
取产生式长度
接受
报告错误
压入符号栈
弹出状态
获得CST
压入状态
弹出CST节点
解析成功
i++
创建新CST节点
添加子节点
填写AST动作
填写语义动作
获取栈顶状态
存在GOTO?
是
否
获取状态
GOTO错误
压入节点
报告错误
压入状态
解析失败

7) AST 动作执行与 AST 构建

前文所设计的 AST 动作就是一种简单的语义动作。如果将 CST 节点的对应 AST 树视为属性的话，因为这个属性要么是固有属性（即终结符的 AST 可以直接确定），要么是综合属性（一个非终结符的 AST 是由其子节点的 AST 按一定规则计算得到的），因此这个属性是 S 属性，使用简单的后续遍历即可。

这里的后续遍历使用递归实现，递归后执行 AST 计算动作即可。将一个 CST 节点转化为 AST 节点的流程图如下，



4. 错误处理

前文中，报告已经展示的内容有：语言成分的文法描述，语法分析程序的总体结构及物理实现，语法分析表及其数据结构和查找算法，语法分析表的生成算法。尽管前文已经包含了错误处理的部分内容，这一部分将集中展示程序的错误处理机制。

1) 解析文法的错误处理

解析文法使用了递归下降法，因此很容易得知哪一部分出错。出错时，没有递归子解析器能解析剩余内容，此时直接打印出错行并终止。

2) 未定义非终结符的错误处理

文法文件中可能存在未定义的非终结符符，这是不合法的。程序通过一个简单的方法进行检查：首先遍历所有的产生式，将产生式右侧出现的所有非终结符加入到一个集合中。然后进行二次遍历，从集合中删除每个产生式左侧的非终结符。如果每个终结符都有定义，这个集合应该为空，否则这个集合里的终结符就是未定义的非终结符。
在解析完文法文件后，会做这样的检查。如果有未定义的非终结符则终止分析。

3) SLR(1)冲突的错误处理

SLR(1)规约表生成时发生冲突意味着文法有问题，直接中断分析即可。为了方便 debug，会打印出冲突的具体单元格。

4) 查表出错的处理

查表出错意味着当前位置出现了无法被识别的符号。此时会遍历当前状态的 action 表，输出有非报错动作的终结符，即期待在当前位置出现的终结符。然后打印当前位置的相关信息。

三、实验结果

要求：将实验获得的结果进行描述，基本内容包括：
（1） 针对一测试程序输出其语法分析结果；
（2） 输出针对此测试程序对应的语法错误报告；
注：其中的测试样例需先用已编写的词法分析程序进行处理。

四、实验中遇到的问题总结

要求：主要阐述两方面的问题

（一） 实验过程中遇到的问题如何解决的？

着重从实验内容的实现、上机实践以及结果分析方面进行阐述。

（二）思考题的思考与分析

思考题 1：给出在生成语法分析表时所遇到的困难，以及是如何处理的？

思考题 2：思考还可以什么形式来给出语法分析的结果？

思考题 3：如果在语法分析中遇到了语法错误，是应该中断语法分析呢，还是应该进行适当处理后继续语法分析，你是怎么处理的？

五、实验体会

指导教师评语：

日期：