

东南大学

编译原理课程设计

设计报告

组长： 09019204 曹邹颖

成员： 09019231 许志豪

09019104 陈逸彤

东南大学计算机科学与工程学院

二〇22年4月

设计任务名称		SeuYacc	
完成时间	2020-05-04	验收时间	2020-05-22
本组成员情况			
学 号	姓 名	承 担 的 任 务	成 绩
09019204	曹邹颖	1. Yacc输入文件的读入与解析 2. 符号表的构建与相关管理程序 3. 语义动作程序的加入：类型检查、中间代码生成 4. 语法分析器代码Yacc.cpp生成 5. 整体Yacc+Lex联合测试Debug 6. 解析所有产生式，完成可视化 7. 报告的前三部分、第六部分	
09019231	许志豪	1. 构造LR(1)项目集的算法(包含函数closure、计算预测符、计算First集的算法) 2. 项目集的简化算法 3. LR(1)项目集的打印 4. 报告第三、四、六部分	
09019104	陈逸彤	1. 构造LR(1)项目集规范族的算法 2. 构造LR(1)分析表的算法 3. LR(1)分析表的打印功能 4. 报告的第三、五、六部分	

1 编译对象与编译功能

1.1 编译对象

我们以c99.y文件为标准，将文件的格式修改呈现如下：

- 说明部分

%{

头文件表、宏定义、数据类型定义、全局变量定义

%}

语法开始符号定义、语义值类型定义、终结符号定义、运算符优先级及结合性定义

%%

- 语法规则部分

%%

- 用户子程序部分 (包含主程序、错误信息报告程序、词法分析程序、其它程序段)

原c99.y说明部分内没有%{...%}内容，为了便于分析，我们补充了这一部分，只不过内容为空而已，同时以“%start”标记的文法开始符号是translation_unit，为了省去在语法规则部分检索文法开始符号的步骤，我们与原版Yacc的行为保持一致，如果定义段中没有“%start”的说明，Yacc自动将语法规则部分中第一条语法规则左部的非终结符作为语法开始符，从而我们删去说明部分“%start”的定义说明，将语法规则中

translation_unit: external_declaration

| translation_unit external_declaration;

这一条语法规则移至第一条位置，以此表示translation_unit为文法开始符号。

另外，原c99.y说明部分没有左右结合关系声明，所以我们经网上查阅，整理了C运算符优先级与左右结合关系表，添加到其中，以便解决结合性和优先级冲突。最后，直接将c99.y作为编译对象，即使程序经过一定优化，也需耗时很久，但是最终测试只需要用输出Yacc.cpp即可。

具体请参考 ../SEULex/c99_1.y。

1.2 编译功能

项目整体功能包括：

① c99.y输入文件的解析，主要在analysisInput()函数中完成，其中调用了

ExplanationProcess()函数完成说明部分%{...%}解析，SymbolProcess()函数完成说明部分%开头标记的终结符与非终结符的解析，RuleProcess()函数解析语法规则部分，ProducerProcess()函数处理规则部分的产生式，AdditionalProgramProcess()函数处理附加程序段；

② 灵活使用set, map, vector等数据结构，存储符号、产生式等；

③ 构造LR(1)项目集，包含LR1Item类设计，closure()函数求闭包、findPredictiveSym()函数计算预测符、getFirstSet()函数计算First集；

④ 构造LR(1)项目集规范族的详细设计，主要在ProcessItemSet()函数中完成；

⑤ 语法分析器C++代码的生成，主要在GenerateCode()函数中完成。

项目还有一些特色功能，如LR(1)分析过程的可视化（包含所有产生式、终结符、非终结符、LR(1)项目集、分析表打印等）、C++代码的美化、C++代码的自动编译等。

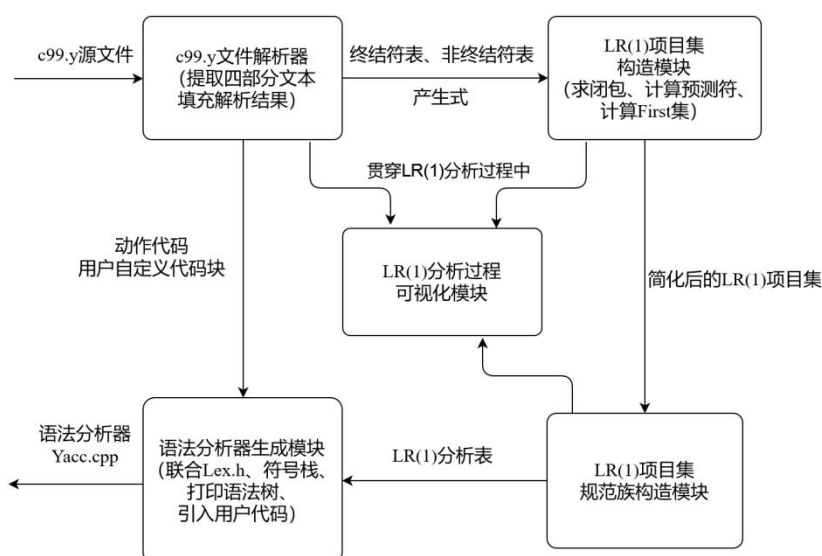
2 主要特色

- ① .y 文件头部声明支持%token、%type、%left、%right、%union
- ② 动作代码部分可以使用\$\$、\$1、\$2、\$3 等获取栈内元素
- ③ 广泛使用接口与面向对象编程，代码各部分独立，从而易于调用
- ④ 代码具备一定的错误处理能力
- ⑤ 在进行 LR(1)分析时，对符号（终结符、非终结符）、产生式、LR(1)项目集等全部进行编号，利用编号索引，节省空间，效率较高
- ⑥ 对 LR(1)项目集进行了简化算法
- ⑦ 对构造ACTION-GOTO分析表时，会处理可能出现的归约-归约冲突与移进-归约冲突，以保证语法分析不被打断
- ⑧ 提供了所有产生式可视化、终结符表与非终结符的打印功能，ACTION-GOTO分析表可视化、语法树可视化功能
- ⑨ 生成 LR1 语法分析表时，求取 GOTO 时采用了空间换时间的缓存技术，避免重复求取GOTO(I,X)
- ⑩ 提供了简易的符号表功能，用户可以在动作代码内新增或更新指定类型的符号元素，这为中间代码生成打下基础

3 概要设计与详细设计

3.1 概要设计

SeuYacc 分为 .y 文件解析器、LR(1)项目集构造模块、LR(1)项目集规范族构造模块、语法分析器生成模块、LR(1)分析过程可视化模块。模块之间的关系和职责可以用如下的工作流图来表示：



3.2 详细设计

3.2.1 C99.y文件解析的详细设计

Yacc输入文件解析器需要解析的c99.y文件主要分为四个部分：一，包含头文件表、宏定义、数据类型定义、全局变量定义的说明部分（以%{开头，以%}结尾）；二，包含语法开始符号定义、语义值类型定义、终结符号定义、运算符优先级及结合性定义的部分（在第一个%%之前）；三，语法规则部分，即产生式-动作部分（起始于第一个%%，终止于第二个%%）；四，用户子程序部分（第二个%%之后），包括主程序、错误信息报告程序、词法分析程序、其它程序段。对于第二部分需要先读到第一个空格判断此定义是有关终结符的还是非终结符的亦或语义值类型还是运算符优先级或结合性的，之后再读取剩下整行内容，通过特别字符做分割进行定义解析，其余部分（第一、三、四部分）可以通过对文件内容逐行扫描。注意：解析的同时要判断c99.y文件的结构是否符合要求，如果结构不正确需要报错处理。

对于第一和第四部分，处理方式简单：直接读入并存入Yacc.cpp即可。

对于第二部分，先读到第一个空格判断此定义是有关哪一内容，“%token”代表定义的是终结符+语义值类型，“%type”代表定义的是非终结符+语义值类型，“%left”与“%right”是有关运算符结合性与优先级的，最后使用“%union”定义来使用其它的数据类型，如struct结构体。接下来，再逐行读取剩下整行内容，对于终结符/非终结符，输入文档中用“ \diamond ”标记语义值类型声明，如果没有规定语义值类型则置为空，之后逐空格截取字符读取每个终结符/非终结符的定义，给其编号并用相应的map存储符号与编号一一对应。在解析运算符结合性声明部分（%left、%right）时，遵循同一行的运算符具有相同的优先级，越后声明的运算符优先级越高的策略确定优先级关系，同时操作符用‘’标记从而能从字符串中截取出来，也作为终结符编号存入终结符map中，根据操作符结合性也相应地存入左右结合表（set数据结构）中。使用%union定义的其它数据类型需要存入yyval.h头文件中，并且加上定义语言直接读入Yacc.cpp中。注意：将#号加入终结符表。

对于第三部分，因为每行的语法规则构成是产生式+动作，每一个语法规则的左部后面会有若干数量不一的候选式，换行由“|”分隔开，我们用Producer结构体存储，先读取规则左部，理应存在非终结符表中，然后逐字符读取右部的候选式，将对应字符串序号存入产生式右部，每一个候选式识别完毕后，会检测是否存在语法动作（以{}标记），完毕后存入producerVector, producerActionVector, producerPriorVector这三个vector数据结构中对应产生式、产生式动作与产生式优先级的存储。注意：在读取文件所有产生式定义前，要预先加入产生式 $S' \rightarrow S$ ，以便产生拓广文法。

最后对于所有解析出来的产生式再做一次整合，填充入名为producerMap的map<int, vector<int>>结构中，以产生式的左部符号编号为key，收集对应候选式的所有编号装入vector中，供下层使用。

使用的变量
<pre>set<int> leftTable; // 左结合表 set<int> rightTable; // 右结合表 map<string, int> nonterminalTable; // 非终结符表 符号映射至数字编号 map<string, int> terminatorTable; // 终结符表 符号映射至数字编号 map<int, int> PriorityTable; // 优先级表, 前一项编码后一项优先级 map<int, string> tokenTypeTable; // 存储语义值类型的表, 前一项编码后一项语义值类型 struct Producer//产生式数据结构 { int left; // 语法规则的左部符号编号 vector<int> right; // 右部候选式中符号编号排列 }; vector<Producer> producerVector; // 存储所有的产生式 vector<int> producerPriorVector; // 存储所有的产生式优先级 vector<string> producerActionVector; // 存储所有产生式的动作 // 终结符<数字编号,在分析表中的列号>,第二项为了构造LR(1)分析表 map<int, int> terminSet; // 非终结符<数字编号,在分析表中的列号>,并同时存储一个到action表头的对应关系 map<int, int> nonterminSet; // 以产生式的左部为关键字, 以对应产生式的编号为内容的vector map<int, vector<int>> producerMap;</pre>
<p>3.2.2 构造LR(1)项目集的详细设计, 包含LR1Item类设计, closure()函数、findPredictiveSym()函数、getFirstSet()函数</p> <p>1. closure()函数</p> <p>(1) 构造closure()函数的算法, 我们采用的算法依据的是上学期编译原理课上用的中文教材《编译原理及编译程序构造》中第6.4.1章“构造LR(1)项目集规范族的算法”中的closure部分, 具体思路如下:</p> <p style="text-align: center;">构造 LR(1)项目集规范族的算法</p> <p>构造 LR(1)项目集族的算法本质上和构造 LR(0)项目集族的算法是一样的, 先介绍两个函数: CLOSURE 和 GO。</p> <p>(1) 函数 CLOSURE(I) — I 的项目集。</p> <p>① I 的任何项目都属于 CLOSURE(I);</p> <p>② 若项目 $(A \rightarrow \alpha \cdot B\beta, a)$ 属于 CLOSURE(I), $B \rightarrow \gamma$ 是一个产生式, 那么对于 FIRST(βa) 中每个终结符 b, 如果 $(B \rightarrow \cdot \gamma, b)$ 原来不在 CLOSURE(I) 中, 则把它加进去;</p> <p>③ 重复步骤②, 直至 CLOSURE(I) 不再扩大为止。</p> <p>因为 $(A \rightarrow \alpha \cdot B\beta, a)$ 属于 CLOSURE(I), 那么 $(B \rightarrow \cdot \gamma, b)$ 当然也属于 CLOSURE(I), 其中 b 必定是跟在 B 后面的终结符, 即 $b \in \text{FIRST}(\beta a)$, 若 $\beta = \epsilon$, 则 $b = a$。</p> <p>(2) 具体代码设计</p> <p>① 代码的传入参数是一个 LR1Item 的 set, 也就是一个集合。而且传引用, 也就是说最终在函数里对于 set 的修改会落实到 set 的实际变化中。</p>

② 代码首先获得一个项目集里的所有的已有的项目，利用一个LR1Item类型的queue来存该项目集里的所有的项目，然后利用循环遍历itemSet的LR1Item，首先取得所有的已有项目放入queue中。

③ 利用findPredictiveSym()函数计算当前处理项目的搜索符（预测符）。

④ 针对项目还未进栈的句柄部分（即圆点右侧的部分），逐个进行判断是否还有终结符以及产生式，如果有这样的一个产生式，为其添加圆点形成项目、利用checkEqual()函数判断该项目在不在项目集中，如果不在则加入项目集，并且为其计算并添加搜索符，这里需要将最终结果落实到closure()的传入参数LR1Item的set，所以需要对该set进行insert。

⑤ 同时，加入新项目的项目集，在这里意味着一开始的queue尾部push了一个新的LR1Item。

2. findPredictiveSym()函数

(1) 本函数的意义在于“ $A \rightarrow a \cdot Bb$ ”，B为currentSym，b为nextSymbol，算的是产生式左边为B的推出式的搜索符（预测符）。

(2) 具体代码设计

① 代码的传入参数是一个LR1Item项目item，和该项目的搜索符集合，一个int的set，这里用predictiveSymbol表示。

② 首先，如果当前“ \cdot ”已经到了项目的尾部，即句柄完全进栈，那么这里predictiveSymbol就是item的自己的predictiveSymbol，无需做更多的变化。

③ 进入“ \cdot ”未到项目的尾部的情况，还可以分为两种情况，代码中我们利用getNextSymbol()函数首先获得nextSymbol。然后第一种是例如“ $L \rightarrow \cdot i$ ”这种情景，对于这种情况可以通过将 \cdot 后的nextSymbol与终结符表中的所有符号一一比对，只要发现这个nextSymbol是终结符，就将其加入搜索符。由于开始的终结符表存为了map形式，也就是说我们直接在map中find这个nextSymbol，匹配了即加入搜索符。

④ 第二种情景则是最为普遍的情况，即“ $A \rightarrow a \cdot Bb$ ”这样的情形，这时需要计算b和已有搜索符 α 组成的 $b\alpha$ 的first集，代码中采取的方法，是首先利用producerMap（一个全局的map变量，记录了所有的产生式）取出了nextSymbol对应的所有产生式记为v1，然后将v1和已有的predictive Symbol放入getFirstSet()函数中去计算处理。

3. getFirstSet()函数

(1) 本函数完成了对于传入的符号的first集的计算以及同时完成了计算出first集之后，对于项目的predictiveSymbol也就是搜索符的修改。

(2) 具体代码设计：

① 函数传入的参数有int类型的producerID（因为代码中，每一个符号我们用一个不重复的数字进行对应），一个int类型的set的引用，记为firstSet。

- ② 首先, if判断, 若利用producerVector查询producerID对应符号的产生式, 发现产生式右侧为 ϵ , 那么直接结束函数, 并返回false (函数的返回值为bool类型)。
- ③ 进入else, 根据first的定义, if该符号的右部为一个终结符, 则直接将该终结符加入到first集 (即firstSet) 中。
- ④ 进入else, 当为非终结符时, 设置一个bool类型的epsilonFlag, 置为false, 接着取出所有的该非终结符的右侧表达式, 譬如 $\alpha = X_1X_2...X_n$, 则需要各自取出, 然后对每个X继续getFirstSet(), 只要出现了一个子getFirstSet为空的情况, 就将epsilonFlag置为true, 说明有空串。
- ⑤ 然后进入while循环, while判断条件为epsilonFlag==true同时没有计算到右侧表达式末尾则继续循环, 在循环内, 首先将epsilonFlag置false, 接下来if处理的符号为非终结符, 则继续获得当前的符号的右侧符号, 同样的继续上述的步骤, 即譬如 $\alpha = X_1X_2...X_n$, 则需要各自取出, 然后对每个X继续getFirstSet(), 只要出现了一个子getFirstSet为空的情况, 就将epsilonFlag置为true, 说明有空串。而如果处理的字符串为终结符, 则在插入该终结符进入first集 (即firstSet) 中。
- ⑥ 最后跳出各种循环与if...else语句, return true, 同时由于firstSet作为传引用的方式引入, 也完成了对于first集的处理

4. minimizeItemset()函数

- (1) 本函数的意义在于对一个LR1项目集的简化, 由于生成完一个LR1项目集的closure之后, 往往会出现, 一个项目, 不同的预测符同时出现在项目集中, 要对于这样的情况进行简化。
- (2) 具体代码设计:
- ① 函数传入参数为一个未简化的LR1项目集s1, 类型为一个LR1Item的multiset, 返回值为一个简化后的LR1项目集miniset, 类型为multiset;
- ② 函数首先定义miniset并且清空;
- ③ 然后开始针对s1进行循环, 循环内定义一个搜索符的set为predictiveSymSum然后定义bool变量findSimilar, 初始为false, 继续在miniset内创建一个循环 (到此函数已有二层循环), 如果外层循环子与内层循环子相似, 这里使用自定义的checkLike()函数, 则置findSimilar为true, 同时将外层循环子的搜索符赋值给predictiveSymSum, 这里就完成了搜索符合并。记录下此时的内层循环子的位置pos, 跳出内层循环, 接着判断findSimilar是否为true, 如果为false, 那么就直接将外层循环子插入到miniset, 因为没有相似项目。如果为true, 那么令一个LR1Item itm=pos, 并且根据pos在miniset中找到对应位置, 擦除, 再根据predictiveSymSum, 将该合并后的搜索符加入到, miniset中去即可。最后return miniset;
- ④ checkLike()函数自定义的一个功能函数, 作用是判断两个项目是否相似, 遵守规则为产生式一样, 预测符不一样, 返回true。主要完成了两个项目对于产生式的比较与搜索符的比较, 首先比较产生式, 若产生式不一致, 则直接返回false, 其次再去比较搜索符, 若搜索符不一致则返回true, 若一致, 则返回false, 因为二者就是同一项目, 这并不会出现再实际情况中。

3.2.3 构造LR(1)项目集规范族的详细设计, 包含ProcessItemSet()函数

该部分构建了LR(1)自动机并生成了语法分析表, 而这两件事是同步进行的。我们采用的算法依据的是上学期编译原理课上用的中文教材《编译原理及编译程序构造》中第6.4.1章“构造LR(1)项目集规范族的算法”, 具体思路如下:

构造 LR(1)项目集规范族的算法

构造 LR(1)项目集族的算法本质上和构造 LR(0)项目集族的算法是一样的, 先介绍两个函数: CLOSURE 和 GO。

(1) 函数 CLOSURE(I) — I 的项目集。

① I 的任何项目都属于 CLOSURE(I);

② 若项目 $(A \rightarrow \alpha \cdot B\beta, a)$ 属于 CLOSURE(I), $B \rightarrow \gamma$ 是一个产生式, 那么对于 FIRST(βa) 中每个终结符 b , 如果 $(B \rightarrow \cdot \gamma, b)$ 原来不在 CLOSURE(I) 中, 则把它加进去;

③ 重复步骤②, 直至 CLOSURE(I) 不再扩大为止。

因为 $(A \rightarrow \alpha \cdot B\beta, a)$ 属于 CLOSURE(I), 那么 $(B \rightarrow \cdot \gamma, b)$ 当然也属于 CLOSURE(I), 其中 b 必定是跟在 B 后面的终结符, 即 $b \in \text{FIRST}(\beta a)$, 若 $\beta = \epsilon$, 则 $b = a$ 。

(2) GO 函数。

令 I 是一个项目集, X 是一个文法符号, 函数 GO(I, X) 定义为:

$$\text{GO}(I, X) = \text{CLOSURE}(J)$$

其中, $J = \{ \text{任何形如 } (A \rightarrow \alpha X \cdot \beta, a) \text{ 的项目} \mid (A \rightarrow \alpha \cdot X\beta, a) \in I \}$ 。

可见在执行转换函数 GO 时, 搜索符并不改变。

(3) 构造拓广文法 G' 的 LR(1)项目集族 C 的算法如下:

PROCEDURE ITEMSETIR(1)

BEGIN

$C := \{ \text{CLOSURE}(\{ (S' \rightarrow \cdot S, \#) \}) \};$

REPEAT

FOR C 中的每个项目集 I 和 G' 的每个文法符号 X DO

IF GO(I, X) 非空且不属于 C THEN 把 GO(I, X) 加入 C 中

UNTIL C 不再扩大为止

END;

接下来是具体代码设计并实现的部分。

我先介绍一下我们在这部分里使用到的重要的自定义的数据结构、变量和函数:

① 结构体 LR1State, 表示 LR(1)项目集, 即自动机中的一个状态, 它包含一个用于标识该状态的状态编号 stateID (整数类型, 从 0 开始), 一个包含该状态下所有项目的集合 itemSet (它是 LR1Item 类型的 multiset), 一张标明当前状态通过特定符号可以到达什么状态的表格 moveItemSet (它是 unordered_map 类型)。

② LR(1)分析表、项目集转换动作表 actionTable。

③ 项目集队列 Q, 存储了当前待处理的项目 (它是 LR1Item 类型的队列)。

④ 函数 closure(multiset<LR1Item> & itemSet), 求当前项目集的闭包, 它的算法与先前“构造 LR(1)项目集规范族的算法”的第一部分一致。

最后，算法实现步骤如下：

- ① 求初态项目集 I_0 ：从 $(S' \rightarrow \cdot S, \#)$ 项目开始求闭包，再简化（该操作如下），然后将它存入项目集队列Q中。

例如，仍以G(Ex)文法为例，构造其LR(1)项目集族。初态项目集 I_0 ，从 $(S' \rightarrow \cdot S, \#)$ 项目开始求闭包可得图6.9所示的初态项目集：

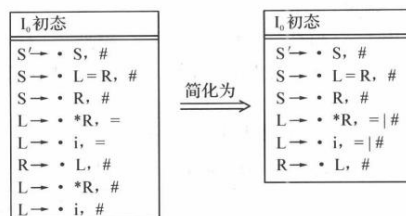


图 6.9 初态项目集

- ② 判断Q是否为空，为空则结束并返回1，不为空则转③。
- ③ 弹出队列Q顶的一个项目集itemset；提前定义好actionTable的一行f，长度为maxnum，是终结符和非终结符个数之和。
- ④ 对于当前项目集中的各项目求后继项目集（其算法与先前“构造LR(1)项目集规范族的算法”的第二部分类似），同时填写分析表actionTable。注意：(1) 对于新产生的项目集要求闭包并最小化后再放入；(2) 对于每个“新”状态，要判断是否与之前的某个状态相同。

3.2.4 LR(1)分析过程可视化的详细设计，包含PrintSymbolTable()、getSymbol()、PrintItem()、PrintItemSet()、PrintActionTable()函数

本部分负责LR(1)分析过程可视化的详细设计，主要是分为四个方面：①对.y文件解析出来的终结符与非终结符进行可视化；②打印所有产生式；③打印LR(1)项目集；④LR(1)分析表的可视化。

- ① 对.y文件解析出来的终结符与非终结符进行可视化

PrintSymbolTable()函数通过map数据结构的迭代器打印出终结符与非终结符与编号，如果存在语义值类型也会随后显示出来。

- ② 所有产生式的可视化

通过getSymbol()函数将Producer结构体中以编号存储的产生式转化为字符串形式，同时若存在语法规则动作会随后显示出来，否则输出“空”。

- ③ LR(1)项目集的可视化

PrintItemSet()函数，首先输出“输出项目集”字样，通过遍历函数输入的参数（一个LR1Item的multiset i0），在遍历的每一次循环中：首先输出当前循环子（LR1Item类型）的项目号，利用getProducerID来实现，接着输出对应的产生式的左侧，利用getSymbol实现，接着进入针对产生式右部的循环，循环条件为循环子超过右部的字符数量时，在循环中，如果发现循环子的大小等于当前外部的循环子（LR1Item类型）的getCurrentPosition大小，则输出·符号，因为getCurrentPosition函数就记录一个项目集中每一个项目的·的位置。结束内部的循环，进入if-else语句进行一些情况的判断：如发现getCurrentPosition值为-1，

那么说明项目规约完毕，输出无待规约符号；对应的else意味着仍然有待规约的符号，输出待规约符号，并在后面跟随输出当前·处理到的位置处的符号利用getSymbol（当前项目.getCurrentSymbol（））来实现。在函数最后，再进行一次遍历，通过遍历当前项目的getPredictiveSym，也就是说遍历当前项目的预测符集合，将预测符集合输出到项目的尾部。函数在输出过程使用“ ”、“\n”来控制输出精度。

④ LR(1)分析表的可视化

通过PrintActionTable()函数打印actionTable中存储的内容，对于每个项目集（状态）经过终结符达到的状态和动作（ACTION部分），以及经过非终结符达到的状态（GOTO部分）。

3.2.5 语法分析代码生成的详细设计，即GenerateCode() 函数

本部分负责从LR(1)分析表生成语法分析器的C代码。

首先，语法分析的对象是Lex送来的Token流，因此Yacc应与Lex联合使用。语法分析代码生成器会利用C语言的extern关键字，从词法分析器处引入Lex的词法分析函数string analysis(char*,int)，从而我们编写的Yacc通过readToken()函数从文件流中反复读入字符串yytext并调用analysis()函数来分析，这一过程中我们实则来测试我们之前的Lex词法分析器代码。

然后，流式的yytext及其对应的TokenName，借助语法分析表就可以完成基本的语法分析工作。主要用到三个数据结构：一个存储符号语义值的符号栈、一个存储状态转移路径的状态栈、一个存储ACTION- GOTO表的结构actionTable。由于符号栈与状态栈操作在语法分析过程中一致，我们使用了Sym结构体：

```
struct Sym
{
    int symbol;
    int state;
};
```

定义stack<Sym> symStack;来模拟符号栈与状态栈，每次查ACTION- GOTO表时，里面记录动作为“移进”时，新建一个将栈结点（Sym）存储状态与语义信息，然后压入symStack；当记录动作为“归约”时，表内存储用于归约的相应表达式编号，执行有关动作代码，symStack弹出等同于表达式中归约符号个数的状态（我们创建了一个名为producerN的一维数组来维护每个表达式的归约符号个数），并立刻递归地解析归约而成的非终结符 号以便执行GOTO动作；当进行到ACCEPT时，语法分析成功，然后退出程序；当进行到ERROR时，报错处理。其中，语法树的打印见3.2.6。

最后，整合上述各部分，以及.y源文件中的直接复制部分和用户代码部分，即完成了语法分析器代码的生成。将它与词法分析器代码进行联合编译，即可完成语法分析。

3.2.6 代码的语法树生成的详细设计，即printMTree()函数

1. 相关变量

首先创建相关的结构体，Tree、node。node结构体中包含，节点名称（String类型），子节点数量（int类型），子节点集合（node*类型的vector变量），Tree结构体中包含一个node节点，作为树的根节点。定义一个节点栈syntaxTreeStack，用于控制结点的记录顺序（node*的一个stack），以及一个语法树syntaxTree（Tree型变量）

2. 具体代码设计

当代码分析结果为ACCEPT时开始打印树，最初，syntaxTree的root要记录为syntaxTreeStack的top。而代码仍在分析的时候，对与syntaxTreeStack进行操作，这一部分为树能够建立起来的关键部分。

如果是在规约项目，首先创建一个空的父节点parentNode，该节点的name填上当前正在使用的规约式的产生式左部，然后进入循环，循环的跳出条件为循环子大于产生式右部符号数量。循环内，不断的将syntaxTreeStack的顶端push出来存入parentNode的child。并且父节点的子节点数目也同时增加。完成后，将该parentNode压入syntaxTreeStack。

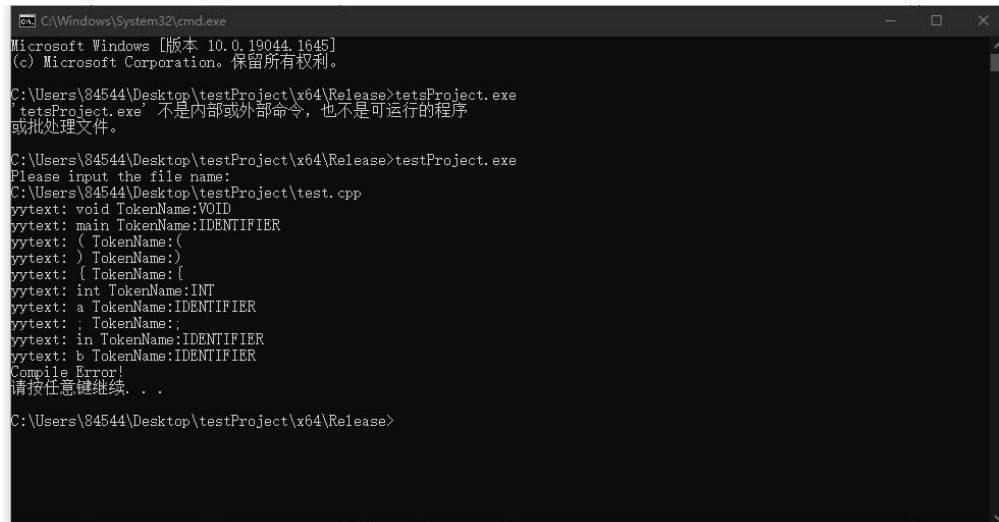
剩下的情况下，就创建一个新节点newNode，newNode的name

来到实际的printMTree函数设计，函数的参数列表（string（当前节点的垂直间隔），TreeNode（当前节点），bool（判断当前节点是否为第一个子节点用isFirst命名））首先当树中的节点不为null时，就一直往下推进，输出一个垂直间隔（用于区分不同的子节点分支，因为是横过来打印的）。利用一个isFirst判断此时为新的子节点还是同一个父节点产生的子节点的兄弟节点，再进入循环，循环的跳出条件为循环子大于当前节点的子节点的数量，然后在循环内部进入函数，形成递归调用，而递归调用中的prefix值有prefix + (isFirst ? " | \t\t\t" : " \t\t\t")来决定；node则设置为当前node的子节点列表中对应该下标为循环子的那一个；isFirst的bool值通过判断循环子是否等于0来决定。

4 使用说明

测试最终支持C99标准语法分析的 Lex+Yacc 使用说明

1. 找到exe文件，在对应的文件夹下打开cmd
2. 输入exe名称进行运行
3. 按照提示输入各种文件名称
4. 得到结果



```
Microsoft Windows [版本 10.0.19044.1645]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\84544\Desktop\testProject\x64\Release>tetsProject.exe
'tetsProject.exe' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

C:\Users\84544\Desktop\testProject\x64\Release>testProject.exe
Please input the file name:
C:\Users\84544\Desktop\testProject\test.cpp
yytext: void TokenName:VOID
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: int TokenName:INT
yytext: a TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: in TokenName:IDENTIFIER
yytext: b TokenName:IDENTIFIER
Compile Error!
请按任意键继续. . .

C:\Users\84544\Desktop\testProject\x64\Release>
```

5 测试用例与结果分析

测试用例 1: test_1.c

Yacc.y 的动作代码是在规约时输出规约用到的产生式，故从上到下为规约过程，从下到上为推导过程。经验证，没有错误。不过由于其语法树较为复杂，这里没有予以打印生成

```
int main()
{
    int a = 0;
    int c = 0;
    if(a == 2)
    {
        return a;
    }
    return c;
}
```

```

yytext: int TokenName:INT
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: int TokenName:INT
yytext: a TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName;;
yytext: int TokenName:INT
yytext: c TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName;;
yytext: if TokenName:IF
yytext: ( TokenName:(
yytext: a TokenName:IDENTIFIER
yytext: == TokenName:EQ_OP
yytext: 2 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: return TokenName:RETURN
yytext: a TokenName:IDENTIFIER
yytext: ; TokenName;;
yytext: } TokenName;}
yytext: return TokenName:RETURN
yytext: c TokenName:IDENTIFIER
yytext: ; TokenName;;
yytext: } TokenName;}
Compile sucessfully!

```

测试用例 2: test_2.c

Yacc.y 的动作代码是在规约时输出规约用到的产生式，故从上到下为规约过程，从下到上为推导过程。经验证，没有错误。由于其语法树较简单，这里进行予以打印生成。

```

void main()
{
}

```

```

yytext: void TokenName:VOID
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: } TokenName;}
Compile sucessfully!

```

<div>语法树输出结果</div> <div><div>└ translation_unit</div><div>└ external_declaration</div><div>└ function_definition</div><div>└ compound_statement</div><div>└ }</div><div>└ {</div><div>└ declarator</div><div>└ direct_declarator</div><div>└)</div><div>└ {</div><div>└ direct_declarator</div><div>└ main</div><div>└ declaration_specifiers</div><div>└ type_specifier</div><div>└ void</div></div>
<div>测试用例 3: test_3.c</div> <div><p>Yacc.y 的动作代码是在规约时输出规约用到的产生式，故从上到下为规约 过程，从下到上为推导过程。经验证，发现测试代码中有错误推导失败。这是正确的，因为测试代码给了一个不完整的的int符号。</p></div> <div><pre>void main() { int a; in b; return a; }</pre></div> <div><pre>yytext: void TokenName:VOID yytext: main TokenName:IDENTIFIER yytext: (TokenName:(yytext:) TokenName:) yytext: { TokenName:{ yytext: int TokenName:INT yytext: a TokenName:IDENTIFIER yytext: ; TokenName;; yytext: in TokenName:IDENTIFIER yytext: b TokenName:IDENTIFIER Compile Error!</pre></div>

6 课程设计总结（包括设计的总结和需要改进的内容）

09019204 曹邹颖

在Yacc文件分析的时候，就感觉到Yacc的难度相比Lex要高，还要注意避免冲突，完成SeuYacc代码后发现这一块的内容量、计算量果真很大。我们在前期对数据结构进行设计时，Producer、Item、State、ItemSet等各层次结构清晰，分工明确，好的架构让我们的编码效率大大提高，但是尽管如此对于处理完整的c99.y文件时速度也不够快，尤其是ACTION-GOTO表的构造阶段。我在最后整体代码优化时，由于时间关系没有将LR(1)这一范围最大的文法简化为LALR，这一过程涉及父类对象各个属性的优化，从而效率不佳。

09019223 许志豪

本次实验，我收获良多。在此前学习编译原理时，我对于“构造LR(1)项目集规范族”这一部分的掌握仅限于按照书上的算法按部就班的求解，只有我对此进行代码实现时候，我才意识到我要做的其实不是先求整个项目集族后填转换表，而是用一个队列记录待处理的项目集，同时用转化表记录每次的转化结果。

不过在构造LR1Item时，我觉得我们的方法效率有些低，这或许是因为算法中引入了较为多次的比较，这样虽然增加了项目的鲁棒性，也一定程度增加了项目的时间复杂度。

09019104 陈逸彤

本次实验，我收获良多。在此前学习编译原理时，我对于“构造LR(1)项目集规范族”这一部分的掌握仅限于按照书上的算法按部就班的求解，只有我对此进行代码实现的时候，我才意识到我要做的其实不是先求整个项目集族后填转换表，而是用一个队列记录待处理的项目集，同时用转化表记录每次的转化结果。然而，我们代码的效率并不高，我个人认为是因为在“构造LR(1)项目集规范族”这个部分中“对于每个‘新’状态，要判断是否与之前的某个状态相同”这一步，我们选择了比较暴力的依次比较两项目集的项目个数、每个项目的产生式与预测符是否相等的算法。由于我们对项目集的定义中有确保其中的每个项目是按自定义好的从小到大的顺序存储，所以该比较算法看起来效率应该还不错，然而，该算法归根结底还是做很多比较，而对于计算机而言，比较运算的速度是要比算术运算慢的，一旦运算量很大，缺陷就会很明显。对于这个问题，我们有考虑过使用拓扑将每个项目集映射成一个数字，然后比较两个数字即可。然而，我们的算法水平还不足以完成这个拓扑映射算法的设计，特别是要保证映射后不会产生重合这一点。

7 教师评语

签名：

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做 PPT 文件、本设计报告等一切可放入光盘的内容的光盘。