東南大學

编译原理课程设计

设计报告

组长：09017227 卓　旭

成员：09017224 高钰铭

09017225 沈汉唐

东南大学计算机科学与工程学院

二0 20 年 5 月

设计任务名称		SeuYacc		
完成时间		2020-06-04	验收时间	2020-06-07
本组成员情况				
学　号	姓　名	承　担　的　任　务		成 绩
09017227	卓　旭	LR1 自动机的构建、LR1 语法分析表的生成、可视化功能		
09017224	高钰铭	.y 文件的解析、LR1 自动机的构建、语法分析器代码生成、符号表构建		
09017225	沈汉唐	LR1 自动机的构建、LR1 向 LALR 的转换		

1 编译对象与编译功能

1.1 编译对象

在对 C 的分析方面，如果将 c99.y 直接作为编译对象，即使程序经过一定优化，也需耗时数十分钟。因此我们对它进行大规模删减，将处理时间控制到 5 分钟左右。

c99.y 内产生式并没有动作代码，为了便于分析，我们补充了一定量的动作代码。

另外，c99.y 头部声明没有左右结合关系，所以我们还写了一个简易计算器的.l 与.y，用于测试结合性和优先级能否正确解决冲突。

具体见 example\Simplified\c99_test.y 与 example\Calculator。

1.2 编译功能

- YaccParser.ts 负责对.y 源文件进行解析
- Grammar.ts 中对一些语法相关的数据结构进行定义
- LR1.ts 负责 LR1 自动机的构造，以及 ACTION-GOTO 表的生成
- LALR.ts 负责将 LR1 自动机重构为 LALR 的
- CodeGenerator.ts 负责语法分析器 C 代码的生成
- Visualizer.ts 负责各类可视化功能的实现

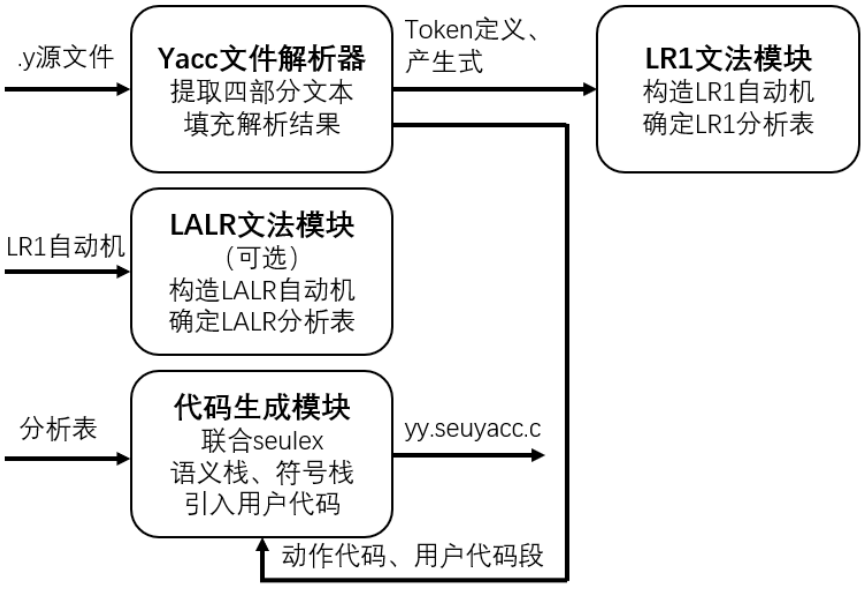
2. 主要特色

- 广泛使用面向对象编程，使代码结构清晰，各部分独立，之间易于相互调用
- .y 文件头部声明支持%token、%left、%right、%start
- 动作代码部分可以使用$$、$1、$2、$3 等获取栈内元素
- 在进行 LR1 分析时，对产生式、状态、符号等全部进行编号，利用编号索引，节省空间，效率较高
- 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)
- 可以选择将文法转换为 LALR 的
- 提供了自动机可视化、ACTION-GOTO 表可视化、语法树可视化功能
- 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 Yacc 文件解析器、LR1 文法模块、LALR 文法模块、代码生成模块、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

YaccParser.ts 的详细设计

负责解析出四部分文本：直接复制部分、%xxx 声明部分、产生式-动作部分、用户代码部分。然后填充 Token 声明数组、运算符声明数组、产生式数组、非终结符数组、开始符号等有关信息，供下层使用。

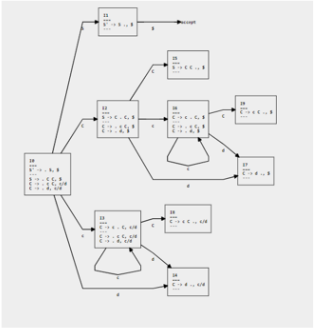
这一部分的整体设计与 LexParser 基本一致，可以通过对文件内容逐行扫描，通过定界符%%和%}等进行判别。解析过程中可以顺便判断.y 文件的结构是否符合要求，如果结构不正确，则需要报错处理。在解析产生式-动作部分时，我们同样借助一定的手段，保持了程序的健壮性，支持多种大括号风格，以及不定义动作代码等形式。此处不再赘述。

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，这与原版 Yacc 的行为是一致的。

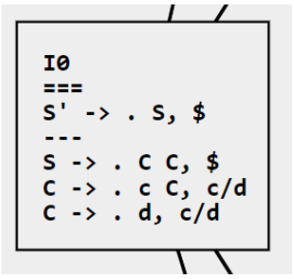
Grammar.ts 的详细设计

该模块主要定义了与文法分析相关的一系列数据结构。

我们采用分级分层、面向对象的思想来构建。LR1DFA（LR1 自动机）包括一系列 LR1State 和连接边；LR1State（LR1 项目集）由一系列 LR1Item 组成；LR1Item（LR1 项目）由 LR1Producer、点号和展望符构成，对于有多个展望符号的，拆解为只有一个展望符号的形式进行存储；LR1Producer（LR1 产生式）包括用符号经过编号后的产生式以及对应的动作代码。具体示意图如下：



LR1DFA



LR1State

```
S' -> . S, $
```

LR1Item

```
S' -> S
```

LR1Producer (无点)

LR1.ts 的详细设计

该部分主要完成了 LR1 自动机的构建和语法分析表的生成。

首先，为全体文法符号（终结符、非终结符、特殊符号）按一定规则分配编号，从而将产生式转换为“编号→编号数组”的表示形式，这样可以方便后续的处理。

然后按照龙书算法 4.53（如下图所示）构建 LR1 自动机。需要注意的是：①求取 GOTO(I, X)是一个极其耗时且可能有深递归的过程，而对于某对特定的参数 I_0、X_0，GOTO(I_0, X_0)可能被多次需要。所以我们采用空间换时间的策略，记录了所有求过的 GOTO(I, X)，再次需要时只要直接查表即可。经实验，该优化策略让算法效率有些许的提高。②增广产生式 S’→S 的动作代码应为缺省的$$=$1。

算法 4.53 LR(1)项集族的构造方法。

输入：一个增广文法 G'。

输出：LR(1)项集族，其中的每个项集对文法 G'的一个或多个可行前缀有效。

方法：过程 CLOSURE 和 GOTO，以及用于构造项集的主例程 items 见图 4-40。

```
SetOfItems CLOSURE(I) {
    repeat
        for ( I 中的每个项 [A → α·Bβ, a] )
            for ( G' 中的每个产生式 B → γ )
                for ( FIRST(βa)中的每个终结符号 b )
                    将 [B → ·γ, b] 加入到集合 I 中;
    until 不能向 I 中加入更多的项;
    return I;
}

SetOfItems GOTO(I, X) {
    将 J 初始化为空集;
    for ( I 中的每个项 [A → α·Xβ, a])
        将项 [A → αX·β, a] 加入到集合 J 中;
    return CLOSURE(J);
}

void items(G') {
    将 C 初始化为{CLOSURE}({[S' → ·S, $]});
    repeat
        for ( C 中的每个项集 I )
            for ( 每个文法符号 X )
                if ( GOTO(I, X) 非空且不在 C 中)
                    将 GOTO(I, X) 加入 C 中;
    until 不再有新的项集加入到 C 中;
}
```

图 4-40 为文法 G'构造 LR(1)项集族的算法

此外，在构建 LR1 自动机的过程中还需要多次求取一串文法符号的 FIRST 集。我们使用了龙书 4.4.2 的算法（如下图所示），事先求出每一个文法符号的 FIRST 集，再根据各个文法符号的 FIRST 集求取指定的一串文法符号的 FIRST 集（不动点法）。

计算各个文法符号 X 的 FIRST(X)时，不断应用下列规则，直到再没有新的终结符号或 ϵ 可以被加入到任何 FIRST 集合中为止。

1）如果 X 是一个终结符号，那么 FIRST(X) = X。

2）如果 X 是一个非终结符号，且 $X \to Y_1 Y_2 \cdots Y_k$ 是一个产生式，其中 $k \geqslant 1$，那么如果对于某个 i，a 在 FIRST(Y_i)中且 ϵ 在所有的 FIRST(Y_1)、FIRST(Y_2)、…、FIRST(Y_{i-1})中，就把 a 加入到 FIRST(X)中。也就是说，$Y_1 \cdots Y_{i-1} \Rightarrow \epsilon$。如果对于所有的 $j=1, 2, \cdots, k$，ϵ 在 FIRST(Y_j)中，那么将 ϵ 加入到 FIRST(X)中。比如，FIRST(Y_1)中的所有符号一定在 FIRST(X)中。如果 Y_1 不能推导出 ϵ，那么我们就不会再向 FIRST(X)中加入任何符号，但是如果 $Y_1 \Rightarrow \epsilon$，那么我们就加上 FIRST(Y_2)，依此类推。

3）如果 $X \to \epsilon$ 是一个产生式，那么将 ϵ 加入到 FIRST(X)中。

现在，我们可以按照如下方式计算任何串 $X_1 X_2 \cdots X_n$ 的 FIRST 集合。向 FIRST($X_1 X_2 \cdots X_n$)加入 $F(X_1)$中所有的非 ϵ 符号。如果 ϵ 在 FIRST(X_1)中，再加入 FIRST(X_2)中的所有非 ϵ 符号；如果 ϵ 在 FIRST(X_1)和 FIRST(X_2)中，加入 FIRST(X_3)中的所有非 ϵ 符号，依此类推。最后，如果对所有的 i，ϵ 都在 FIRST(X_i)中，那么将 ϵ 加入到 FIRST($X_1 X_2 \cdots X_n$)中。

在获得 LR1 自动机后，就可以按照龙书算法 4.56 构造语法分析表（ACTION-GOTO 表）了。但我们的 LR1 自动机中可能存在移进-规约冲突和规约-规约冲突，故仅凭 LR1 自动机不能保证构造出 LR1 的语法分析表，此时需要借助优先级关系进行冲突的解决。对于移进-规约冲突：展望符的优先级就是移进的优先级，最后一个终结符的优先级就是规约的优先级。若后者更高，则动作替换为规约；若二者相等，则考虑结合性，左结合就规约，右结合就移进；对于规约-规约冲突：越早定义的产生式优先级越高，用它进行规约；对于所有未明确定义优先级的情况，我们默认偏爱移进（in favor of shift），这与原版 Yacc 的行为是一致的。

算法 4.56 规范 LR 语法分析表的构造。

输入：一个增广文法 G'。

输出：G'的规范 LR 语法分析表的函数 ACTION 和 GOTO。

方法：

1）构造 G'的 LR(1)项集族 $C' = \{I_0, I_1, \cdots, I_n\}$。

2）语法分析器的状态 i 根据 I_i 构造得到。状态 i 的语法分析动作按照下面的规则确定：

① 如果$[A \to \alpha \cdot a\beta, b]$在 I_i 中，并且 GOTO(I_i, a) = I_j，那么将 ACTION[i,a]设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果$[A \to \alpha \cdot, a]$在 I_i 中且 $A \neq S'$，那么将 ACTION[i, a]设置为“规约 $A \to \alpha$”。

③ 如果$[S' \to S \cdot, \$]$在 I_i 中，那么将 ACTION[i, \$]设置为“接受”。

如果根据上述规则会产生任何冲突动作，我们就说这个文法不是 LR(1)的。在这种情况下，这个算法无法为该文法生成一个语法分析器。

3）状态 i 相对于各个非终结符号 A 的 goto 转换按照下面的规则构造得到：如果 GOTO(I_i, A) = I_j，那么 GOTO[i, A] = j。

4）所有没有按照规则(2)和(3)定义的分析表条目都设为“报错”。

5）语法分析器的初始状态是由包含$[S' \to \cdot S, \$]$的项集构造得到的状态。 □

LALR.ts 的详细设计

本部分负责从 LR1 自动机构造 LALR 自动机。LALR 自动机的构建就是合并同心状态为一个，作为新状态。所谓同心，就是两个 LR1State 的全体 LR1Item 的第一分量（即忽略展望符号）完全相同。这样，可以有效削减自动机状态数，提高运行效率。

但我们了解到，从 LR1 自动机构造 LALR 自动机是简单、空间需求大、效率不高的，原因出在 LR1 项目集族构造的时间空间需求过多。进一步思考，我们发现，全体 LR1Item 的第一分量（即忽略展望符号）实际上就是 LR0 的项目集，因此实际上可以从 LR0 自动机出发构造 LALR 自动机，这可以作为后续优化的一个方向。

在构造好 LALR 自动机后，LALR 语法分析表的构造与 LR1 的完全相同，可以复用程序。经实验发现，对于比较复杂的文法，LALR 能简化的状态数量比较有限，因此我们默认不启用 LALR 文法分析。

CodeGenerator.ts 的详细设计

本部分负责从语法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。代码生成器会先生成 yy.tab.h，其中包含 Token 的编号信息，以供.l 源文件引入，从而 Lex 能够返回正确的编号。然后，利用 C 语言的 extern 关键字，从词法分析器处引入 yyin、yyout、yylex()、yytext，从而 Yacc 只要反复调用 yylex()，就能获得流式的 Token 序列及其对应的 yytext，这与原版 Yacc 的行为是一致的。

值得一提的是，Yacc 在定义产生式时，支持单引号引起的字符出现在产生式中（如 init_declarator : declarator '=' initializer），而不一定要使用 Token 名，也不需要定义 Token 名与引号字符的对应关系。但这实际上是会带来歧义的：仅使用 yytext 与之匹配显然不够严谨；若 Yacc 自行处理该字符，然后挪动 yyin，则会影响 Lex 的正确运行；这也不符合“语法分析的对象是 Lex 送来的 Token 流”的规则。因此，SeuYacc 不支持这种定义方式。

然后，借助语法分析表就可以完成基本的代码生成工作。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、一个同时存储 ACTION 表和 GOTO 表的结构。当进行到移进格时，将目标入状态栈，并新建一个符号栈结点存储语义信息；当进行到归约格时，不消耗当前收到的 token，执行有关动作代码，状态栈弹出等同于归约符号个数的状态，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 GOTO 格时，将目标状态入状态栈；当进行到 acc 时，接收整个输入，然后退出程序；当进行到报错格时，报错处理。

对于动作代码中的$内容，$i 代表归约的产生式右侧第 i 个符号的语义值，$$表示产生式左侧非终结符的语义值。前者替换为语义值存储的位置，由于我们使用数组模拟栈，因此可以随机访问，不需要弹栈；后者替换为临时的存储变量，待动作代码执行完成后，符号栈进行 pop 与 push 时再存入。

由于最终生成的语法分析器代码中不包含有关文法符号名称的信息，为了支持语法树的打印，我们在生成 LR1 自动机时，自动地将有关文法符号的信息写进了各个产生式的 action 代码中。这样就可以通过在推导过程中边走边建立结点，存储文法符号的名称，从而记录语法树结构。我们还提供了符号表的支持，通过暴露给用户两个函数——新建项目和获取项目，从而允许用户在动作代码中寄存某些符号的信息，这为后续的中间代码生成打下基础。

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。

Visualizer.ts 的详细设计

在调试各算法的过程中，如果自动机只是看不见摸不着的一堆数据结构，对我们的调试效率有很大影响。因此，我们使用 dagre-d3 库，通过指定各节点显示内容、边上标签、迁移关系等，实现了 LR1、LALR 自动机的可视化，这给我们的调试过程带来了极大的便利。

另外，我们还通过 HTML 实现了 ACTION-GOTO 表的可视化，这也给我们的调试过程带来了极大的便利。

4 使用说明

SeuYacc 使用说明

①从项目 GitHub 仓库（https://github.com/z0gSh1u/seu-lex-yacc/）下载或克隆整个仓库。

② seuyacc 依赖于 Node.js 运行时，因此你需要预先安装 Node.js（https://nodejs.org/zh-cn/）。然后在项目根目录下执行 npm install 来安装依赖。

③现在便可以使用下列命令运行 seuyacc 的 CLI 工具了：

node <path_to_project>/dist/seuyacc/cli.js <yacc_file>

<yacc_file>为.y 文件的位置。我们为 C 语言准备的版本在 example/SimplifiedTest 目录下，为计算器准备的版本在 example/CalculatorTest 目录下。

④执行上述命令后，稍作等待，就会在当前目录下生成 yy.seuyacc.c 和 yy.tab.h 文件，即为语法分析器的 C 源代码以及 Token 编号定义，如图所示：

```
F:\seulexyacc\example\SimplifiedTest>node ..\..\dist\seuyacc\cli.js .\Yacc.y
[ Running... ]
[ Parsing .y file... ]
[ Building LR1... ]

[ constructLR1DFA or LALRDFA, this might take a long time... ]
Progress: 100.00% ████████████████████ 36366/36366
[ constructACTIONGOTOTable, this might take a long time... ]
Progress: 99.04% ███████████████████▒ 207/209declarations -> declaration
declarations
Progress: 99.52% ███████████████████▒ 208/209

[ Generating code... ]
[ Main work done! Start post-processing... ]
[ All work done! ]
[ Time consumed: 28281 ms. ]
```

⑤将 yy.seuyacc.c 与对应的 yy.seulex.c 联合编译：

gcc yy.seuyacc.c yy.seulex.c -o yy.ok.exe

即可生成完整的语法分析程序 yy.ok.exe。

⑥运行 yy.ok.exe 进行语法分析后，将在目录下生成 yacc.tree，即为对应的语法树文件。

5 测试用例与结果分析

测试用例 1: \example\SimplifiedTest\C.in

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。

```
int main() {
  if (a == 2) {
    printf("hello world!");
  }
  int b = 233;
  return c;
}
```

```
Reduce@type->INT
Reduce@arithmetic_expr->IDENTIFIER
Reduce@arithmetic_expr->CONSTANT
Reduce@logic_expr->arithmetic_expr EQ_OP arithmetic_expr
Reduce@arithmetic_expr->STRING_LITERAL
Reduce@argument_list->arithmetic_expr
Reduce@function_call->IDENTIFIER LPAREN argument_list RPAREN
Reduce@stmt->function_call SEMICOLON
Reduce@stmts->stmt
Reduce@block_stmt->LBRACE stmts RBRACE
Reduce@stmt->block_stmt
Reduce@stmt->IF LPAREN logic_expr RPAREN stmt
Reduce@type->INT
Reduce@arithmetic_expr->CONSTANT
Reduce@assign_expr->IDENTIFIER ASSIGN arithmetic_expr
Reduce@var_declaration->type assign_expr SEMICOLON
Reduce@stmt->var_declaration
Reduce@arithmetic_expr->IDENTIFIER
Reduce@stmt->RETURN arithmetic_expr SEMICOLON
Reduce@stmts->stmt
Reduce@stmts->stmt stmts
Reduce@stmts->stmt stmts
Reduce@block_stmt->LBRACE stmts RBRACE
Reduce@func_declaration->type IDENTIFIER LPAREN RPAREN block_stmt
Reduce@declaration->func_declaration
Reduce@declarations->declaration
Reduce@program->declarations
```

测试用例 2: \example\CalculatorTest\Calculator.in

测试用例 1 的 Yacc 源文件没有定义优先级、结合性，故我们写了一个四则计算器的 Lex 与 Yacc 源文件（位于\example\Calculator）来测试 SeuYacc 能否正确处理冲突，以及能否正确处理语义值。

动作代码为在规约时打印所用的产生式，以及$$、$1、$3，程序最后再输出$$。经验证，结果正确无误。

```
2+3*4-5+1
```

```
r(expr*expr)
12,3,4
r(expr+expr)
14,2,12
r(expr-expr)
9,14,5
r(expr+expr)
10,9,1
Result is 10
```

语法树输出结果

```
expr (10) {
  expr (9) {
    expr (14) {
      expr (2) {
        2
      },
      +,
      expr (12) {
        expr (3) {
          3
        },
        *,
        expr (4) {
          4
        }
      }
    },
    -,
    expr (5) {
      5
    }
  },
  +,
  expr (1) {
    1
  }
}
```

6 课程设计总结（包括设计的总结和需要改进的内容）

09017227 卓旭

在 Yacc 中，我们继续使用了结构清晰的面向对象编程方法，并且在构造 ACTION-GOTO 表时，利用了空间换时间的方法缓存 GOTO(I,X)的计算结果，极大地提高了效率。在前期对数据结构进行设计时，Producer、Item、State、DFA 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。

但由于时间仓促，Yacc 的处理效率并不太理想，尤其是 ACTION-GOTO 表的构造阶段，速度很慢。由于该阶段对 DFA 状态的遍历具有独立性，因此可以考虑使用并行化技术进行加速。

09017224 高钰铭

项目中的 yacc 初期使用了 LR(1)语法，根据用户提供的 yacc 文件构建了 LR1 的项集、状态转换机以及语法分析表，并根据语法分析表进行语法分析。为了节省内存以及方便对语法符号和产生式的查找，项目将所有语法符号和产生式放置在各自的字典中，其他地方通过索引来对其进行调用，这一设计使得程序较为简洁。

但项目中的 yacc 与 lex 具有相同的问题，即程序优化不足，存在运行慢的问题，这一点有待改进。

09017225 沈汉唐

yacc 的难度相比 lex 真的是太高了，这一块的内容量、计算量非常大。尤其是自下而上语法分析这一块，在上学期课程中，通过手工运算就能体会到它的复杂性；如今用程序实现时，虽说复杂的运算能用计算机计算，但是算法本身依然是充满挑战性的。LR(1)是一个范围最大的文法，那么将其化为 LALR 的过程，本质上是一个简化的过程，涉及到父类对象各个属性的优化。和 DFA 最小化相比，它要难多了。

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。