

《编译原理》课程项目

词法和语法分析器实验报告



205XXXX GTY

205XXXX CXX

205XXXX CWQ

指导教师：WZH 老师

2022 年 11 月

目录

一、实验内容.....	5
1.1 词法分析部分	5
1.2 语法分析部分	5
二、需求分析.....	5
2.1 整体需求	5
2.2 词法分析需求.....	6
2.3 语法分析需求	6
三、概要设计.....	7
3.1 词法分析部分	7
3.2 语法分析部分	7
3.2.1 定义准备	7
3.2.2 LR1 文法处理	8
3.2.3 构建 Action Goto 表.....	9
3.2.4 语法分析	10
3.3 模块整合	10
四、详细设计.....	11
4.1 词法分析核心	11
4.1.1 DFA 构建.....	11
4.1.2 词法分析器	12
4.2 语法分析核心	13
4.2.1 Tcey 格式	13
4.2.2 语法	13
4.2.3 LR1 语法数据结构	14
4.2.4 LR1 项目集构造：准备与提取表达式	15
4.2.5 LR1 项目集构造：构造 first 集.....	15
4.2.6 LR1 项目集构造：拓广与引入文件结尾符号	15

4.2.7 LR1 项目集构造：求闭包	15
4.2.8 LR1 项目集构造：构造初始状态	16
4.2.9 LR1 项目集构造：计算转移	16
4.2.10 Action Goto 表及其构造	17
4.2.11 语法分析	17
4.3 语法分析命令行程序	18
4.3.1 基本使用	18
4.3.2 graphviz dot 代码生成	19
4.3.3 缓存优化	19
4.4 ToyCompile 项目框架	20
五、调试分析	20
5.1 词法分析测试文件	20
5.2 语法分析测试 – 正确情况	21
5.3 语法分析测试 – 错误情况	24
5.4 性能分析	24
5.4.1 自动机	24
5.4.2 词法分析	24
5.4.3 Action Goto 表构建	24
5.4.4 语法分析	25
5.4.5 语法分析整体开销	25
六、总结与收获	25
6.1 项目收获	25
6.2 遇到的问题及其解决	26
6.3 课程认识	26
七、获取产品	26
7.1 获取源码	26
7.2 构建运行	26
八、致谢	27

九、参考资料.....	27
-------------	----



一、实验内容

1.1 词法分析部分

给出类 C 语言的单词子集及机内表示，试编写一个词法分析器，输入为源程序字符串，输出为单词的机内表示序列。

在完成以上基本要求的情况下，对程序功能扩充：

- (1) 增加单词（如保留字、运算符、分隔符等）的数量；
- (2) 将整常数扩充为实常数；
- (3) 增加出错处理功能；
- (4) 增加预处理程序，每次调用时都将下一个完整的语句读入扫描缓冲区，去掉注释行，并能够对源程序列表打印。

1.2 语法分析部分

1. 根据 LR(1)分析方法，编写一个类 C 语言的语法分析程序，可以选择以下两项之一作为分析算法的输入：

- (1) 直接输入根据已知文法人工构造的 ACTION 表和 GOTO 表。
- (2) 输入已知文法，由程序自动生成该文法的 ACTION 表和 GOTO 表。

2. 语法分析程序要能够调用词法分析程序，并为后续调用语义分析模块做考虑。

3. 对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程和语法树。

二、需求分析

2.1 整体需求

本阶段，本项目需要完成对 C 语言代码单文件的识别，提取其中每个符号，构建语法树并展示。完整的编译过程需要经历头文件展开，词法和语法分析，生成中间代码，生成汇编，生成二进制文件，链接过程。本项目仅对其中的“词法和语法分析”展开研究，在未来会对后续步骤展开研究。头文件展开不在本项目的研究范围内。

2.2 词法分析需求

C 语言代码可以视为由一个个符号序列组成。每个符号由单个或多个字符组成，符号之间可能有空格或 tab 等分隔符隔开，也可能没有分隔。词法分析的目的是将 C 语言代码转换为符号串，以便后续分析。

现今的代码经常包含多字节字符。如在 GB2312 标准下，可能存在 2 字节编码的字符；在 UTF8 标准下，字符编码可能更长。识别多字节编码字符需要一定技巧，但不是本项目的研究重点。因此，我们人为规定输入的代码文件不存在多字节字符。即：所有字符都仅使用 1 字节编码。

本项目目的是巩固课程知识，而不是真正商业化。因此，我们选择以软编码方式构建词法分析器。即：词法分析器同时接收源代码文件和词法识别自动机构建文件，现场构建自动机，再识别源代码文件中的符号。

我们参考 LLVM Clang⁴，在词法部分支持 C++11 标准内的所有符号（编译预处理指令除外）。对于异常情况（如多行注释未闭合），输出错误位置和错误符号类型。如果不存在异常情况，则得到完整的符号列表，供后续语法分析部分使用。

2.3 语法分析需求

经历词法分析，我们得到整个程序的符号列表。同时，每门程序语言都有一套语法定义规则。根据语法定义规则，我们可以看懂符号列表表达的意思，并构建语法树，以便进一步分析。

本项目采用 LR 分析方式完成对输入符号串的分析。为实现较强的分析能力，将程序文法构建成 LR1 项目集，并构建 Action Goto 表。

我们在网络上找到一套完整的 C99 语法规则，遵循其规则进行语法分析。因此，首先需要读取该语法定义文件，将其转换成程序内部表示形式，构建 LR1 项目集，最后填充 Action Goto 表，交给语法分析器使用。

为巩固课程知识，我们采用程序动态构建 Action Goto 表的形式，完整执行语法定义读取到 Action Goto 表的构建过程。即：语法分析程序在接收符号串的同时，依赖外部输入的语法定义文件。

当语法定义文件存在错误时，应及时报错，并停止程序执行。语法定义文件解析完毕，即可对符号串进行分析。若某符号在某状态下，在 Action Goto 表内没有项目，则报错，并停止分析过程。分析成功后，分析器内部形成一棵语法树，以合适的形式可视化即可。

三、概要设计

3.1 词法分析部分

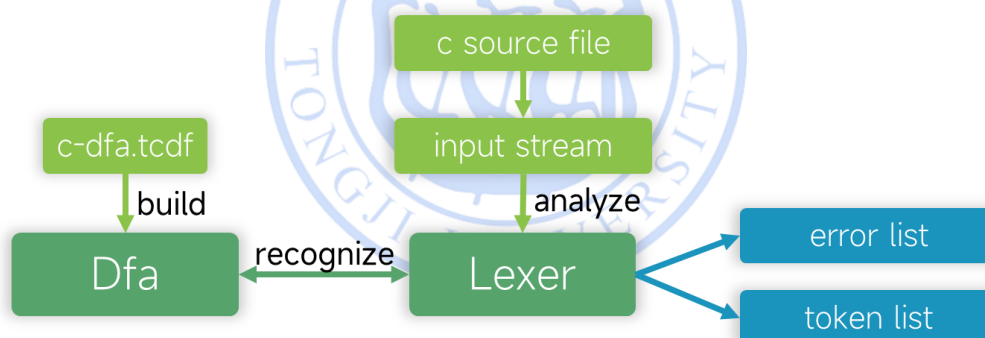
词法分析部分的任务相对简单。程序面对的代码文件由一个个字符组成。几个字符连在一起，形成有一定含义的符号。将这些符号提取出来，并且过滤符号之间的空白内容，则是词法分析的任务。

识别符号的核心是自动机。我们选择在外部分编辑自动机，并按配置文件形式在程序内动态构建自动机。如此设计让自动机的错误调试及修改编辑变得更加简单。

借助构建完毕的自动机，将代码文件内的一个个字符传入自动机进行识别，即可提取一个个符号。

课程中介绍的词法识别器拥有双缓冲区设计，其意义在于避免完整读入文件后统一处理，以此降低内存压力。我们选择以输入流形式代替双缓冲区设计，以更优雅的方式实现更好的效果。

词法分析器整体结构如下图所示：



3.2 语法分析部分

3.2.1 定义准备

符号列表本身包含一定信息，但这些信息不足以形成对整个代码文件的理解。语法分析的任务则是根据已知语法规则，尝试理解一个符号列表的含义，在其上构建语法树。

首先，我们需要知道 C 语言的文法是什么。很幸运，我们在网络上找到由 Jutta Degener 整理的 C99 文法规则⁷。该规则按照 Yacc 格式定义，很适合我们的项目使用。

```

495 expression_statement
496 : ';'
497 | expression ';'
498 ;
499
500 selection_statement
501 : IF '(' expression ')' statement
502 | IF '(' expression ')' statement ELSE statement
503 | SWITCH '(' expression ')' statement
504 ;
505
506 iteration_statement
507 : WHILE '(' expression ')' statement
508 | DO statement WHILE '(' expression ')' ';'
509 | FOR '(' expression_statement expression_statement ')' statement
510 | FOR '(' expression_statement expression_statement expression ')' statement
511 ;
512
513 jump_statement
514 : GOTO IDENTIFIER ';'
515 | CONTINUE ';'
516 | BREAK ';'
517 | RETURN ';'
518 | RETURN expression ';'
519 ;
520
521 translation_unit
522 : external_declaration
523 | translation_unit external_declaration
524 ;
525
resources/ansi-c-mod.tcey.yacc

```

对于该 Yacc 格式文件，我们需要通过一个模块将其转换为程序内部数据结构。该模块工作流程如下图所示：



以输入流形式将 Yacc 定义文件输入给分析模块，其产物为原始文件的机内数据结构，以便进一步处理。

3.2.2 LR1 文法处理

仅有语法信息，无法分析词法列表。我们需要将它转换成一张 Action Goto 表，供分析器使用。转换前，需要先将语法信息转换成 LR1 项目集，基于后者可以很轻松构建 Action Goto 表。

LR1 项目集构造模块结构如下图所示：



该模块接收一个文法规则。然而，该规则包含很多不方便识别的描述，如：

A -> Bc | de

为方便后续处理，我们将这样的表达式全部改造成不带“|”符号的。即，将上述表达式改成 2 条独立的表达式，对其他表达式进行同样的处理。

拥有所有表达式，即可很方便地构建所有符号的 first 集合，在后续分析过程中需要频繁使用。

在正式开始前，还需要对文法进行拓广。即，对于开始符号为 S 的文法，添加如下产生式：

$$S' \rightarrow S$$

由于文法定义文件没有指明文件结尾符号，我们需要额外补充文件结尾符号的定义。

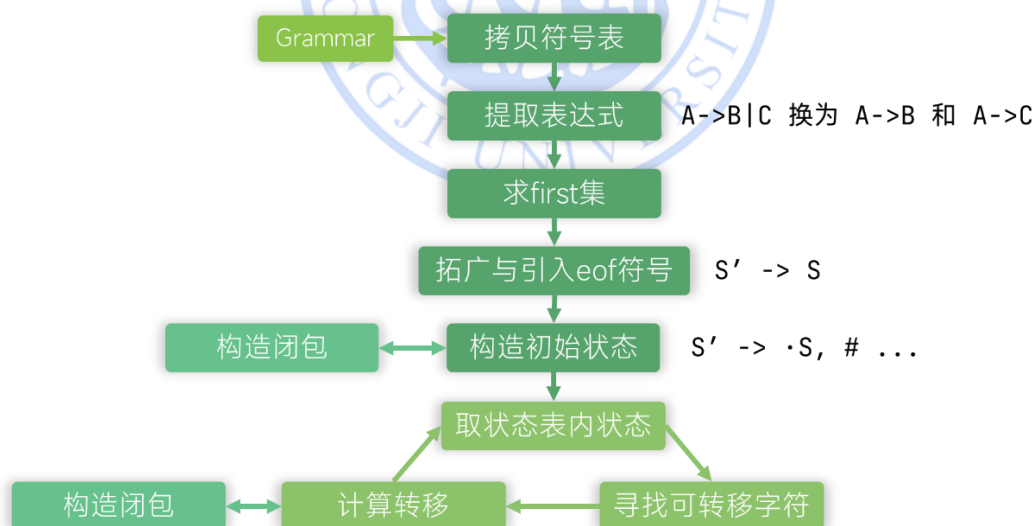
万事俱备，创建初始状态的第一个表达式：

$$S' \rightarrow \cdot S, \#$$

对该表达式求闭包，得到完整的初始状态，将其放入状态列表。

之后，对于状态集合中的每个状态，求可以令其发生转移的符号。对于每个符号，计算转移后的状态，并将新产生的状态添加到状态集合。如此循环，直到所有状态处理完毕，且不产生新状态。

整体工作流程如下：

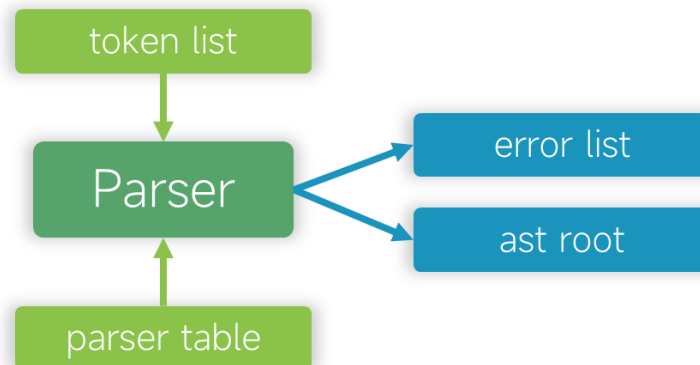


3.2.3 构建 Action Goto 表

基于处理完毕的 LR1 文法，观察每个状态内的每个表达式，即可很轻松地构建 Action Goto 表。该表将作为语法分析器的重要分析参考。

3.2.4 语法分析

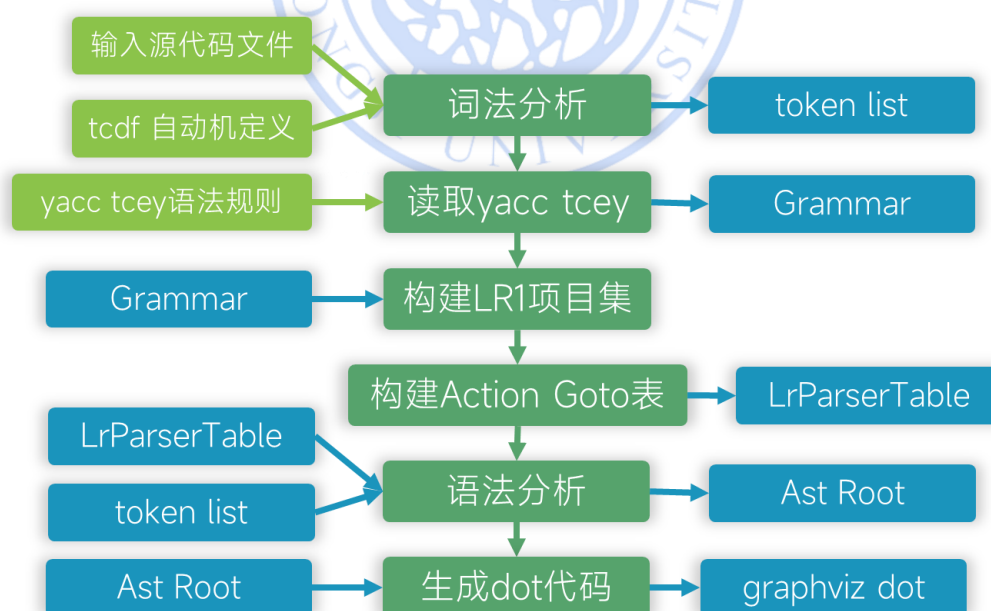
万事俱备。语法分析器持有 Action Goto 表和输入符号串，构建语法树。其工作流程如下：



3.3 模块整合

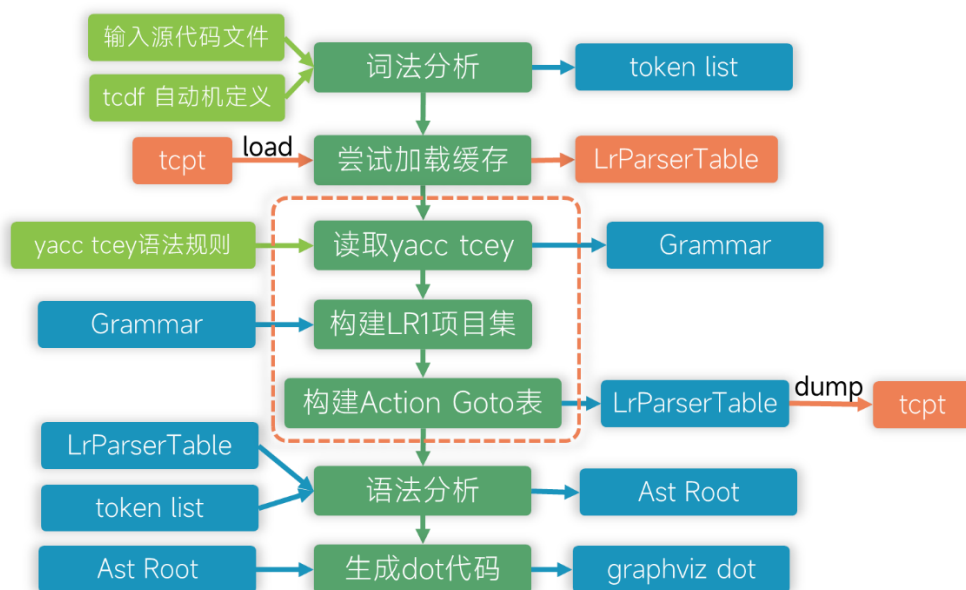
整合词法分析和语法分析模块，即可完成源代码读取到语法树构建的整体流程。为方便观察，基于得到的语法树生成 graphviz dot 代码，并通过 graphviz 程序将语法树可视化成 pdf 文件。

整体工作过程如下图所示：



从文法定义构建 Action Goto 表是十分费事的，需要花费数秒时间。考虑到对于相同的语法定义文件，生成的 Action Goto 表显然是相同的，我们选择将单次构建完成的表缓存到文件，并在下次启动程序时尝试从缓存直接加载，以提升启动速度。

改进后的流程如下图所示：

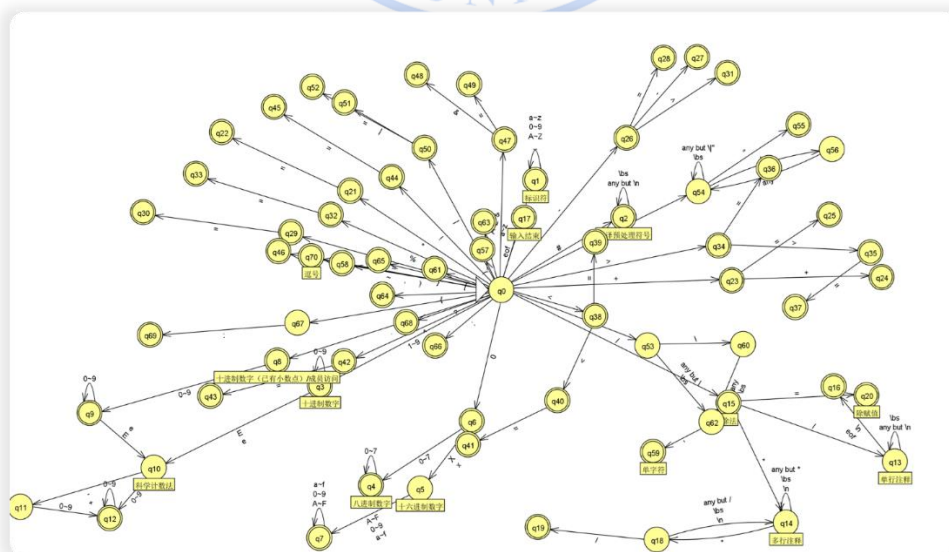


四、详细设计

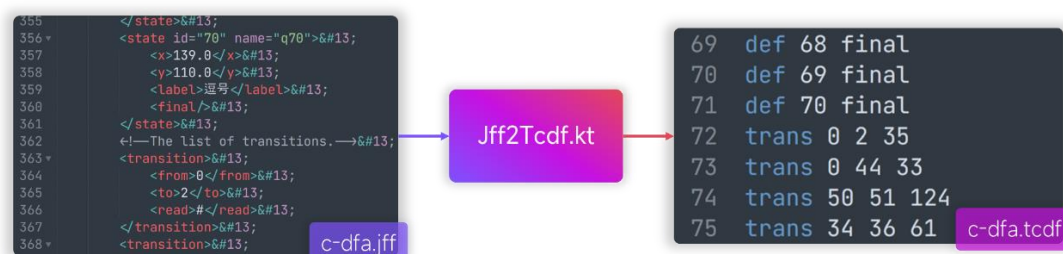
4.1 词法分析核心

4.1.1 DFA 构建

我们使用 JFLAP 软件绘制识别 C++ 语言字符的自动机。绘制得到自动机如下：



比较困难的事情。为此，我们制作一个 Kotlin 脚本程序，将 jff 格式文件转换成我们规定的一种较为简单的格式。我们称之为 tcdf (ToyCompile DFA Format)。转换后得到的描述文件十分简洁，利于程序读取。



基于 tcdf 格式文件，即可快速完成词法识别自动机的构建。自动机内包含三大成员，分别是节点列表、进入节点和转移表。当遇到待识别内容，自动机指向进入节点，并根据转移表不断切换状态。若遇到无法转换的情况，则将当前节点返回给调用者。调用者根据节点是否为接收状态判断该符号的识别是否成功。

tcdf 格式转换工具源码：

`tools/ToyCompileToolsKt/src/main/kotlin/Jff2Tcdf.kt`

dfa 定义和实现源码：

`src/include/tc/core/Dfa.h`
`src/core/Dfa.cpp`

4.1.2 词法分析器

为完成词法分析，首先需要构建自动机。由于自动机定义文件是从外部传入的，可能存在错误信息。自动机构建完成后，词法分析器会第一时间检查自动机构建过程是否遇到错误，并评估错误是否致命，进一步决定是将事件报告给用户或忽略。

此外，词法分析器还需要拥有一个可以将字符串形式的符号映射到内部符号类型的映射表。如，词法分析器需要知道 `int` 代表 `int`，而 `abc` 代表标识符。我们参考叶 my 学长组的项目²及 LLVM Clang 项目，巧用 C++ 编译过程的头文件展开机制，构建得到一个符号类型枚举类。同时，以单例模式构建一个映射表。

基于健康状态的自动机，以及外部传入的输入流，词法分析器不断将输入流送入自动机，在每次自动机退出后检查其状态，提取识别到的符号，直到抵达文件结尾。

对于每个符号，首先尝试在映射表内查找符号类型。若无法找到，则尝试转换成数字。对于以双斜线 (`//`) 和斜线星号 (`/*`) 开头的符号，分别标记为单行和多行注释。对于其他符号，标记为标识符即可。

如此，我们很优雅地完成符号列表的构建。

符号表定义源码：

```
src/include/tc/core/TokenKinds.h
src/include/tc/core/TokenKinds.def
```

词法分析器定义和实现源码：

```
src/include/tc/core/Lexer.h
src/core/Lexer.cpp
```

4.2 语法分析核心

4.2.1 Tcey 格式

本项目选择使用 C99 Yacc 语法作为识别参考。然而，该语法中的终结符定义依赖一个 Lex 文件⁸，我们已经使用其他方式完成该文件的功能。因此，我们只能选择对 yacc 规范做一定改变，令其适应我们的程序。

我们在 yacc 的基础语法上额外添加“可解析注释”，在其中对所有终结符做补充定义。我们称改变后的 yacc 格式为 tcey (ToyCompile Extended Yacc)。

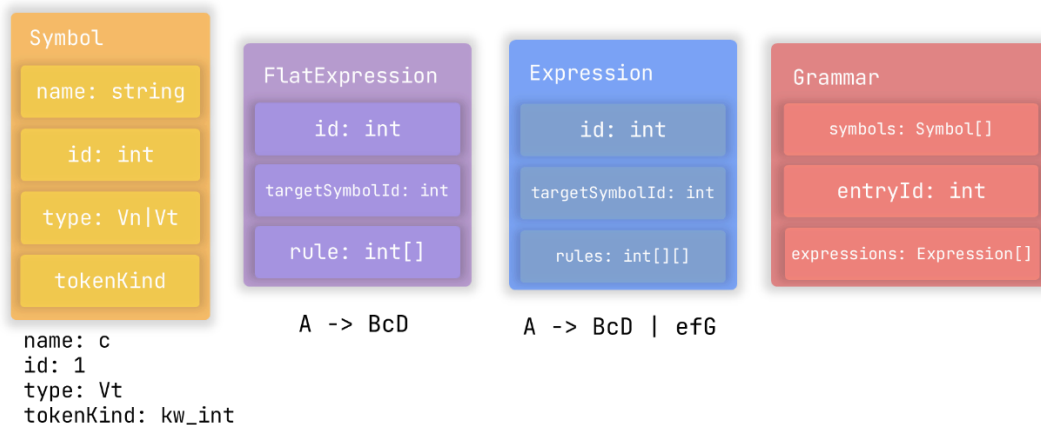
YaccTcey 解析器定义与实现源码：

```
src/include/tc/core/YaccTcey.h
src/core/YaccTcey.cpp
```

4.2.2 语法

为更好地将 yacc（或 tcey）应用到后续分析过程，我们需要先将其文件形式转换成更易使用的机内数据结构。

我们定义“语法”及相关结构如下：



Symbol 即符号。符号可以是终结符，也可以是非终结符。对于终结符，额外记录其属性。每个符号分配一个唯一的 id，供外部关联识别。

Yacc（或 tcey）内直接记录的产生式对应“语法”内的 Expression 结构。其中，targetSymbolId 为产生式左值，rules 则为每一条用“或”连接的语法规则。

符号列表与表达式列表，辅以文法开始符号 id，则构成一个语法。

当然，带“或”的表达式是不好处理的。为此，额外定义一个 FlatExpression 结构，表示不含“或”定义的产生式。

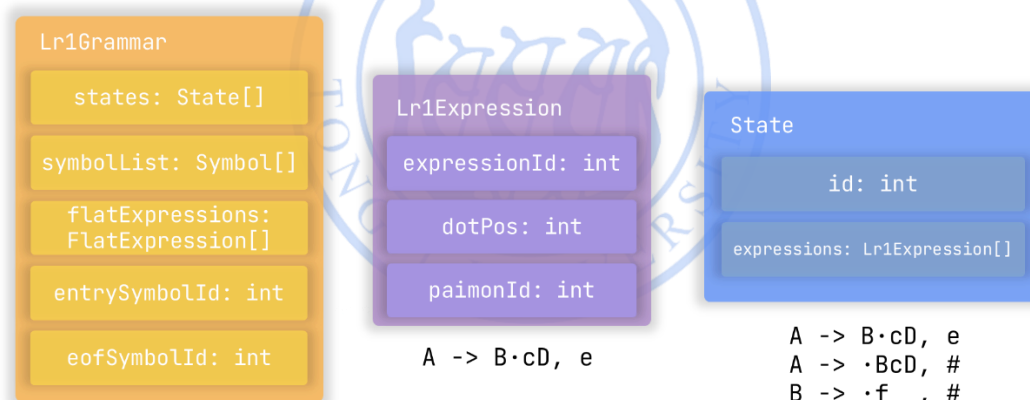
语法结构定义及辅助方法源码：

```
src/include/tc/core/Grammar.h
```

```
src/core/Grammar.cpp
```

4.2.3 LR1 语法数据结构

基于前面步骤得到的语法，构建 LR1 项目集，后者可以轻松转换成供语法分析器直接使用的 Action Goto 表。



LR1 语法由多个项目集（状态）组成。每个状态包含一系列 LR1 表达式。

我们用 Lr1Expression 结构表示一条 LR1 表达式。其中，expressionId 关联到一条 FlatExpression，dotPos 是 LR1 表达式内“·”所在的位置，paimonId 对应表达式后跟随的字符。

对于表达式“A->B·cD, e”，expressionId 对应的表达式为“A->BcD”；dotPos 为 1，代表“·”位于 1 号位置。paimonId 对应的符号是“e”。

可以看到，在我们的语法分析模块，“id”是一个很重要的东西。它可以避免很多重复的拷贝，在略微提升性能（减少内存拷贝）的同时大幅度降低内存占用。相比在每个状态和每个表达式内存储完整的符号序列，我们选择仅存储符号 id 序列，并额外维护一个符号

表，降低冗余存储带来的开销。同样，我们维护一个表达式列表，供每个 LR1 表达式关联使用。

4.2.4 LR1 项目集构造：准备与提取表达式

准备好 LR1 文法的程序数据结构，即可开始构建了。

考虑到开发者可能有在同一个 LR1 文法对象上构建不同语法的需求，我们需要先清空结构内已有数据，即清空现有状态集、表达式列表和符号表。

考虑到输入语法的产生式是包含“或”关系的，不利于分析，我们应即将其转换成不含“或”关系的产生式集合。展开后，得到超过一千条产生式。

与此同时，我们为每个非终结符构建一个产生式列表，以便快速寻找该符号对应的所有产生式，而不是从整个产生式集合中一个个查找。

4.2.5 LR1 项目集构造：构造 first 集

LR1 项目集构造过程中，会频繁使用到文法内各个符号的 first 集合。因此，我们应提前将它准备好。

首先，我们为每个终结符构建 first 集合。这是十分容易的，因为每个符号的 first 集合内只有它自己。

之后，对于每个非终结符，将其每条产生式的首字母的 first 集合元素融入该符号的 first 集合内。一轮融合过后，若存在某个符号的 first 集发生改变，则重新执行整个融合流程，直到所有符号的 first 集合不再改变。这种做法效率较低，但考虑到数据量很小，且相对高效的作法会带来循环依赖问题，得不偿失，我们选择坚持现在的做法。

4.2.6 LR1 项目集构造：拓广与引入文件结尾符号

LR1 文法需要拥有唯一的接受状态。为此，我们需要进行文法拓广，即对于开始符号为 S 的文法，补充定义产生式 $S' \rightarrow S$ 。我们将该产生式添加到产生式列表，并分配 id（正如对每条产生式一样），并将 S' 添加到符号集，同样分配 id。之后，文法的开始符号从 S 变为 S' 。

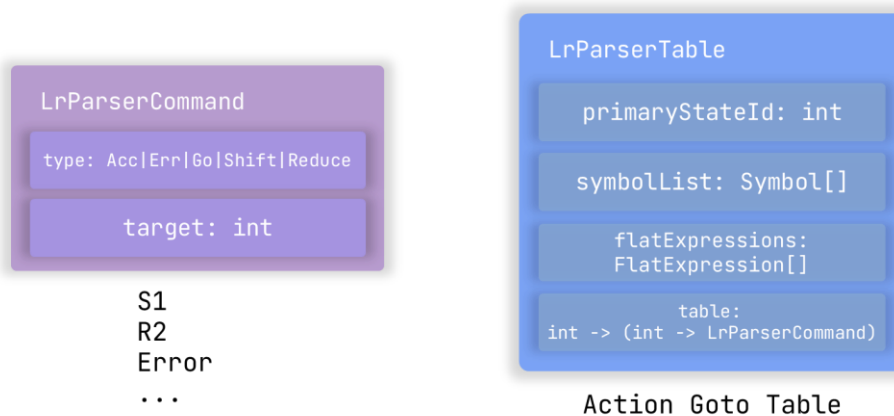
由于原始文法定义不包含文件结尾字符（eof），LR1 分析过程需要这个符号，我们对其进行补充定义，并添加到符号列表末尾。

4.2.7 LR1 项目集构造：求闭包

每当构建项目集，求闭包是不可少的。因此，我们统一设计一个求项目集闭包的方法，供后续分析过程使用。

4.2.10 Action Goto 表及其构造

首先，我们需要定义 Action Goto 表的数据结构。



我们希望语法分析只依赖分析表，而不关心分析表的构建过程。因此，我们在分析表内完整存储符号列表和表达式列表。

一般情况，初始状态编号为 0。但是，如果有人希望以其他数字代表初始状态呢？为此，我们将初始状态的编号存储到表内，而不是固定设置为 0。

表格关键部分即为 table 结构。通过当前状态和下一个符号的 id，取得一条指令。该指令可能是移进，可能是归约，或者是错误等。

由于整张表格十分庞大，我们选择仅记录移进、归约、接受和转移指令，“错误”指令不存入表格。当无法从表格内得到转移命令时，自动构建一条“错误”指令供分析器使用。

语法分析表定义与辅助方法源码：

```
src/include/tc/core/LrParserTable.h  
src/core/LrParserTable.cpp
```

4.2.11 语法分析

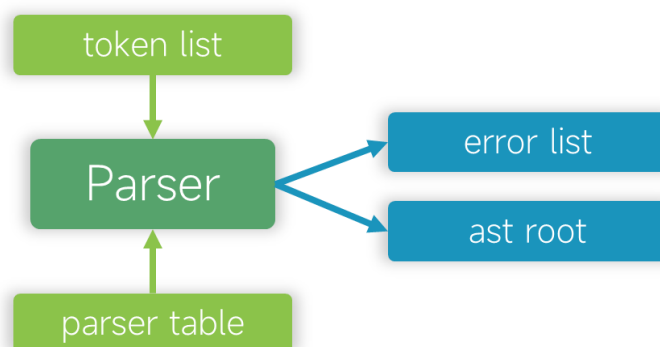
正式开始语法分析之前，我们需要补充定义语法树节点结构。



对于每个节点，需要记录其上级节点信息，孩子节点列表，以及该节点自己的符号。
对于终结符，需要额外记录其具体词符号信息。

好了，现在可以正式开始语法分析了！

语法分析器的工作定位很简单，如下图所示：



它接收一张分析表，一个符号串列表，产生一个错误列表和一个语法树根节点。该语法树的内存由分析器负责管理，即当分析器生命周期结束时，会同步回收语法树的内存。当然，外部可以通过一定方法接管对语法树内存的管理权。

分析器内部维护一个状态栈和一个符号栈。根据每个状态和下一个符号的 id，从分析表获取操作指示。若遇到错误，则登记到错误列表，并停止分析。

当遇到归约指令，会创建一个新节点，将产生式右侧符号出栈并绑定到该节点内，再将该节点放入符号栈内。如此分析，直到遇到“接受”命令。此时，符号栈内的首个元素（也是唯一的元素）即为语法树的根节点。当然，程序内实际会对该节点进行祖先节点查找操作，以避免一些意想不到的结果发生。

语法树定义与辅助方法源码：

```
src/include/tc/core/AstNode.h
src/core/AstNode.cpp
```

语法分析器定义与实现源码：

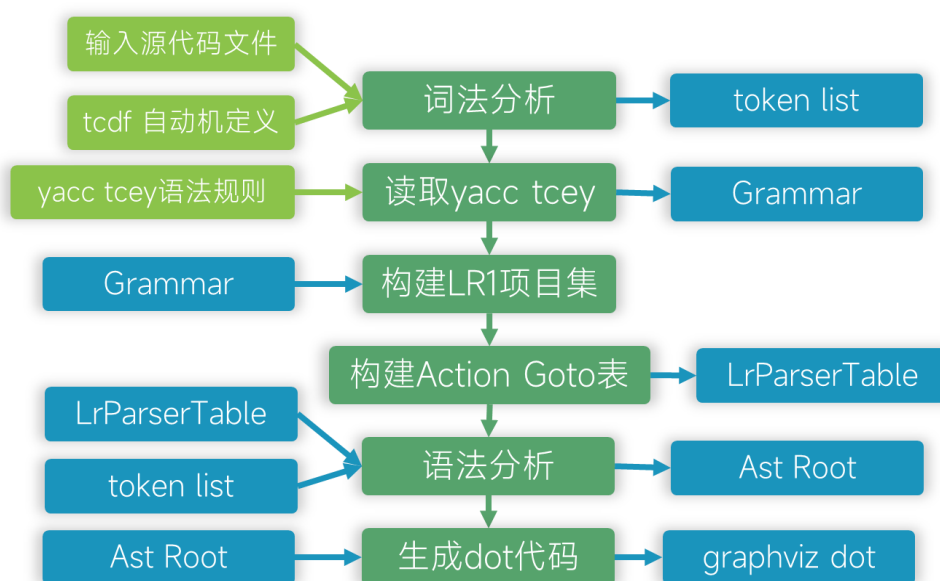
```
src/include/tc/core/Parser.h
src/core/Parser.cpp
```

4.3 语法分析命令程序

4.3.1 基本使用

灵活使用词法分析器和语法分析器，即可针对单源文件构建语法树。

完整分析流程如下：



分析过程中，一旦遇到错误，会立即将错误报告给使用者，并停止后续工作。

4.3.2 graphviz dot 代码生成

为更清晰地观察到语法树，我们选择采用 graphviz 工具进行简单的可视化。语法分析命令程序在成功得到语法树根节点后，会按照 dot 语言的语法规则生成 dot 绘图命令。对于得到的 dot 命令，使用 graphviz 将其转换成 pdf 格式或 jpg 等格式，即可清晰地看到语法树结构。

4.3.3 缓存优化

对于相对复杂的语法规则（如我们采用的完整 C99 语法），构建 Action Goto 表是一件很令人头疼的事情。仅产生式就有超过一千条，状态集更是超过一千五百个，带给用户的直观感受是整个分析过程十分缓慢。

考虑到对于同样的语法规则，构建得到的 Action Goto 表应该是相同的。我们选择在单次构建后，将该表存储到本地文件。后续启动时，首先尝试加载缓存，仅当加载失败才重新构建，以此实现瞬间启动。

当然，这一步是否执行，以及如何执行，是由用户通过命令行控制的。我们的程序为用户提供一个友好的操作说明，让程序按照用户希望的方式运行。

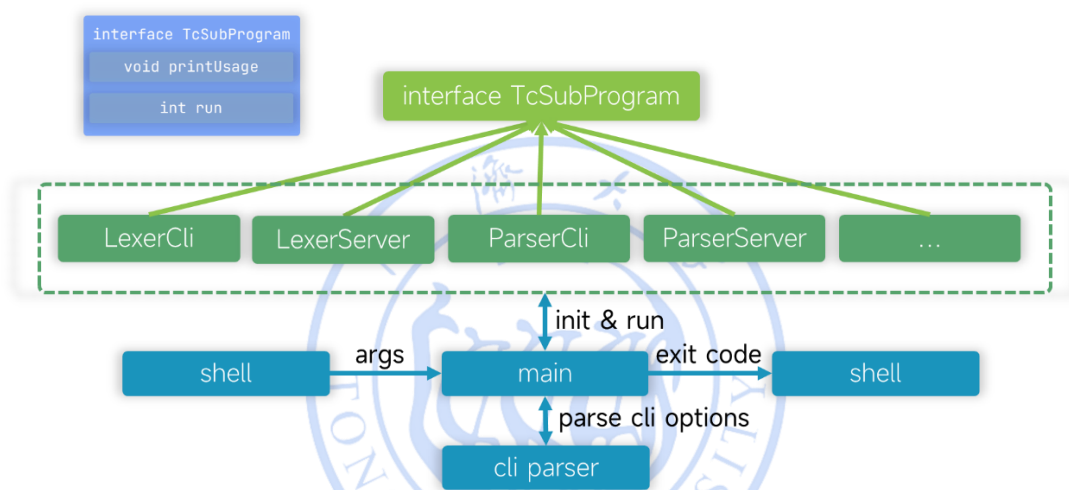
语法分析命令程序定义与实现源码：

```
src/include/tc/main/ParserCli/ParserCli.h
src/main/ParserCli/ParserCli.cpp
```

4.4 ToyCompile 项目框架

本项目建立的目的之一是巩固小组成员对课程知识的学习掌握。同时，我们希望为未来的学习者提供参考样本。此外，本项目存在多种展现形式，不限于本地命令行与 C/S 模式。因此，我们希望将整个项目的用户交互层做成子程序模式，词法分析命令程序、词法分析服务器、语法分析命令程序等皆为挂在本项目下的子程序。对此，我们设计一个“子程序抽象类”，每个子程序继承自该抽象类，实现其“运行”方法。

主程序负责完成命令行参数解析，根据命令行参数创建合适的子程序，并传递处理过的命令行参数给子程序使用。子程序运行结束，主程序将子程序运行返回码呈递给外层运行时控制程序，然后退出。



按照如此结构，我们的项目变得更加清晰。以语法分析器为例，当我们希望在本机调试时，选择使用 `ParserCli` 子程序即可。当我们希望连接到远程前端，则可启动基于 WebSocket 技术的 `ParserServer`，为远程客户端提供服务。

对于学习者，如果只希望研究词法分析部分，直接查看 `LexerCli` 的实现即可。如果希望学习语法分析部分，则可查看 `ParserCli` 的代码，十分友好。

五、调试分析

5.1 词法分析测试文件

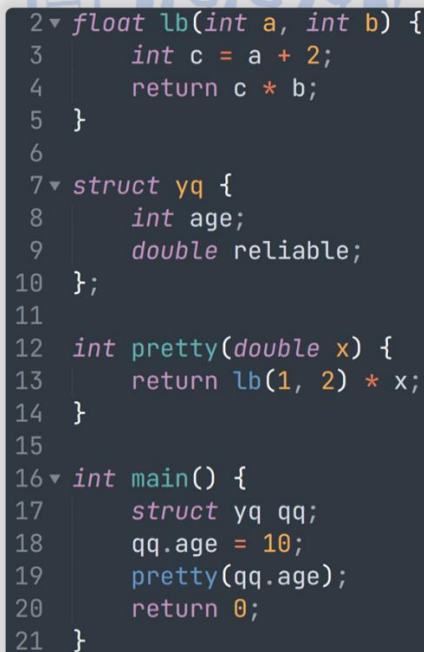
```
11
12 int pretty(double x) {
13     return lb(1, 2) * x;
14 }
15
```

我们将上图所示函数传入 LexerCli 程序，得到输出结果如下：

```
token
pos   : <12, 1>
kind  : kw_int
kind id: 82
content:
int
--- end of token ---
token
pos   : <12, 5>
kind  : identifier
kind id: 2
content:
pretty
--- end of token ---
token
pos   : <12, 11>
kind  : l_paren
kind id: 10
content:
(
--- end of token ---
...
```

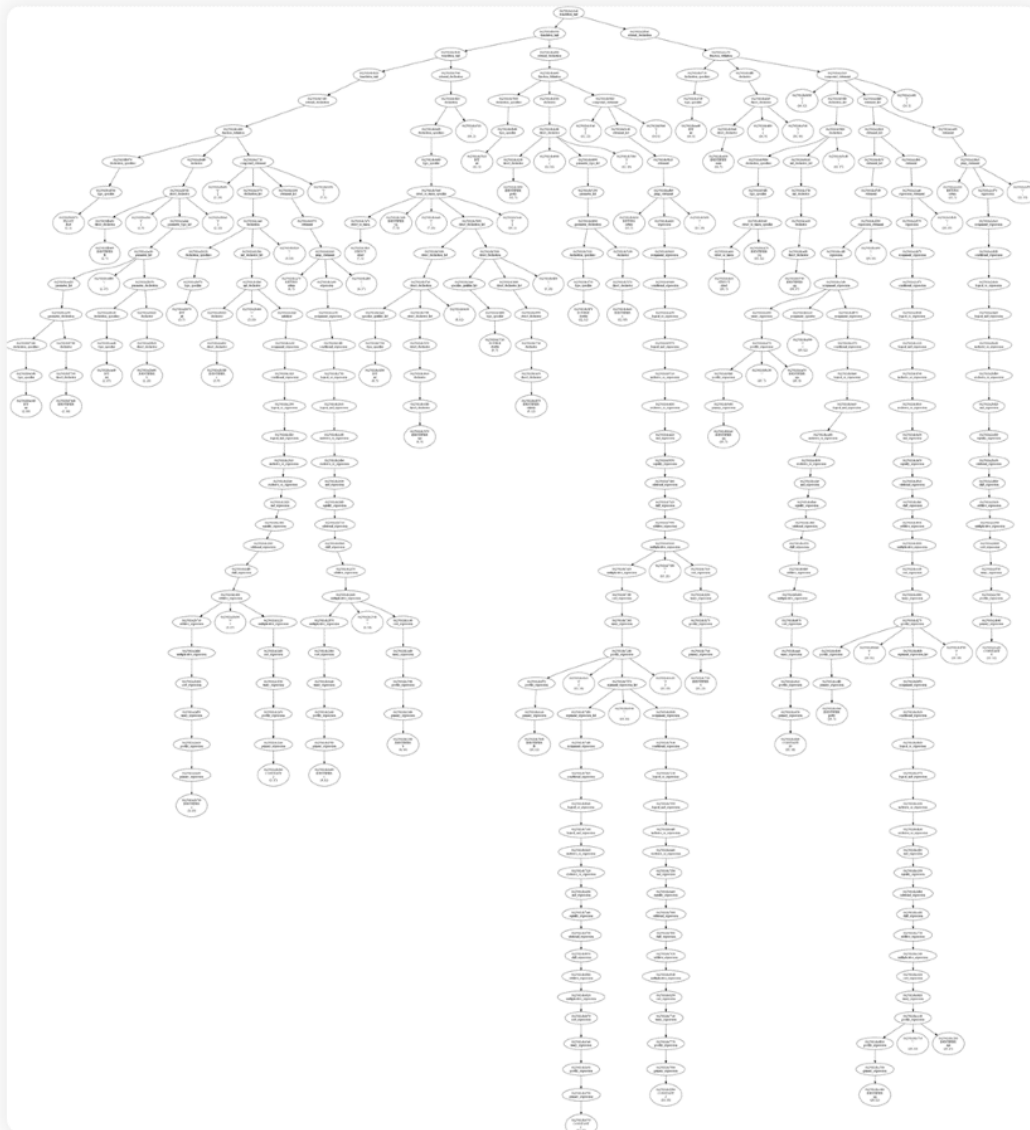
由于输出结果较长，仅展示前三个符号。可见，词法分析很准确。

5.2 语法分析测试 – 正确情况



```
2 ▾ float lb(int a, int b) {
3     int c = a + 2;
4     return c * b;
5 }
6
7 ▾ struct yq {
8     int age;
9     double reliable;
10 };
11
12 int pretty(double x) {
13     return lb(1, 2) * x;
14 }
15
16 ▾ int main() {
17     struct yq qq;
18     qq.age = 10;
19     pretty(qq.age);
20     return 0;
21 }
```

我们将上图所示程序送入 ParserCli 程序，将程序输出的 graphviz dot 格式代码编译可视化，得到下图所示语法树。我们称之为 YQ Tree。



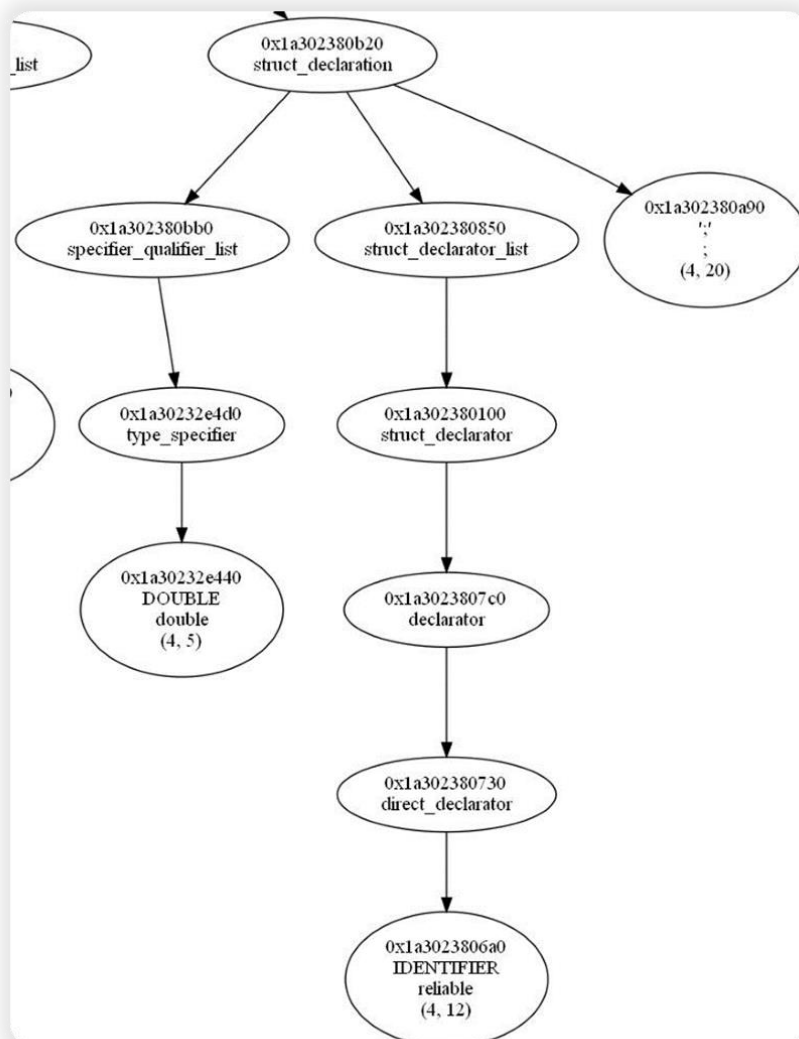
由于我们采用的是完成的 C99 文法，这棵语法树看上去非常大。我们将它上传到服务器，你可以扫描二维码获取原始图片，以便放大查看。



我们选取一个局部认真观察。

```
7 struct yq {  
8     int age;  
9     double reliable;  
10 };
```

上图代码的第 9 行对应的语法树如下图所示：



可以看到，我们的程序正确理解每个符号的含义，并构建得到这个局部的语法树。

与此同时，我们注意到，C99 语法定义十分严谨。平时看来很普通的语句，在标准的语法规则下，实际经历多层转换（归约）才能得到。

5.3 语法分析测试 – 错误情况

当然，输入的程序不可能总是正确的。我们希望语法分析器可以识别输入代码中存在的错误。

```
1
2 double f(tj int x) {
3
4     return 1.0 * x;
5 }
6
```

我们将上图所示代码输入 ParserCli 程序，得到如下输出信息：

parser error: (2, 13) unexpected token: int. at: (2, 13), int.

最初，我们认为代码中的错误字符是 tj。当分析器认为错误是 int 时，我们陷入沉思。但在观察语法树局部后，我们意识到，“tj”可能是此前定义过的数据类型。这时，int 会被理解成 tj 类型对象的对象名，恰好与保留字冲突。因此，tj 是正确的，错误的是 int。可见，我们的分析程序正确识别出输入代码文件中的错误情况。

5.4 性能分析

5.4.1 自动机

本项目子模块“自动机”内部转移表为基于红黑树的哈希表结构。若自动机转移数为 n ，单次转移的时间复杂度为 $O(\log n)$ 。事实上，我们可以将它换成简单的哈希表，以达到接近 $O(1)$ 的性能。

对于长度为 m 的符号，自动机识别所需时间复杂度为 $O(m \cdot \log n)$ 。

5.4.2 词法分析

假设不存在词法错误。词法分析过程，采用先识别，再提取的方式，相当于对整个文件进行 2 次扫描。其中，第一次扫描借助自动机完成。因此，对于包含 n 个转移的自动机，总长度为 x 的文件，总体时间复杂度为 $O(x \cdot \log n + x)$ ，可视为 $O(x \cdot \log n)$ 。

5.4.3 Action Goto 表构建

该过程涉及多个步骤。YaccTcey 格式文件解析的复杂度相对文件长度为线性。其中，涉及的映射表结构全部使用哈希表，以降低时间复杂度。

语法展开（去除“或”符号）的时间复杂度与语法所含子产生式数量呈线性。

first 集合构造的复杂度相对符号总数呈线性，但需要乘以一个不小的常数。

初状态构建的耗时操作发生在求闭包环节。求闭包的复杂度与产生式总数呈线性，但用时远不及线性时间多。

状态转移前，寻找转移符号的复杂度为状态内产生式的数量。转移过程复杂度与状态内产生式数量相同。之后则是求闭包。

状态表的构建是一件很耗时的事情。由于组成状态的可能性非常多，理论上复杂度可以达到指数级。我们可以很明显感受到，程序在这部分停留的时间最多。

构建转移表则是相对简单的事情。其复杂度与所有状态内所有产生式的数量总和呈线性关系。

5.4.4 语法分析

借助构建完毕的语法分析表，即可完成对符号串的分析。由于语法分析表采用二重哈希表结构，查询复杂度为常数级别。因此，语法分析所需时间复杂度与符号串长度呈线性。

5.4.5 语法分析整体开销

如果从文法定义文件构建 Action Goto 表，带来的时间开销主要与状态转移计算有关。该过程十分缓慢，需要花费数秒时间。

由于缓存优化机制的引入，当文法分析器找到此前计算过的 Action Goto 表时，会以线性时间复杂度直接加载并使用，达到快速启动的效果。实际使用时，基于缓存的分析器让我们几乎无法感知到停顿。

六、总结与收获

6.1 项目收获

本项目中，我们使用 C++17 语言完成词法分析和语法分析功能，它是整个编译器系统的最基础部分之一。研究过程中，我们不断巩固在课堂上学到的知识，并应用到实践。同时，我们也在努力优化项目结构，增强代码可读性，制作能够指引学弟学妹们参考学习的产品。通过本项目的开发，我们亦对 C++ 语言收获更多理解，并看到无数前人努力的身影。现在的我们拥有很幸福的开发环境。当我们打下一个字符，语言服务器会自动为我们提供输入提示，并进行潜在错误的分析。多文件项目的构建变得几乎无感。这一切都是前人努力的成果。

6.2 遇到的问题及其解决

研发过程中，我们坚持先规划后开发，在逻辑层面未遇到任何问题。然而，由于对 C++ 语法中的 auto 理解不到位，导致 LR1 分析器的内存指向出现混乱，使得构造的项目集乃至 Action Goto 表不符合预期，最终导致语法分析器无法正常工作。

我们根据语法分析器现象，在程序内部设立多处检查点，逐步定位问题。在经过三小时的检查后，我们成功找到错误源，并将其解决。

前文曾提到，我们的语法分析器带有缓存优化功能。该功能在项目设计初期并未考虑。但在制作完成后，Action Goto 表构造过程的卡顿令我们很不满，令我们立即设计并实现 Action Goto 表的导出与加载功能，并在其上制作缓存优化功能，提高程序性能。

6.3 课程认识

《编译原理》作为计算机系的一大重要课程，在整个教学体系内扮演不可替代的作用。作为计算机系的学生，学习该课程是研究计算机工作原理时必不可少的一部分。

课程上，我们学习到的是相对理论的知识，是对实际应用的抽象。而在课程设计项目中，我们将理论应用到实践，巩固课堂所学的知识，感受前人留下的智慧，收获颇丰。

正如枫铃树在文章《红黑树详解》结尾处所属，上世的智者将一个个伟大的火把交到我们手中，我们要牢牢握住，并用它照亮前方的路，点燃远方的灯⁹。上世的贤者们开创形式语言相关理论，提出相关方法，制作大量工具。他们的努力为我们带来无比幸福的学习研究环境。我们接过他们的理论，重温经典，力求在经典之上有所创新，将知识的火把一代代传递下去。

七、获取产品

7.1 获取源码

本项目在 GitHub 开源。前往项目仓库以获取源代码：

<https://github.com/FlowerBlackG/ToyCompile>

7.2 构建运行

项目使用 CMake 管理，依赖 Boost 库。对于 Windows GCC 环境，Boost 库已经内置。对于其他环境，需自行解决依赖问题。

程序运行依赖词法自动机定义文件（tcdf）和语法定义文件（tcey）。这些文件已经放置在 resources 文件夹内，默认情况会自动加载。

编译构建，启动构建产物 ToyCompile.exe，即可体验产品功能。

八、致谢

本项目的词法分析部分实现参考叶 my 学长所在小组的项目，以及高 qt 学姐的项目，从他们的项目中学到不少设计方式。同时，我们参考 LLVM Clang 项目中对于 token 类型的管理方式，并融入到我们的项目中。

我们的项目中，部分模块按照单例模式设计。单例对象的管理是一件比较复杂的事情。我们有幸得到张 h 学长的指导，以更优雅的方式实现单例对象的管理。

文法分析部分采用的 C99 文法规则由 Jutta Degener 整理并发布在网络上。我们在其版本上做简单修改后，作为文法分析器的输入材料之一。

项目研发过程，卫老师和曾 sr 同学一直在为我们答疑解惑，帮助我们更好地理解课程知识。

ToyCompile 的诞生与成长离不开每一个直接或间接帮助我们的个人和群体。在此，由衷对他们表达谢意。

九、参考资料

- [1] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社, 2000
- [2] Maoyao233. ToyCC. <https://github.com/Maoyao233/ToyCC>
- [3] GQT. 词法分析器. 2021
- [4] LLVM. Clang C Language Family Frontend for LLVM. <https://clang.llvm.org/>
- [5] jsoup. jsoup: Java HTML Parser. <https://jsoup.org/>
- [6] Cplusplus.com. std::stringstream - sstream.
<https://cplusplus.com/reference/sstream/stringstream/>
- [7] Jutta Degener. ANSI C Yacc grammar. 2004
- [8] Jutta Degener. ANSI C grammar, Lex specification. 2017
- [9] 枫铃树. 红黑树详解（下）（红黑树的删除操作）. CSDN, 2022