文章编号:1671-4598(2022)02-0151-09 DOI:10.16526/j.cnki.11—4762/tp.2022.02.022 中图分类号:TP311 文献标识码:A

基于 JavaCC 的抽象语法树生成错误处理技术研究

王国隆，金大海，宫云战
（北京邮电大学 网络与交换技术国家重点实验室，北京 100876）

摘要：随着 C++语言标准的不断演进，词法语法解析工具如 JavaCC 等对于很多扩充的新特性以及复杂的语法结构不能做到完全支持，这可能会导致抽象语法树生成错误且不完整；针对这一问题，提出一个针对抽象语法树生成错误的处理框架；首先，通过对 JavaCC 的扩充，实现一套可以解析 C++语言的词法语法分析器，生成抽象语法树并记录报错行；其次，根据报错行寻找所在函数区间即不支持或不匹配的语法片段；最后，通过注释函数区间的方式来跳过不支持或不匹配的语法片段进行错误处理并迭代生成抽象语法树；实验结果表明，对抽象语法树生成进行错误处理后可以更全面的分析代码，抽象语法树完成率上升 37.8%，分析行数提高 3.9 倍。

关键词：抽象语法树；C++程序；JavaCC 解析工具；语法变更；错误处理

Research on Error Handling Technology of Abstract Syntax Tree Generation Based on JavaCC

WANG Guolong, JIN Dahai, GONG Yunzhan
(State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: With the continuous evolution of C++ language standard, lexical parsing tools such as JavaCC cannot fully support many expanded new features and complex grammatical structures, which may lead to incorrect and incomplete abstract syntax tree generation. Aiming at this problem, an errors processing framework generated by abstract syntax trees is proposed. Firstly, through the extensible JavaCC, a set of lexical parsers that can parse C++ compiler are implemented, the abstract syntax trees and record error lines are generated. Secondly, look for the grammatical fragments that are not supported or matched in the function range according to the error line. Finally , By annotating the function intervals, the skip unsupported or unmatched grammar fragments for error handling and iteratively generate an abstract syntax tree. The experimental results show that the code can be analyzed more comprehensively after the error processing of the abstract syntax tree generation. The abstract syntax tree completion rate increases by 37.8%, and the number of analysis lines improves by 3.9 times.

Keywords: abstract syntax tree; C++ program; JavaCC parsing tool; syntax change; error handling

0 引言

为保障计算机软件质量，应尽早进行软件测试，而软件测试的重要手段之一就是静态测试[1-5]。随着 C++语言标准 11/14/17 的不断演进，新标准对 C++语法进行了诸多补充和优化，同时也带来了许

收稿日期:2021-11-30； **修回日期**:2021-12-27。
基金项目:国家自然科学基金(U1736110)。
作者简介:王国隆(1997-),男,黑龙江哈尔滨人,硕士,主要从事软件测试、缺陷检测方向的研究。
通讯作者:金大海(1974-),男,辽宁沈阳人,副教授,硕导,主要从事软件测试、缺陷检测方向的研究。
宫云战(1962-),男,山东威海人,教授,博导,主要从事软件测试、软件可靠性方向的研究。
引用格式:王国隆,金大海,宫云战.基于 JavaCC 的抽象语法树生成错误处理技术研究[J].计算机测量与控制,2022,30(2):151-159.

多扩充的新特性[6]。随着 C++新标准在市场上大面积的普及应用，对支持 C++新标准的静态测试也显得尤为重要。

目前存在很多用于 C++静态测试的工具，如 Cpplint[7]，PMD[8]，Cppcheck[9] 等。在这些静态测试工具中都必须对代码解析。抽象语法树是对代码的一种重要的中间表示形式，也是一个对代码进行静态测试的不可或缺的数据结构，在代码测试分析领域有着广泛的应用[10]。

目前存在的很多用于 C++语言的词法语法分析工具，如 JavaCC[11]，ANTLR[12]，LEX/YACC[13] 等不能完全支持 C++新标准的某些特性。而文中采用错误处理技术，较好地解决了这个问题，它可以确保对抽象语法树生成出现错误的地方采取对应的策略进行错误处理并完成抽象语法树的创建。在本文中，提出了一个针对抽象语法树生成的错误处理框架，用以解决抽象语法树生成错误问题。

1 相关工作

在词法语法分析工具方面，彭虎臣[14]以 LEX 作为词法分析器，读入字符串后根据词法规则，将单词或者字符转换为 token；采用 YACC 作为语法分析器，通过在符合 BNF 范式的语法规则中嵌入语法动作，之后搭建抽象语法树提取网页中 CSS 部分。Liu 等[15]采用用户自定义的语法规则及词法规则，利用 ANTLR 来生成相应语法分析器及词法分析器的代码。之后，首先把输入的字符流，通过词法分析器转变为由 token 组成的流，然后利用语法分析器的输出获得最后的抽象语法树进行 Scratch 语言的特征提取和检测。黄松等[10]使用纯 Java 代码编写的免费编译工具 JavaCC，经过用户自定义的词法语法规则文件 jjt 生成抽象语法树。肖一飞[16]提出一种基于 JavaCC 的通用的缺陷检测预处理方法，通过修改 jjt 规则文件对不同类型的词法异常和语法异常进行支持解决。孟春辰[17]提出一种基于 JavaCC 的中间文件化简的方式，通过保留一些类型定义信息从而避免了对头文件中导致静态分析失败的复杂语法结构的分析。本文鉴于 JavaCC 的易用性以及平台无关性等优势继续使用 JavaCC 作为抽象语法树的生成工具。

但是，JavaCC 也有缺点。JavaCC 遇到语法错误或者不匹配的语法时，仅仅会报告第一处错误并停止分析。也就意味着，对于代码的抽象语法树生成错误且不完整，所以需要对此进行错误处理。

对于错误处理方面，罗海丽[18]提出抛弃输入记号直到某个定界符，JavaCC 默认的错误处理就是使用了跳过字符到指定符号的方式，但是这种抛弃可能会引入更多的错误；郝丽波等[19]提出受到最大重复次数约束的可重试的错误处理策略，但是对于 JavaCC 来说不做出改动每次生成结果都是一样；Jia 等[20]提出了可替换的对输入做局部修改的错误处理策略，但是很难猜测符合意愿的替代方式；曾祥文[21]提出了一种可以回退 K 个的分析器，使用两个分析栈，一个栈负责将新单词压入栈，另一个栈负责维护 K 个单词，相当于对 K 个单词的窗口进行修复。由于无法预测出错的常见形式和替换方式，本文使用跳过符号的方式进行错误处理。

现有的词法语法分析工具对 C++新标准支持的不多，一些研究人员是面向 C++98 标准构造抽象语法树并进行分析，与他们工作不同的是，本文是面向 C++新标准生成抽象语法树并对生成错误进行处理，此方法既可应对不支持或者不匹配的语法片段，也可应对 C++日后的语法更新。

2 抽象语法树生成错误分析

2.1 抽象语法树生成错误原因

抽象语法树（AST，abstract syntax tree）是以树状的形式表达源代码语法结构的一种表现形式[17]，用 $T=<N, E>$ 表示，其中：N 为树的节点，表示代码中的一种语法结构；E 为树的边，表示代码中的语法逻辑关系。

抽象语法树生成过程如图 1 所示。

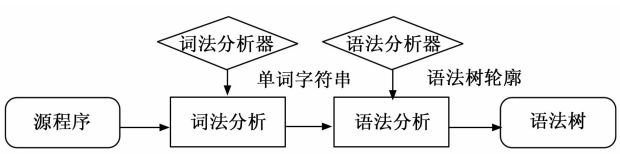

图 1 语法树生成过程示意图

源程序经过词法分析和语法分析生成抽象语法树。BNF（Backus－Naur Form）是描述编程语言的文法。词法分析和语法分析[22]根据对应的符合 BNF 范式的规则文件生成对应的语法树节点组成语法树。

词法分析错误可以通过在规则文件中添加新 token 来支持，语法分析则需要针对语法逻辑对规则文件进行修改。现有的规则文件可以较好地支持 C++98 标准和 C++14 标准的绝大部分语法和特性，但是针对语法逻辑修改的规则文件不能做到对不断迭代的 C++新标准的完全支持，进而就会产生语法树生成错误。

2.2　语法规则文件不能完全支持的原因分析

语法规则文件不能完全支持 C++新标准的原因主要有以下 3 点：

2.2.1　C++新标准 BNF 范式无法获取

官方的 C++新标准的 BNF 范式文档无法获取，并且网上存在的新标准 BNF 范式各有出入，无法确定是否和官方一致。如果范式文档选择出现问题，会对后续分析处理产生重大影响。

2.2.2　C++新标准语法结构变化大

C++新标准语法更改中，提出了新的语法结构，新的功能，在新关键字的配合之下或不需要新的关键字，在原有关键字的基础之上，提出新的结构，完成新的功能。这主要目的是为了更加方便地实现相应的功能。以语法树中 statement 节点的语法结构的变化为例，如图 2 所示。

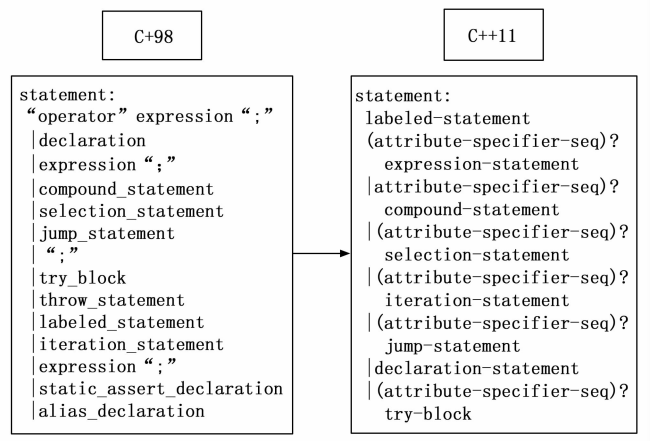

图 2　C++新标准语法结构变化示例

标准更新后出现更多的是语法结构的变化，有些结构是在原有结构的基础之上进行扩展，这种在构建抽象语法树的时候相对容易进行更改，但是很多结构的更改是在推翻原有结构的基础上进行重新架构（如图 2），同时还会糅合其他新的结构进来，层层嵌套。这种在破坏原有结构的基础上进行的更改在构建抽象语法树时相对困难。

由于破坏原有的语法结构，所以在修改抽象语法树规则文件时，也需要对相应结构进行破坏，这样无法保证在破坏现有结构后仍然可以对原有结构进行支撑，这是较难的问题。对于这种问题，一方面对新的语法尽量修改规则文件进行支撑，无法支撑的语法需要通过错误处理技术进行处理。

2.2.3　C++新标准核心库变更大

C++新标准的库更乐于使用复杂的嵌套结构和模板类来对相关代码进行声明，所以结构变得相对复杂，而之前的 C++98 的库更多的是使用直接声明的方法。例如 C++98 和 C++11 核心库 iostream 的变化如图 3 所示。

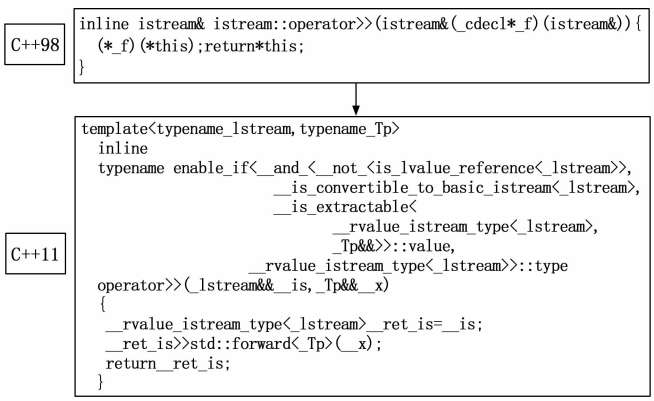

图 3　C++新标准核心库变化示例

可以看出，新标准的库文件结构变得异常复杂。因此，C++98 的库文件内容更加容易被抽象语法树支撑。由于测试时，为了获取到更全面的分析数据，会将全部的头文件进行展开，以获取到更多的信息来分析，获取分析结果。但是目前来说，在极度复杂的库文件的情况之下，很难做到对头文件的完全支撑。相反地，用户所写源文件的结构相对简单，不会大量使用复杂的结构，更加容易进行语法的支撑。所以建立抽象语法树时，构建的主体应为源文件，要做到不丢失源码信息。

综上，在对 C++新标准的语法逻辑做到最大可能的支撑的基础上，对于还不支持的会导致语法树生成错误的语法要进行错误处理。因为错误会导致语法树生成中断，错误点之后的源代码不能生成相关的语法树节点，导致对应源文件的语法树不完整，从而影响后续静态分析。所以本文重点讨论在

语法树生成中跳过不支持或不匹配的语法片段的错误处理技术。

3 针对抽象语法树生成的错误处理框架

对于语法树生成失败的源文件，希望通过错误处理技术可以继续生成语法树并且保证语法树尽量完整，也由此本文提出了针对抽象语法树生成的错误处理框架，框架设计如图 4 所示。

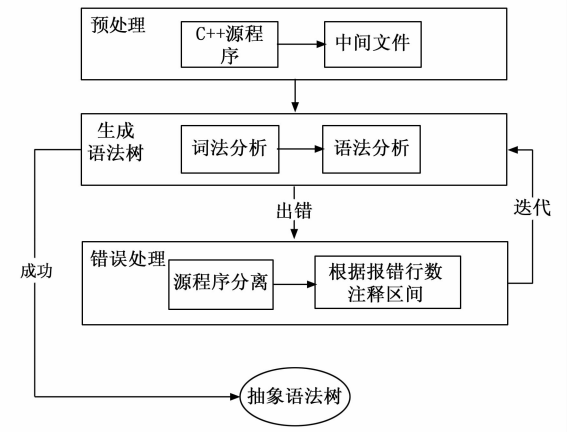

图 4 抽象语法树生成错误处理框架

首先，源代码经过预处理后生成中间文件。之后是一个迭代的过程，如果中间文件生成抽象语法树成功，直接结束；如果抽象语法树生成错误，则需要根据报错行数跳过附近语法片段以此继续生成抽象语法树直至成功。

3.1 预处理

中间文件（intermediate file）是在分析 C++程序时使用编译器对源代码进行预处理后生成的文件，以方便进行后续分析。

先要对 C++源文件进行预处理，预处理是通过编译器进行的，包括一些宏替换，条件编译以及头文件展开等操作。正是因为需要上述操作，所以不能使用 .cpp 文件直接进行处理，需要使用预处理后生成的中间文件 .i 进一步生成抽象语法树。

3.2 语法树生成错误处理

中间文件经过词法和语法规则文件生成抽象语法树，如果成功生成，当前文件分析完成；如果生成出错，那么接下来进入语法树生成错误处理流程。

JavaCC 是一种自上而下的语法处理器，在语法分析的每个时刻，token 流总是可以划分成如式（1）的形式[23]：

$$\alpha_1\alpha_2\cdots\alpha_n = w_1\alpha_i w_2 \tag{1}$$

式中，w_1 为已经分析并且生成语法树节点的部分；α_i 为当前分析的 token；w_2 为当前文件剩余的 token 流。

假定分析到 α_i 生成语法树时发生错误，因为分析是自上而下的，那就意味着当前已经建立了部分语法树，并且已经生成的语法树已经涵盖了 w_1，但是语法树却无法继续生成以涵盖 w_2。此时，就必须应用错误处理技术来继续生成语法树。可采取的措施如下：

1）删除当前 token α_i 并继续分析；

2）于 w_1 和 α_i 之间插进终结符号 α，从而把式（1）改写成式（2）：

$$\alpha_1\alpha_2\cdots\alpha_n = w_1\alpha\alpha_i w_2 \tag{2}$$

然后再从 α_i 开始分析；

3）从 w_1 的尾部删去若干个 token 后继续分析。

上述措施可以单独使用也可以结合使用，其中措施（2）直接添加终结符可能会导致程序不能正常编译。这里可以结合措施（1）和（3）进行错误处理，多次使用措施 1 并结合措施 3 进行处理相当于在式（1）中从 w_1 的尾部删除若干个 token 并且在 w_2 的首部删除若干个 token，即把式（1）修改为式（3）：

$$\alpha_1\cdots\alpha_{i-p}\cdots\alpha_i\cdots\alpha_{i+q}\alpha_2\cdots\alpha_n = w_0 U w_3 \tag{3}$$

式中，

$$U = \alpha_{i-p}\cdots\alpha_i\cdots\alpha_{i+q} \tag{4}$$

如果这样的 U 存在，就删除或注释 Uu 后继续分析。

基于以上措施，对于每一个 cpp 文件，先不考虑源代码中引用的头文件，只对纯源代码部分生成抽象语法树。如果在这个过程中出现异常，解决措施是不考虑报错行数附近的函数区间，继续分析其他可以正常生成抽象语法树的部分。

解决办法是，从完整的中间文件中分离出源程序部分，根据这部分用户写的源代码生成抽象语法树，如果发生异常，通过报错行数和函数区间标识算法略过附近函数区间后重新生成语法树直至成功。

模块流程设计如图 5 所示。

如图 5 所示，新语法错误处理模块可以分为 3 个部分：1）源程序分离；2）文件内函数区间划分；3）注释报错区间迭代生成抽象语法树。

3.2.1 源程序分离

在完整的中间文件中，有用户写的源代码部分，

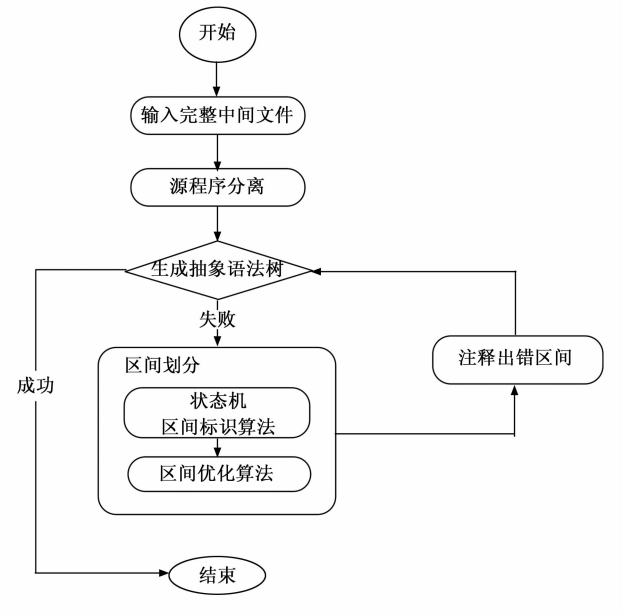

图 5　新语法错误处理模块流程图

也有头文件展开的部分，这里先不考虑头文件展开部分，只对源程序生成抽象语法树。所以，需要从完整的中间文件中得到源程序部分。

中间文件片段以及预期分离效果如图 6 所示。

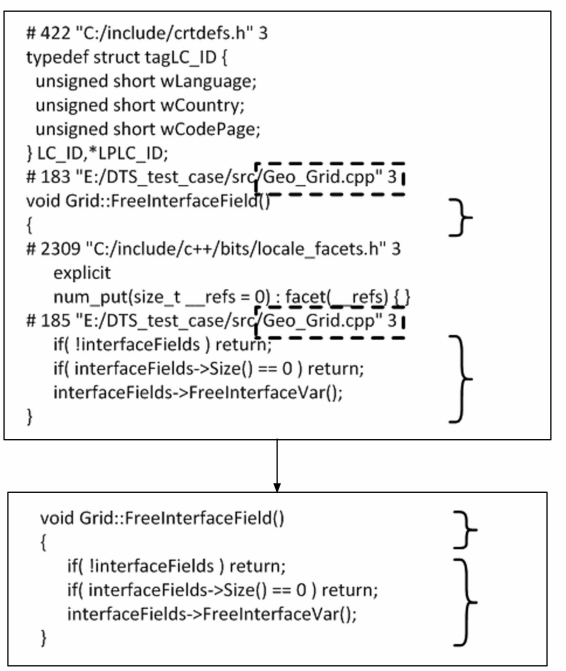

图 6　中间文件片段以及预期分离效果

孟春辰[17]根据“# 行号 文件存放路径”从后向前遍历，直到遇到文件后缀 .h 就结束遍历。但是对于图 6 代码，在源程序中还插入了一些扩展进来的头文件部分，之前的算法不能准确截取出源程序部分，所以提出了算法 1 所示的通用的源程序分离算法。

算法 1：源程序分离算法

输入：源文件名 fileName，完整中间文件 content；

输出：只包含源程序部分的中间文件 result。

```
index ← 0
isSourceLine ← false
rightPattern← "#\s(\d+)\s"(.*)" + fileName + ""\s(\d+)"
falsePattern ←"#\s(\d+)\s"(.*)"\s(\d+)"
while index < content.size() do
if line matches rightPattern do
index++
isSourceLine ←true
continue
end if
if line matches falsePattern do
isSourceLine ←false
end if
if isSourceLine do
result.add(line)
end if
index++
end while
return result
```

算法描述：算法从上向下扫描，通过两种正则表达式进行识别，一种表示含有源文件后缀的文件提示信息，另一种表示普通的文件提示信息，并且通过一个布尔值标识当前行是否加入结果集中，最后返回源程序部分代码。

3.2.2　文件内区间划分

对上述中间文件分离出的源程序部分生成语法树，如果发生错误，就需要根据报错行数注释相应的区间，所以接下来就需要对中间文件进行区间划分。

定义 1：函数区间：函数区间是文件内用来标识一个函数名称和范围的结构，该结构由一个三元组表示，该结构描述记为 $I = \{Name, StartLine, EndLine\}$，其中：

Name：表示该函数区间的函数名称；

StartLine：表示该函数区间的起始行数；

EndLine：表示该函数区间的终止行数。

文件内区间划分分为两部分：第一部分是函数区间的标识，第二部分是函数区间优化。

1）函数区间标识算法：函数区间标识算法通过状态机来实现，定义 11 种状态，利用互相的转化关系求出函数范围。状态机转化关系如表 1 所示。

表 1 函数区间标识状态机转化关系

状态\符号	\n\t\r	;	空格	(	,	)	{	}	:	"	′	\	others
0 起始	1	1	1	0	0	0	0	1	0	0	0	0	0
1 开始检测函数	1	2	1	4	2	2	2	2	2	2	2	2	2
2 类名::函数名	1	0	1	4	0	0	0	0	2	2	2	2	2
3 成员变量初始化列表	3	0	3	3	3	3	7	0	3	3	3	3	3
4 参数	4	0	4	0	4	6	0	0	4	4	4	4	4
5 "引号里	5	5	5	5	5	5	5	5	5	7	5	8	5
6 函数头结束	6	1	6	1	1	1	7	0	3	1	1	1	6
7 函数体开始	7	7	7	7	7	7	7	0	7	5	9	7	7
8 "引号里的转义符	5	5	5	5	5	5	5	5	5	5	5	5	5
9 ′引号里	9	9	9	9	9	9	9	9	9	9	7	10	9
10 ′引号里的转义符	9	9	9	9	9	9	9	9	9	9	9	9	9

主要算法是在函数头结束状态中遇到｛和｝的处理逻辑，具体的算法如算法 2 所示。

算法 2：函数区间标识算法

输入：文件路径 path；

输出：函数划分后起始行结束行的集合 list。

```
switch state do
……
case 7 do
if 3=lastState and 6=lastState then
startLine ←line / * lastState 表示当前状态的上一个状态 * /
end if
if c ='{' then
bracket ←bracket + 1 / * bracket 表示大括号的个数 * /
end if
if c ='}' then
bracket← bracket − 1
end if
if 0 =bracket then
endLine← line + 1
list.add(startLine, endLine) / * list 是保存起始行和结束行的集合 * /
end if
……
end switch
```

算法描述：查看状态机中的状态，状态 7 表示函数体开始，如果从成员变量初始化列表状态（状态 3）或者从函数头结束状态（状态 6）转换而来，需要记录函数开始行数 startLine.bracket 表示括号个数，当前字符 c 如果是｛符号需要增加括号个数，如果是｝需要减少括号个数，如果括号个数减为 0，则当前函数结束，记录相应的结束行数 endLine，随后加入到函数划分的集合中。

2）函数区间优化算法：对于区间划分后的文件中，会存在部分没有在任何区间中的代码行需要进一步被划分到现有区间中，区间划分后剩余部分如图 7 所示。

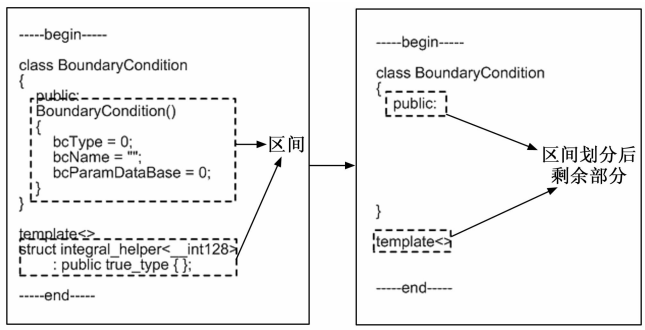

图 7 区间划分后剩余部分

可以看出，例如“public:”“＜template＞”等剩余行需要进一步划分进现有区间中，否则如果注释掉当前区间后这些行会残留导致语法树生成不精确。除此之外，还有换行函数头的剩余部分也是通过类似的区间优化算法进行进一步划分。函数区间优化算法如算法 3。

算法 3：函数区间优化算法

输入：划分区间 funcList，文件内容 content；

输出：经过区间优化的区间 funcList。

```
pattern← " #\s(\d+)\s"(. * )"\s(\d+)"
foreach func in funcList do
startLine ←func.startLine
while startLine > 1 do
behindLine ←content.get(startLine −2)
if behindLine.trim().length()=0 then
```

```
startLine－－
continue
end if
if lastChar =';' || lastChar ='{' || lastChar ='}' then
break
else if Pattern.matches(pattern, behindLine) then
break
else
startLine－－
end if
end while
func.startLine← startLine
end foreach
```

算法描述：从当前区间向上扩展，如果上面的行是以“;”或者“{”或者“}”结尾的，或者遇到形如“# 577 " C:/Program Files/stl _ multimap.h" 3”的标识文件名称的行就停止扩展，最后返回优化后的区间。

3.2.3　注释迭代生成语法树

生成语法树的报错信息可由定义 4 进行描述：

定义 2：语法树错误信息特征：语法树错误信息特征是用于对一个语法树报错信息进行特征描述的结构，该结构由一个三元组进行表示，语法树错误信息 E 的特征记为：E= {ErrorLine，TokenImage，TokenKind}，其中：

ErrorLine：表示产生错误信息的中间文件行数；

TokenImage：表示产生错误信息的 token 名称；

TokenKind：表示产生错误信息的 token 种类。

一次迭代可以通过语法树错误信息特征中的 ErrorLine 去之前划分的区间中查找对应区间并注释，再继续生成抽象语法树。从语法树角度来看，注释相当于从当前节点开始向上删除部分节点后继续向下生成语法树，如图 8 描述。

图 8 表示的是一次迭代的效果，注释区间之后重新生成语法树，如果还是失败继续根据 ErrorLine 注释区间直到成功生成为止。

注释并迭代生成抽象语法树的具体算法如算法 4 所示。

算法 4：注释迭代生成语法树算法

输入：文件路径 path；

输出：成功生成的抽象语法树根结点 astroot。

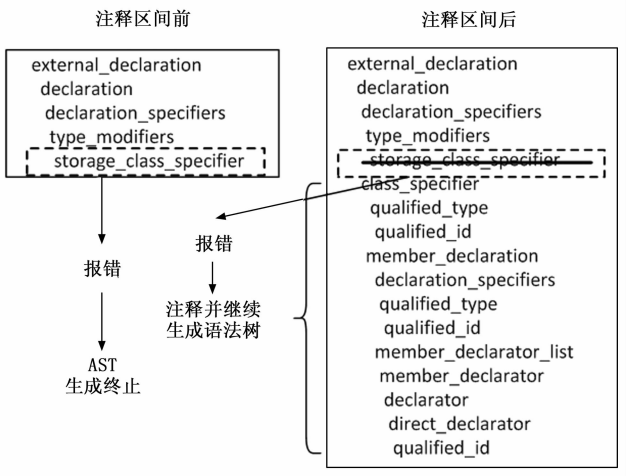

图 8　注释区间前后语法树效果示意图

```
flag ←true/*语法树生成成败的标志*/
try:
astRoot← createAST(path)
catch Exception e:/*生成语法树失败*/
flag ←false
errorLine ←e.getBeginFileLine()/*记录错误行号*/
if ! flag then
funcList ←funcDivide(path)/*调用函数划分算法*/
end if
while ! flag do
commentFunc(errorLine)/*注释出错函数区间*/
try:
astRoot ←createAST(path)
flag ←true
catch Exception e:
errorLine← e.getBeginFileLine()
```

算法描述：通过 flag 记录生成语法树是否成功，成功后跳出循环。通过 errorLine 记录具体生成语法树错误的行数。如果生成抽象语法树异常，根据报错的行数到文件内划分的区间中寻找对应区间并注释代码，之后重新生成抽象语法树，并迭代直至成功生成语法树。

4　实验及实验结果分析

本文提出了一种针对抽象语法树生成的错误处理框架并进行了实现。为了对此框架构建抽象语法树的完成度和错误处理效果进行验证，本文选取了应用 C＋＋新标准的 5 个开源工程利用本文开发的工具进行抽象语法树的生成对比实验。选取的测试工程如表

2 所示。

表 2 选取的测试工程列表

工程名	源文件数量/个	头文件数量/个	文件总数/个	代码数/行
cxx11tests	73	41	114	377
fl	762	1 867	2 629	191 133
libctiny	24	66	90	3 537
rbv8	117	626	743	102 511
tesseract	220	1 608	1 808	97 389

4.1 抽象语法树完成率

由于 JavaCC 自上而下的分析特性，在遇到语法不匹配的时候报错就结束了当前文件的分析，也就是报错行后的源代码并没有生成语法树节点，也就丢失了源代码信息。

假设开源工程中的源文件数量为 $totalNum$，分析完整源代码后生成的抽象语法树数量为 $astNum$，则本文定义完成率如式（5）：

$$\text{完成率} = \frac{astNum}{totalNum} \times 100\% \tag{5}$$

将表 2 中 5 个工程进行抽象语法树生成后结果如表 3 所示。

表 3 使用错误处理框架前后语法树完成率对比结果

工程名	$totalNum$/个	$astNum$/个		完成率/%	
		前	后	前	后
cxx11tests	73	48	67	65.7	91.8
fl	762	424	685	55.6	89.9
libctiny	24	24	24	100	100
rbv8	117	15	93	12.8	79.5
tesseract	220	120	215	54.5	97.7
平均	239	126	216	52.8	90.6

从表 3 可以看出使用错误处理框架后，超过 90.6%源文件的抽象语法树都是分析完全部源代码后生成的，其余部分源文件的抽象语法树可以分析部分源代码后生成。同时，使用错误处理框架前语法树完成率为 52.8%，使用错误处理框架后语法树完成率为 90.6%，这里提高了 37.8%完成率的原因主要是对于生成错误相关的代码区间进行注释，使得文件跳过错误后能继续生成语法树。

但是可以发现还是有 9.4%源文件没有在分析完全部源代码的基础上生成语法树，主要原因是因为个别源文件区间划分的结果没有包含报错行数，所以导致根据报错行数无法注释对应区间导致无法跳过出错的语法片段进而无法继续生成语法树。

4.2 抽象语法树错误处理效果

以上部分对抽象语法树能否能在分析完源代码后生成进行了测试，下面利用错误处理策略前后生成的抽象语法树进行分析行数以及时间相关的对比，效果见表 4。

表 4 使用错误处理框架前后分析行数及时间对比

工程名	分析行数		生成 AST 用时/个/ms		释并迭代用时/(个/ms)
	前	后	前	后	
cxx11tests	134	156	7.5	5.7	0.33
fl	14 593	120 060	7.8	40.5	64.93
libctiny	1 149	1 149	27	27	0
rbv8	2 508	59 961	132.3	210.2	237.52
tesseract	41 902	115 201	78.5	205.8	570.62
平均	12 057	59 305	50.62	97.84	174.68
错误处理前后对比上升率/%	391.9		93.3		78.5

表 4 对错误处理框架使用前后的抽象语法树生成用时，分析行数等方面进行了对比，并统计了注释迭代生成语法树的时间。经过错误处理后，单个文件语法树生成用时上升 93.3%，分析行数上升 391.9%，这是因为跳过出错的语法片段后，语法树生成不会因为报错而停止，而是继续分析，所以语法树更加完整，更多的代码也加入了分析。同时也伴随着额外 78.5%的注释并迭代生成语法树的时间开销，但牺牲了时间却换取了更完整的抽象语法树和更全面的代码分析，这是非常值得的。

但是注释的部分代码可能含有符号类型信息，导致后续生成的符号表中可能含有未知类型，这对后续的分析可能造成影响。

5 结束语

本文提出一个针对抽象语法树生成错误的处理框架，通过注释报错行所在函数区间来跳过不支持或不匹配的语法片段的方式进行错误处理并迭代生成语法树，用于解决词法语法工具对 C++新标准无法完全支撑的问题。实验结果表明，经过错误处理后语法树

完成率较高，有较好的错误处理效果；由于迭代注释并重新生成语法树，会导致分析时间上升。但以上代价所带来的效果是更全面的代码分析，这具有重要的价值。本研究不仅可以应对不支持或不匹配的语法片段，还可以应对 C++日后的语法更新。成功生成的抽象语法树可以应用于静态测试分析领域。在下一步的研究工作中，一方面可以改进函数区间标识算法，使得区间覆盖更加全面，另一方面可以优化注释区间范围，使得注释粒度更小影响更小，并通过这两个方面达到更好的错误处理效果；同时，如何提高错误处理效率也将是下一步研究的主要内容。

参考文献：

[1] 周　培. 基于 LDRA Testbed 的民用机载软件静态测试方法 [J]. 计算机测量与控制，2019，27 (7)：107－110，149.

[2] 杨晓庆. 软件测试技术现状与发展趋势研究 [J]. 电脑编程技巧与维护，2020 (4)：62－63.

[3] 陈建锋. 软件测试发展趋势研究 [J]. 无线互联科技，2019，16 (19)：41－42.

[4] 邓梅淇. 计算机软件测试方法及发展趋势软件开发与应用 [J]. 信息与电脑 (理论版)，2021，33 (8)：114－116.

[5] 杨丰源，谷　威，刘　静一，等. 面向 C/C++的静态测试技术应用研究 [J]. 工业控制计算机，2019，32 (6)：23－27.

[6] DUFFY D J，PALLEY A R. C++ 11，C++ 14，and C++ 17 for the impatient：opportunities in computational finance [J]. Wilmott，2017 (90)：38－47.

[7] NABIL R，MOHAMED N E，MAHDY A，et al. EvalSeer：an intelligent gamified system for programming assignments assessment [C] //2021 International Mobile，Intelligent，and Ubiquitous Computing Conference (MIUCC). IEEE，2021：235－242.

[8] TRAUTSCH A，HERBOLD S，GRABOWSKI J. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects [J]. Empirical Software Engineering，2020，25 (6)：5137－5192.

[9] 张仕金，尚赵伟. Cppcheck 的软件缺陷模式分析与定位 [J]. 计算机工程与应用，2015 (3)：69－73.

[10] 黄　松，黄　玉，惠战伟. 基于 JavaCC 的抽象语法树的构建与实现 [J]. 计算机工程与设计，2016，37 (4)：938－943

[11] GUPTA K，NANDIVADA V K. Lexical state analyzer for JavaCC grammars [J]. Software：Practice and Experience，2016，46 (6)：751－765.

[12] CHANG Z，SUN Y，WU T Y，et al. Scratch analysis Tool (SAT)：a modern scratch project analysis tool based on ANTLR to assess computational thinking skills [C] //2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC). IEEE，2018：950－955.

[13] LU J，JING L. Design of codes conversion compiler based on LEX and YACC [C] //Proceedings of the 2012 Second International Conference on Electric Information and Control Engineering，2012，2：518－521.

[14] 彭虎臣. Lex 与 Yacc 在提取 C 程序结构与网页 CSS 中的应用 [J]. 华章，2014 (18)：370－371，377.

[15] LIU P，SUN Y，LUO H. An ANTLR－based feature extraction and detection system for scratch [C] //2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)，IEEE，2019：1743－1748.

[16] 肖一飞. 支持多编译器的缺陷检测预处理方法研究 [D]. 北京：北京邮电大学，2016.

[17] 孟春辰. 静态分析中 C++中间文件化简方法研究及实现 [D]. 北京：北京邮电大学，2020.

[18] 罗海丽. 语法分析中错误恢复机制的构造 [J]. 内蒙古科技与经济，2007 (14)：71－72，74.

[19] 郝丽波，许靖祺. 网格工作流错误恢复方法研究 [J]. 计算机与数字工程，2012，40 (6)：91－94.

[20] JIA Z Y，LI S S，YU T T，et al. Detecting error－handling bugs without error specification input [C] //2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)，IEEE，2019：213－225.

[21] 曾祥文. 关于编译器设计中语法错误处理的研究 [J]. 中国电力教育，2007 (s3)：231－233.

[22] 李　泱，颜学龙，陈寿宏. 编译器技术在边界扫描中的应用研究 [J]. 计算机测量与控制，2014，22 (8)：2606－2608.

[23] 李　娟，王冬星. 错误处理技术的研究 [J]. 大庆师范学院学报，2008 (2)：17－19.