计算机科学2001Vol. 28№. 12

非结构化编辑环境下的增量语法分析

The Incremental Semantic Analysis for Language-based Editor

姜 宁 史忠植

(中国科技大学研究生院计算机学部 北京100039)(中国科学院计算技术研究所 北京100080)

Abstract With a table drived parser engine, in this paper, we discuss an incremental L R analysis algorism. The algorism is efficient and siultable for a language-based enitor generator. there are also some about incremental attribute evaluation based on this algorism.

Keywords Language-based editor, Incremental semantic analysis, Incremental syntax analysis

1 引言

基于语言的编辑器(language-based editor)是一种交互式的开发工具,根据语言的语法、语义,在编辑器中对输入的程序进行分析,及时地为程序员提供交互的信息提示,指出语法、语义错误,引导其输入。它也是构成分布式软件开发环境的重要部分,可以在多个开发者之间报告语义不一致,协调开发过程[9]。不同于结构化的编辑器,基于语言的编辑器采用了标准的文本编辑方式,没有对使用者的输入方式作任何限制。这使设计者面临两个挑战[1]:

1. 如何在文字编辑过程中从输入获得语法语义信息,传统的分析算法在速度上无法满足要求。

2. 如何处理输入中的错误,结构化的编辑器至少在语法层次上不会包含错误。

增量分析技术的提出使得实时地进行语法分析并完成静态语义的检查成为可能,这方面的研究至少从80年代开始,成果也非常显著:增量语法分析出现了Kiong and Welsh Parser(LL)[6],Limited Backtracking(LL)[8],Wagner and Graham Parser(LR)[2]等一系列的算法。

正象 Phil Cook[1]指出的,这些算法并不完善。有些算法对处理的语法作了过多的限制,一些算法存在无穷循环的可能;此外还有效率问题,大部分算法保存了太多额外的属性,而更新这些属性本身变得非常耗时。

相对而言,增量的语义分析算法更加成熟,但大多是为结构化编辑器所设计的,必须作相应的调整才能适应新的非结构化编辑环境。

文本提出一个新的算法的框架,克服了以往的算法中对语言语法过于严格的限制,并且使得编制类似于 YACC 的增量语法分析器构造程序成为可能,这有助于维护基于语言的编辑器与编译程序之间的一致性。算法适用于有冲突的 LALR(1)文法,期望可以采用类似 YACC 的程序自动构造语法分析过程。其中一些讨论主要考虑了 C++风格的语言,但并不失一般性。在最后,讨论了该算法对已有的增量语义分析算法的影响。

2 增量的语法分析

基于文本的增量分析的挑战就是效率,不可能每进行一次编辑操作就重新进行一次完整的分析。程序仅对编辑动作所波及的最小范围进行分析,速度大大快于相应的非增量实现[4]。

由于语法树的所有的叶节点从左到右连接起来,即是源程序中的记号序列,保留了增量语法分析所需要的信息,因此在下面的增量分析的讨论中,将基于语法树,而不是一般编译中的分析树。

一般地,由于语法中存在错误,分析过程得到的并不是一棵完整的语法树,而是一个语法子树的序列,序列中的所有树叶对应了输入的记号序列。当用户修改了一系列的词法记号之后,分析程序应该从第一个变动的记号处开始重新分析。重新分析的过程涉及这样三个关键问题:1)为了提高分析的速度,如何利用以前的分析中已构造的语法树;2)为了从变动点开始新的分析过程,如何将分析栈、值栈恢复到移入变动点之前的状态;3)如何处理语法错误,进行错误恢复。

2.1 扩展的无冲突 LR(1)项目集合的 DFA

姜 宁 硕士生,研究方向为人工智能、软件开发环境与开发技术、网络。史忠植 研究员,博导,主要从事人工智能、神经计算和多媒体信息检索的研究。

在增量分析的过程中，完成了对变动区间的分析后，需要继续对余下的没修改的记号进行重新分析。这是一个耗时的过程，必须通过某种途径利用前次分析产生的语法树序列来加快重新分析的速度。如果观察修改前后的两次分析过程，可以注意到大部分的语法树或其子树都是相同的。比如对一般的程序设计语言而言，函数内某条语句的修改对语法分析产生的影响，不会波及到后面的语句和后面的函数定义。因而有理由直接将变动区间后面的语法子树（对应于非叶节点）直接"移进"值栈，分析栈则按照分析表中 goto 部分的要求操作。这里通过将前次分析产生的语法树序列作为输入，代替了记号序列的输入，缩短了输入的长度，从而降低了分析的复杂性。判断语法子树是否可以直接"移进"的充要条件是相应的记号串在前后两次分析中能否产生相同的语法树结构。

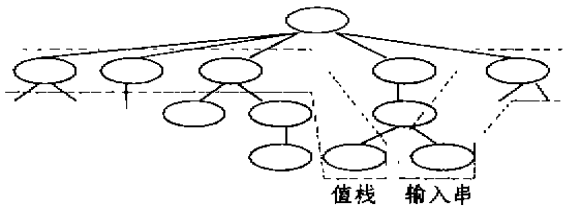


图1　增量分析中值栈与输入中都可以出现语法树的非终结符节点

为了清晰起见，下面的讨论针对 LR(1)分析，并合理地假定文法中不存在形如 E→ε 的产生式。

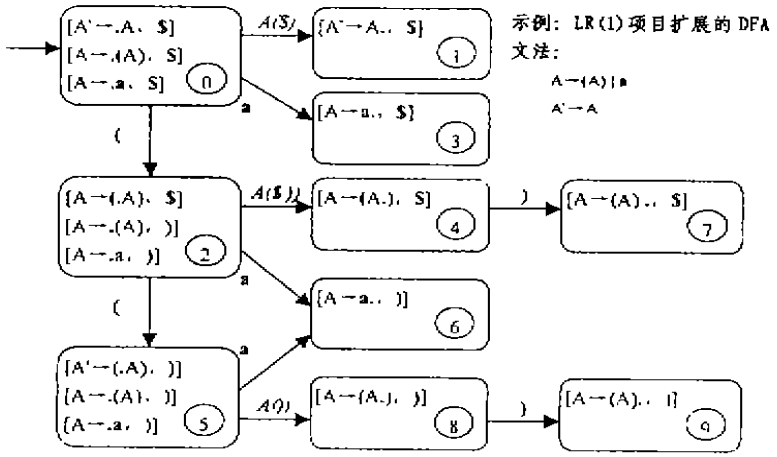
引理1　K 是一个 LR(1)文法 P 的 LR(1)项目集合的 DFA。E 是 P 中的一个非终结符，ω 是 P 的一个终结符序列，E→ω 是 P 中的一个产生式。状态 a 是 K 中的一个状态，a 存在在非终结符 E 上的状态转换（转换目标为 a'），并且包含项目[E→ω·,e]。e 是 P 中的终结符。在应用 K 的 LR(1)分析过程中：如果当前符号栈顶状态是 a，当前输入记号串是 ωeγ $，则 LR(1)算法在输入串中取出 e 之前，将应用产生式 E→ω，把 ω 规约为非终结符 E，使得当前状态转换为 a'。其中，γ 是任意终结符串。

证明：（略）。

引理1意味着，在 LR(1)分析中，当前输入串是 ωeγ $，只要当前分析栈栈顶状态满足上面的条件，在不同的几次分析中 ω 都将规约为 E。

类似地，可以讨论 E 的产生式右端包含非终结符时的情况：

引理2　K 是一个 LR(1)文法 P 的 LR(1)项目集合的 DFA。E、A 是 P 中的一个非终结符，ω 是 P 的一个终结符序列，$E \xrightarrow{+} \omega$ 是 P 中的一个多步最右推导。状态 a 是 K 中的一个状态，a 存在在非终结符 E 上的状态转换（转换目标为 a'），并且包含项目[F→α·,Eβ,k]，e 是 P 中的终结符，e∈First(βk)。在应用 K 的 LR(1)分析过程中：如果当前符号栈顶状态是 a，当前输入记号串是 ωeγ $，则 LR(1)算法在输入串中取出 e 之前，将把 ω 多步规约为非终结符 E，使得当前状态转换为 a'，并且相应的语法子树与最右推导 $F \xrightarrow{+} \omega$ 所构造的语法树相同。其中，γ 是任意终结符串，First() 表示句型或句子的先行集合。



可用归纳法证明（略）。引理2将引理1推广到了多步推导的情况。这样，在增量语法分析中，如果能够满足引理2的条件，可以直接使用以前构造的语法树的非叶节点，极大提高了语法分析的效率。根据这样的思路，依照 First(βk)，可以为每一个发生在非终结符上的状态转换附加一个允许的后缀集合。状态 a 在 E 上的状态转换的可移进条件 ξ(a,E)是一个终结符的集合，定义为：

ξ(a,E)={e|[p,e]∈a,[p,e]是闭包项，p 的左部为 E，e 是终结符}。

在增量语法分析中，如果设当前分析栈的栈顶状态为a，当前输入为语法树的节点E，其后序终结符为e，如果在E上存在由a到状态b的转换，并且e⊂ξ(a，E)，分析程序可直接将E压入值栈，将状态b压入分析栈之中，而不影响分析结果的正确性。

2.2 扩展的有冲突LR(1)项目集合的DFA

以上的讨论是针对没有冲突的DFA的情况，但现实中的语言一般都存在少量的冲突，这或许是因为语言的设计首先是从使用者的角度进行的，而不是编译器的角度。比如著名的C++语言的实现g++，含有4个规约-规约冲突。

在有冲突的情况下的LR(1)项目集合的DFA为了消除冲突，总是按照某种策略保留一个有冲突的LR(1)项目，而删除其它的。为了与YACC相配合，这里采用和YACC相同的策略：移进-规约冲突，总选择移进；规约-规约冲突，按定义的先后顺序选择。这样问题就变成了，在某一个状态，删除了某一规约项后，对DFA中状态转换的可移进条件有什么影响。

设DFAD已经构造完成，构造过程中在状态a存在冲突，删除的规约项的产生式为E→ABC，先行记号为e。在D的反图(D^{-1})中从a出发，依次通过CBA标记的状态转换可以到达的状态为b(实际中不止一个)。如果规约项没被删除，当执行此规约时，在出栈操作后b应该是位于分析栈顶的状态，并紧接着执行一个在E上的状态转移。然而，此规约项已被删除，使得b中闭包项[E→ABC，e]完全失效，是可删除的。

定义状态a在E上的状态转换的可删除条件是一个二元组的集合，定义为：

σ(a，E)={q，e|[p，e]∈a，[p，e]是可删除的闭包项，p的左部为E，p的产生式是q，e是终结符}。

为每一个非终结符上的状态转换计算其ξ(a，E)和σ(a，E)集合，即完成了"扩展的LR(1)项目集合的DFA"的构造。

为了在增量语法分析过程中应用"扩展的LR(1)项目集合的DFA"，需要为每个语法树节点设置一个属性Flag。设构造基本的LR(1)项目集合的DFA时，被删除的规约项所对应的产生式的集合为p，则Flag属性由以下规则确定：

1)所有记号的Flag属性都是0；

2)Flag=0，当构成本节点和子孙节点的产生式在不在集合P中。

3)Flag=1，当子孙节点的产生式都不在集合P中，但本节点的产生式在P中。

4)Flag>1，当子孙节点的产生式至少有一个在集合P中。

在增量语法分析中，设当前分析栈的栈顶状态为a，当前输入为语法树的节点E，相应的产生式为p，其后序终结符为e，在E上存在由a到状态b的转换。满足下面的情况之一时，分析程序可直接将E压入值栈，将状态b压入分析栈之中，而不影响分析结果的正确性。

1)Flag=0，并且e∈ξ(a，E)，

2)Flag=1，并且e∈ξ(a，E)，(p，e)∉σ(a，E)

其它情况，都不可以直接移入E。下面，给出具体的判别算法：

算法2.1 判断是否可以直接移入非终结符节点E的算法

```
BOOLEAN ISACCEPTABLE(a,N,e)
a=当前的分析栈栈顶状态
N=即将输入的语法树非叶节点
e=输入中E后面的第一个记号
BEGIN
  E=N所对应的非终结符
  P=N所对应的产生式
  IF(存在状态转换 a --E--> b)
  BEGIN
    IF(N.Flag==0)
      IF(e∈ξ(a,E))RETURN TRUE
    ELSE IF(N.Flag==1)
      IF(e∈ξ(a,E)AND(p,e)∉σ(a,E))RETURN TRUE
  END
  RETURN FALSE
END
```

2.3 扩展的LR(1)分析算法

考虑增量分析算法所面临的问题：输入是一棵完整的(或一系列)语法树，由于用户的编辑动作，语法树被记号的变动区间分成了前后三个部分：无需重新分析的语法树(子树)序列、变动区间的记号序列、需要进行增量分析的语法树(子树)序列。处理过程需要三个阶段：

1. 首先要从无需重新分析的语法树序列中恢复出变动区间前的分析栈、值栈状态。

2. 然后以正常的过程(只处理记号输入)完成对变动区间的分析。

3. 最后，扩展的分析算法处理后序的语法树序列。

对第1个阶段，由于没有记号发生了改变，重新分析的过程必然和已有的分析产生一样的结果。这样，只要简单地允许LR(1)分析算法接受非终结符(语法树非叶节点)作为输入，并直接在非终结符上进行移进操作即可。移进操作将相应的状态压入分析栈，并将输入的节点压入值栈中。这里对非总结符的移进总是成功的。第2个阶段是一个正常的对记号进行LR(1)分析的过程，第3个阶段由于受变动区间的影响，已有的分析结果不一定仍然有效，一个语法树非叶节点能否直接移进，需要由上面的ISACCEPTABLE算法来指定，这就是扩展的LR(1)分析算法；算法同时要完成Flag属性的更新，具体描述如下：

算法2.2 ELR(1)扩展的LR(1)分析算法

```
BOOLEAN ELR1(a,N,e)
a=当前的分析栈栈顶状态
N=输入队列首部的语法树节点(可以是叶或非叶节点)
e=N 后面的第一个记号
BEGIN
 first=N 的第一个记号
 IF(a 中包含了规约项[A→α. ,first])
 BEGIN
  //按照 LR(1)算法的要求,用规则 A→α 进行规约
  IF(规则 A→α 是 S'→S)//S'→S 时开始符号的产生式
       RETURN TRUE//接受状态
  ELSE
  BEGIN
   应用 A→α,在值栈中弹出|α|个节点,并构造 A,将 A 压入
   值栈中;其中,A.Flag 属性:
                 ⎧2×⊕αi.Flag+1   如果 A→α∈P
   A.Flag=       ⎨
                 ⎩2>⊕αi.Flag+0   如果 A→α∉P
   其中 P 是被删除的规约项所对应的产生式的集合,αi 表示
   α 的第 i 项,⊕表示逻辑或,并且假设 ture≠0,false=0。
   将分析栈顶部的|α|个状态弹出,此时栈顶状态 b 一定存
   在状态转换 b --A--> c,将 c 压入分析栈中。
   RETURN TRUE
  END
 END
 //尝试移进
 note=N,语法树的节点
 WHILE(note≠NULL)
 BEGIN
  first=note 的第一个记号
  E=N 所对应的非终结符
  IF(N≠$)//文件结尾标记不应被移进
  BEGIN
   follow=note 后面的第一个记号,初值为 e
   IF(ISACCEPTABLE(a,note,follow))
   BEGIN ×××××××//操作分析栈、值栈、输入位置
    将 note 从输入队列中删除,然后压入值栈;此时栈顶状
    态 a 一定有状态转换形如 a --E--> b,将 b 压入分析栈中。
    RETRUN TRUE
   END
  END
  IF(note 是非叶节点)
     note=note 的最左子节点
  ELSE
     note=NULL//叶节点(记号)无法移进将推出循环,并
     产生一个错误。
 END
 RETURN FALSE//失败,出错
END
```

算法中应注意两点:

1)必须先考虑规约的情况再考虑移进,因为移进的可能是一个非终结符,有可能使应该发生的规约被跳过。

2)ELR(1)算法无法直接移进非叶节点 N 时,并不是立刻考虑将记号移进,而是逐层地检查 N 的子节点是否能移进。

2.4 扩展的 LALR(1)分析算法

对于无冲突的 LALR(1)项目集合的 DFA,引理1和引理2仍然成立。这是因为 LALR(1)项目集的 DFA 可以从 LR(1)项目集合的 DFA 合并产生,其中的每个状态所包含的项目只比 LR(1)中的多,而不会少。

同样由于 LALR(1)与 LR(1)的分析过程相同,已有的 ξ(a,E)、σ(a,E)的构造过程依然有效。只是 σ(a,E)不再象 LR(1)分析中那么有效了。在 Flag=1的条件下,移入非终结符的条件较 LR(1)下更加宽松。由于更严格的条件要依赖于分析栈的完整信息,使得算法过于复杂,可以继续使用 ELR1 算法。

2.5 错误恢复策略

错误恢复是发生错误时对正确输入的一个合理的预期,分析程序在发生错误后根据这个预期来尝试完成余下部分的分析。由于在增量分析中,先前的分析过程已经给出了出错代码的一种可能的结果,这对于这次分析也是一个合理的预期,最简单的策略是将当前输入串中的语法树序列直接加在值栈中语法树序列的后面,构成新的序列作为语义分析和下一次的语法分析的基础。

相对传统的错误处理,在增量分析中的错误恢复不能舍弃任何记号,为了使错误处理的过程是可重现的,只能基于分析栈中的状态和输入中的单个先行来选择处理方法。最关键的一点是,错误处理不能影响增量分析的正确性。

本文采用一种和 YACC 接近的错误处理策略,主要是避免了记号的丢弃:

首先将 YACC 中 error 记号考虑为一个非终结符,所对应的语法节点的 Flag 属性标记恒为2。形式上 error 可由任意终结符、非终结符构成,即 $error \to \sum^{*}$,其中 $\sum$ 是终结符、非终结符的集合,$\sum^{*}$ 表示 $\sum$ 的幂集。

在输入串中出现 $ 之前,执行 ELR1(a,N,e)返回值为 FALSE 时:

1)报告错误。

2)开始尝试进行错误恢复。检查分析栈栈顶状态,如果有一个在 error 上的移进动作,则将 error 插入输入串。接着,分析程序执行 error 的移进动作,分析程序将转入错误恢复状态;如果分析栈顶的状态没有 error 上的移进动作,从栈分析中弹出一个状态,同时从值栈弹出一个节点,此节点保存在临时栈中。重新执行步骤2,如果堆栈已经弹空,分析过程失败。

3)转入错误恢复状态时,值栈的栈顶是 error 节点,分析程序将首先把临时栈中的节点逐一弹出,加入到 error 的子节点序列中。

4)分析器在错误恢复状态中,继续使用 ELR1()进行分析。

a)如果连续三次成功地移进了至少三个记号,分析程序回到正常的分析状态。当移进了非终结符时应考察其对应的记号的数目,而不只是非终结符的数目。

b)当 ELR1()返回错误时:i)如果进入错误恢复状态后,还没有任何成功的移进操作,当前记号应被插入位于值栈栈顶的 error 的子节点序列中,分析栈不动。ii)如果进入错误恢复状态之后,已经成功地移进了一

个或两个记号，将回到步骤2，重新开始错误恢复过程。

以上的错误处理保持了和 YACC 的一致。

实际上，error 已经被当成了一个非终结符节点，此节点可以由任意记号的序列构成。这样处理之后，对于分析步骤的第一步将无需执行错误处理的步骤，直接将包含 error 的非终结符节点移进，就可以恢复分析栈和值栈的状态。而对于后续的增量分析过程，由于包含 error 记号的语法树节点的 Flag 属性大于1会被 ELR1()过程分解，从而避免了直接的移进。这种错误处理策略还使得语法树趋于完整，有利于后面的语义分析。

3 对增量语义分析的影响

这里的语义分析指的是根据语言的规则来验证程序的合法性。传统的增量静态语义分析技术是基于结构化编辑的，其中的每个编辑动作可以抽象成对某个语法树分支的替换操作。这样每次编辑动作后，语义的冲突只发生在此分支的顶点。或者从属性文法的角度来说，只有与此分支顶点的局部语法树相关联的属性之间才存在不一致[7]。这样增量的语义分析过程就是在属性的依赖图上，以此顶点为初始节点，向外围扩张更新的过程。这一过程主要的复杂性体现在如何决定属性更新的顺序，一般地这需要事先计算出语法树中的每个节点自身的属性的依赖图。

虽然一些算法(如 Synthesizer Generator[5])可以扩充到包含多个初始节点的情况[7]，但是在非结构化的编辑过程中，一个简单的改动就可能使语法树中的大量节点发生变化，甚至完全改变语法树的结构。这时不一致的属性集合变得非常庞大，会使得属性的更新成为非常耗时的操作。除此之外，非结构化的编辑其中并不能保证语法树总是完整正确的，实际中需要有一种较为松散的策略来容忍不完整的语法。

对于 C++，Pascal 风格的编程语言的各种实现，语义检查是以符号表为中心进行的，为了提高分析效率可以将语义分析分割成两个部分：符号表的更新，基于符号表的语义检查。按照这种思路，可以将属性的依赖图进行分割。

对于没有循环的属性文法，通过与语法树相关联的符号表将整个属性依赖图分割成三部分：

1. 符号表的依赖图，此图中所有的属性都直接或间接地被符号表所依赖。

2. 依赖于符号表的属性的依赖图，此图中所有的属性都是直接或间接依赖于符号表的。这里的属性并不被及时更新。

3. 与符号表无关的依赖图。

完成语法树的更新后，只须处理1、3的情况，而2中这种依赖关系可以推迟到显示的时候再检查。这样划分可以将属性的依赖图中大部分的依赖关系截断，缩小了 PROPAGATE[7]过程搜索的空间，从而使属性更新的过程加快。

依赖于符号表的语义检查，可以通过2图的反图来进行。检查开始的范围仅限于当前编辑器中所显示的部分语法单元的属性，通过计算这些属性，就可以在编程环境中指出当前部分源代码的错误，而不用对整个属性依赖图进行计算。

结束语 文本给出了一种非结构化编辑环境下的增量 LALR(1)语法分析算法，极大地提高了语法分析的速度(一般情况下，移进的单位至少是语句，而不是记号，这将使输入的长度和算法的时间复杂度下降数十倍)。由于语法分析部分的设计采用了表驱动的方式，并尽可能地接近 LALR(1)分析算法，使得构造与 YACC 相兼容 的增量语法分析器自动构造程序成为可能。这样可以为现有的基于 YACC 的编译器用最小的代价构造出一致的增量分析程序，从而提供高效的程序开发环境。

参 考 文 献

1 Cook P, Welsh J. An Incremental LR Parse Strategy for Language-Based Editor: [Technical Report, NO. 99-24]. The University of Queensland

2 Wagner T A, Graham S L. Efficient and flexble incremental parsing. ACM Transcations on Programmung Languages and Systems, 1998, 20(5): 980～1013

3 Kenneth, Compiler Construction Principles and Practice. 1997. 170

4 徐智晨. 增量静态语义分析的一个对象模型. 软件学报, 1994, 5(9): 30～37

5 Reps T, Teitelbaum T. The Synthesizer Generator: A System for Constructing Language-Based Editors. Springer-Verlag. New York, NY, 1998

6 Kiong D. Program Analysis in Language-Based Editiors: [PhD thesis]. Department of Comprter science. The University of Queensland, 1987

7 Reps T. Ineremental Context-Dependent Analysis for Language-Based Lditors. ACM Transactions on Programming Languages and Systems, 1983, 5(3): 447～477

8 Broom B. Aspects of Interactive Program Display: [Ph D thesis]. Department of Computer Science. The University of Queensland. Sep. 1987

9 Kaiser G E, Kaplan S M, Micallef J. Multiuser, distributed language-based environment IEEE Software, 1987