语法制导的通用结构化编辑器的面向对象设计

纪锡强　钱乐秋

(复旦大学计算机科学系　上海 200433)

摘　要　通用结构化编辑器(以下简称GSE)是编辑系统研究的一个重要分支。本文介绍了如何用面向对象的方法来分析、设计和实现一个GSE的过程。本文提出的GSE融语法制导的结构化编辑器和正文编辑器于一身,克服了一般的结构化编辑操作限制太多、而正文编辑又缺乏语法制导的弱点,真正达到了两者的和谐统一,而且用面向对象方法开发本系统,使其具有较强的可重用性和易维护性。

关键词　语法制导,结构化编辑器,面向对象,产生式。

OBJECT-ORIENTED DESIGNING IN THE GENERAL SYNTAX DIRECTED STRUCTURAL EDITOR

JI XIQIANG　QIAN LEQIU

(*Department of Computer Science, Fudan University, Shanghai* 200433)

Abstract　General Structural Editor (GSE) is one of the most important research aspects of editor system. This paper tries to describe the process of analysising, designing and implementing a GSE using the Object-Oriented developing method. The GSE presented here integrates the features of both structural editor and text editor. It overcomes the limitations of common structural editor and the weakness of text editor which is lack of syntax direction. Since it is developed with OO method, there are many advantages over those editors which are developed using other methods such as waterfall model.

Keywords　Syntax directed, structural editor, object-oriented, production.

一、引　言

一个程序并不是简单的一系列字符的组合，而是由一系列语法结构单位有机地组成的

本文1996年2月6日收到。本课题得到国家八五攻关项目"青鸟II型系统(JB2)的研制开发"的资助。纪锡强,硕士生,钱乐秋,教授,现主要从事软件工程领域的研究。

一个整体。因此，一个程序文件其实就是一棵语法树，而语法树上的一个语法结点，就是相对于该程序语言的一个语法结构单位。若能有一个语法制导的编辑器来支持程序设计过程，则将大大减少程序的语法错误率。其实早在60年代末期，对结构化编辑系统的研究就已取得了一系列重要的进展。例如，国外有Teitelbum, T.开发的Cornell程序合成器[1]；Robert A. B.等开发的Pan编辑系统[2]等。在国内，复旦大学、中科院软件所等也陆续开发了一系列结构化编辑器(以下简称SE)系统，可这些SE系统都只针对某一特定语言(如Cornell是针对Pascal语言等)，有很大的局限性。人们不得不针对不同的语言开发不同的SE以适应不同的语言编辑要求，但这势必会大大限制了所开发出的SE的推广和适用性，而且不利于已开发出的软件构件的重用。本文提出的GSE具有很强的通用性，表现在用户只要输入合乎一定规范的类BNF文法和所要编辑的文本名，GSE就能提供基于多窗口的编辑界面，让用户在语法制导下编辑符合该文法的正确文本。因此，GSE既可以引导编辑各种程序设计语言，还可以引导编辑规格说明、SA/SD结构方法产生的小说明、信函以及程序框架等任何可由BNF文法描述的结构序列。复旦大学软件工程实验室在七五、八五期间参与研制的青鸟系统就是使用GSE来编辑有关的文档和程序的。以往的SE系统大都只能在语法的制导下严格地进行结构化编辑操作，整个文本(包括短语、表达式等)都是通过一步步的语法制导被动地完成的，用户不能通过正文编辑来主动地输入文本，即使有也是通过结构化编辑和正文编辑的不同状态切换来实现的，这样就会导致因语法制导过深而引起对语法已足够熟悉的用户的厌烦，而且也会导致不够直观和编辑效率的低下。而我们的GSE是结构化编辑和正文编辑的混合体，用户可以在当前状态下同时进行结构化编辑和正文编辑。其中产生式模板可由结构化编辑命令产生，而短语和表达式等则可通过正文编辑一次输入一个字符。由于模板是预先定义的，不可能出错，而当用户输入短语时所产生的正文编辑错误，可在编辑光标离开这个短语时去调用语法分析器而被检测到，因此就保证了所编辑的文本不管是结构化编辑产生的或是正文编辑产生的在语法上都是正确的。同时，达到了结构化编辑和正文编辑的和谐统一。

二、系统总体结构和流程

GSE的系统总体结构由语法规则读入模块、文本的语法分析模块、文本的格式显示模块以及文本的语法制导编辑模块和编辑总控模块等组成，可表示成如图2.1所示。

GSE的总体工作流程是：语法制导的结构化编辑器将所有要编辑的正文看成是由一系列语法结构单位组成的有机整体。当用户编辑一个文本时，首先要调入一个所要编辑文本的语法规则，GSE读入该语法规则，变成GSE的内部语法表示，以后编辑各文本缺省使用

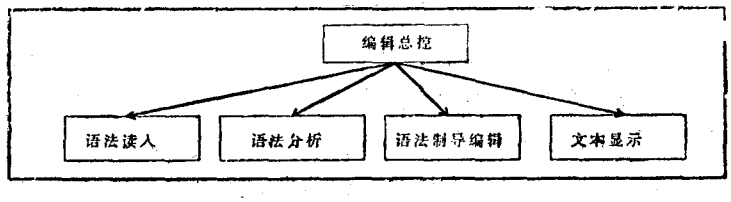

图2.1 系统总体结构

的都是该语法规则，除非用户又重新调入新的语法规则。接着，用户便可调入一个已存在的文本或创建一个新的文本。若是已存在的文本，则 GSE 先用相应的语法规则来对该文本进行语法分析，将其转换成为 GSE 的内部格式。最后，GSE 将文本按格式显示在屏幕上，用户即可通过各种编辑操作与通用编辑器进行交互编辑。若是结构化编辑，用户可用特定的光标移动键将编辑光标移动到所要编辑的地方，然后按下某个编辑命令键，GSE 将所有可适用的语法结构显示于屏幕上供用户选择。GSE 用用户选择的语法结构来替换光标所在处的语法结构，并调用格式显示模块将编辑修改后的文本重新显示在屏幕上。若是短语的正文编辑，则当用户按下第一个字符键时，系统便知道开始输入该模板处的短语，一旦回送该字符至屏幕，则短语模板消失，短语的右边上下文自动地调整位置以容纳键入的字符。当短语输入结束后光标从该短语离开时，系统就会激发语法分析器来分析该短语，有错则显示出错信息，然后可通过插删字符操作重新编辑短语，直至无语法错误，才允许将光标移至下一个模板。

三、GSE 的面向对象分析与设计

面向对象的开发方法从其产生、发展至成熟不过短短二十年的时间，但却已渗透到计算机技术的各个方面，成为当前计算机领域的研究热点。当前流行的面向对象开发方法主要有三种：文献[3]中 Rumbaugh,J. 等提出的 OMT，[4]中 Peter Coad 和 Edward Yourdon 的 OOA 和[5]中 G. Booch 的 OODA。三种方法各有侧重点，也各有优劣。我们在开发 GSE 的过程中使用了[3]中提出的 OMT 开发方法。

3.1 GSE 的对象模型

从 GSE 的系统分析出发，通过识别出正确的对象和标识出对象类之间的正确关联，可得出 GSE 对象图(简化)如下：

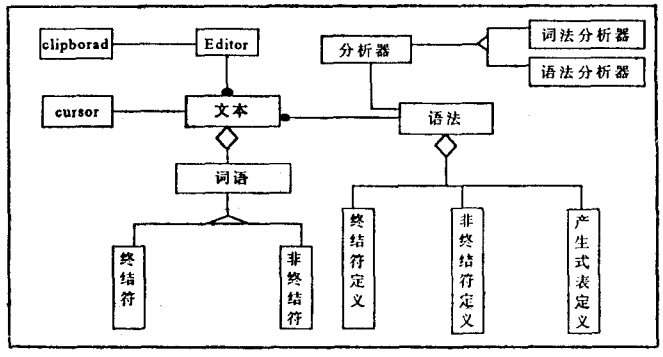

图 3.1

在图中，clipboard 对象是子树缓冲区，用于结构子树的删除、存储、粘贴，是文本修改编辑功能的基础。Editor 对象控制多个文本，并指示当前文本。它用于实现多个文本的同时编辑，以MDI(Multiple Document Interface)的形式提供。cursor 对象是文本的结构编辑光标，用于指示当前可进行编辑操作的语法结构单位。文本对象是 GSE 的核心对象，它是由若干个词语对象组成的。一个文本对象都有一个语法对象与其相对应，而一个语法对

象又对应于一个语法分析器和一个词法分析器，它包括终结符定义、非终结符表定义以及产生式表定义。

3.2 GSE 的动态模型

通过考察图 3.1 中标识出的对象，根据 GSE 的典型交互过程可以识别出系统与外界之间的事件。若把每一事件同对象模型中的类联系起来，找出事件发送者和接收者，然后根据其对事件的发送和接收，就可构造出每个对象类的状态图。例如，类 Editor 的状态图如下：(限于篇幅，其余的状态图略)

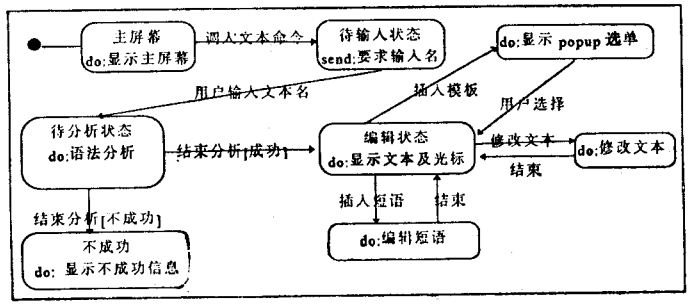

图 3.2 类 Editor 的状态图

3.3 GSE 的功能模型

通过标识系统与外界之间的输入/输出关系，根据输入/输出值可构造出 DFD 图，并逐步求精，直至不能继续分解为止。根据问题陈述和工作流程描述，我们构造 GSE 的顶层数据流图如图 3.3 所示。(其余分层数据流图略)。

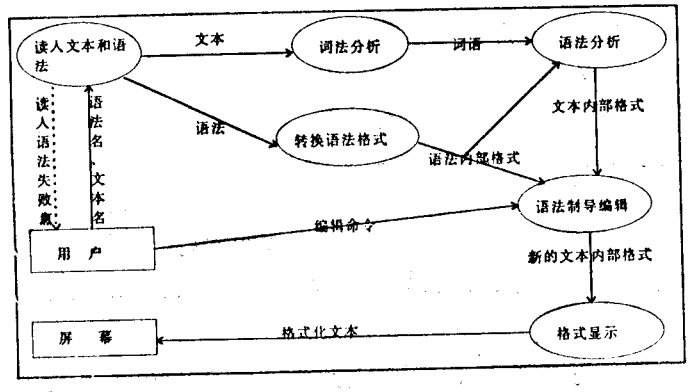

图 3.3 顶层数据流图

3.4 GSE 的对象设计

(1) "语法"对象的设计 如何表示各式各样的语法结构是实现 GSE 通用性的关键。这里我们选用上下文无关文法来描述文本的各种不同语法结构。我们知道，一个上下文无关文法由四部分组成：一组终结符、一组非终结符、一个开始符号和一组产生式规则，且由于短语输入采取正文编辑的方法，因此在文法规则的定义中须将短语当作系统内定的终结符。具体的关于 GSE 结构化正文语法的语法规则定义如附录所示。例 1：

⟨S⟩::=⟨IFS⟩@

⟨S⟩::=⟨IFS⟩⟨S⟩@

⟨IFS⟩::=IF %exp% THEN ⟨S⟩ ELSE ⟨S⟩ @

⟨S⟩::=do action; @

即是一个合乎GSE语法规则的语法定义。我们可看到，一个上下文无关文法实际上是由三表一符组成：终结符表、非终结符表、产生式表和开始符号。而产生式本身也是表，一条产生式就是一个由终结符和非终结符组成的有序表。其特殊性在于它要求表中的第一个符号即产生式左部为非终结符。因此，从中就可抽象出一个对象：表。加入此对象并改写前面对象图中的语法部分如图 3.4 所示。另外，我们还注意到，开始符号实际上就是一个非终结符，所以若在非终结符表中始终将开始符号置于表头，则无需额外定义一个开始符号。而且从问题陈述可知，语法是语法制导编辑的依据，编辑过程中对语法的访问全都是读访问，没有写操作，所以不必担心语法中具有位置关系约束的表会被打乱。至此，我们已基本构造出了"语法"对象框架。以下对各对象用类C++语言来描述。

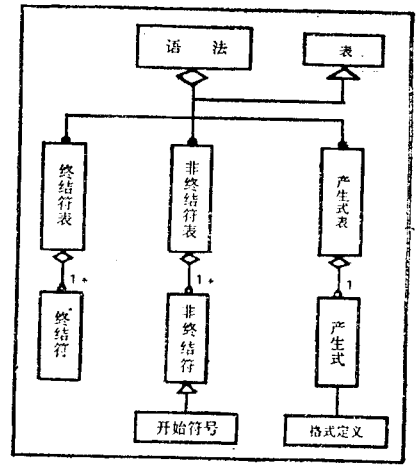

图 3.4

由于将终结符和非终结符都看成是词语，因此可从中先抽象出一个 Token 对象来：

```
class Token {char*name; //词语名
             BOOL Terminal; //终结符标志
             int val; //该词语在终结符表或非终结符表中的序号
             …//其它的数据成员和成员函数}
```

有了 Token 对象类，我们就可以将一个语法在内部表示成由终结符表(TermT)、非终结符表(UnTermT)、产生式表(ProducerT)和格式定义表(FormatT)组成的对象类，而且从 TermT 和 UnTermT 中还可以抽象出一个词语类 TokenT 来，而 TokenT 类又是更抽象的表类 List 的子类，在从更高的抽象层次上，Token 类和 List 类又是抽象类 Object 的子类。而产生式本身又是一个词语表 TokenT，因此在内部格式中各个类之间的关系可表示如下：

```
class Token: Object; class List: Object; class TokenT: List;
class ProducerT: List; class FormatT: List; class Producer: TokenT;
```

有了上述定义后，语法对象可表示成如下：

```
class Syntax {TokenT *pTermT; //终结符表
              TokenT *pUnTermT; //非终结符表
              ProducerT *pProducerT;//产生式表
              FormatT *pFormatT;//格式定义表
              …//其它的数据成员和成员函数}
```

接下来看看，语法对象必须提供的操作：FindProducer (struct), FindItems(struct)。

FindProducer 发生在扩展语法结构命令时，此时应根据光标所在的未扩展语法结构来寻找可以扩展该语法结构的产生式，并由用户来选择进行扩展，也就是根据左部非终结符查找匹配的产生式，即：ProducerT Syntax:: FindProducers(Token *pNonTerm)。这个操作应由通过 Syntax 中的 ProducerT 来完成，因此在 ProducerT 中加入此操作：ProducerT ProducerT ::FindProducers (Token *pNonTerm)。

FindItems 发生在插入表项命令时，光标停留在语法结构 S 上，此时应判断 S 是不是一个表结构，如果是，就应立即在 Syntax 中查找可构成这种表结构的表项并插入到 S 中。表结构是一种特殊的语法结构，凡是带有左右递归的产生式都将构成表结构。例 1 中的 S 就是一个表结构，它有表项 IFS。亦即，形如 P⇒……P 的产生式就是表结构，因此查找表项就是查找形如 P⇒……P 的产生式。显然，该操作也应由 Syntax 中 ProducerT 来完成：

ProducerT Syntax::FindItems (Token *pNonTerm)

ProducerT ProducerT::FindItems (Token *pNonTerm)

(2) "文本"对象的设计　在 GSE 中，文本的内部表示是一棵逻辑语法树，树上的每个结点都带有各自的结构信息，文本的显示、打印以及各种编辑命令的执行都是在该语法树上进行的。语法中的每个产生式、终结符和非终结符都可以看成是树上的一个结点。产生式所对应的结点可以看作为树上的一棵子树的根，它相当于产生式名，用以标识结构成分。指针则将一个结点中的某些叶结点与另一个结点的根节点连接起来，因此每个结点可表示成如下：

结构信息	儿女指针	父母指针	兄弟指针

从上分析，我们又可得到两个对象类：语法树(SyntaxTree)、语法结点(SyntaxNode)，而其中的语法树对象类即是文本对象的内部表示，它是由一系列语法结点对象组成的。以文本：

if aa then ⟨S⟩

do action;

为例，看看文本对象提供的操作：

文本对象支持用户操纵文本建立、修改、显示等各种结构编辑操作，以及光标移动、文件存取等辅助操作。当文本和编辑光标显示在屏幕上时，用户可执行以下一些编辑操作：

● 文本建立。文本是通过自顶向下在现有的产生式模板中插入新的模板和短语来建立和扩展的，它由非终结符的扩展与反扩展以及短语编辑等功能来完成。

● 扩展非终结符。若用户选择产生式 S⇒do action; 对文本中的非终结符进行扩展，即当光标停留在语法树的叶结点上时，用户键入扩展命令并选择上述产生式后，文本应变成：

if aa then do action;

do action;

● 反扩展非终结符。它是扩展非终结符的逆操作。

● 插入表项。若在文本中语句 do action; 前插入一个表项，此时文本应变为：

if aa then ⟨S⟩

⟨IFS⟩

do action;

● 删除表项。它是插入表项的逆操作。

● 编辑短语操作。短语是由用户在当前光标处输入的。因此，为了保证短语的语法准确性，当光标从短语离开时，会触发相应的分析器来分析该短语，直至无语法错误方允许用户将光标移开。

● 光标移动。由于编辑操作都在语法树上进行的，因而可控制光标按顺序有层次地遍历抽象语法树结点，以支持用户寻找编辑目标。提供的编辑操作有：移至树根结点；移至同层结构的前(后)一个结点；移至上(下)层结构的头一个结点；顺序移至下一个模板、短语以及在短语编辑时的左右移一个字符。如光标由结点 'aa' 移至结点 '⟨S⟩'。

● 程序显示(美观格式化输出)。每当一个编辑操作完成后，GSE 重新计算文本的输出格式并刷新文本的显示，至于文本的输出格式可定义在语法规则中。

● 文本修改。GSE 提供一组灵活、方便的修改操作，包括：1)删除：删去当前光标所在处的语法结点，重新出现原来替换掉的位标，并将其存入子树缓冲区；2)拷贝：将当前光标所在处的语法结点存入子树缓冲区；3)粘贴：将子树缓冲区中的语法结点复制到当前光标处。

(3) "通用语法分析器"对象的设计　通用语法分析器(Gparser)对象对文本进行语法分析，将其变成内部语法树供用户继续编辑修改。Gparser 的工作过程是：GSE 以文本和语法为参数发出开始语法分析消息，Gparser 收到这个消息后，立即对文本进行语法分析，并将分析得到的语法树传回给 GSE。GSE 使用递归下降分析方法对输入的文本进行语法分析并将其转变成内部格式。Gparser 与传统的递归下降分析所不同的是：Gparser 所分析的文本中可能含有未展开的非终结符，这样它对应的语法分析表和总控程序都应该有所不同，因此我们对非终结符进行了特殊的处理。以下用类 C++ 语言对 Gparser 中的语法分析操作及其部分算法进行描述。

● 预测分析总控程序。预测分析程序的总控程序可形式化地表示成如下：

```
Gparser :: parser( ) {stac k. push (ENDTOKEN); //把结束符 ENDTOKEN "#"推进栈
          token=GetNextToken( );//把第一个输入符号读进 token
          FINISH=FALSE;
          while (not FINISH) {
            topToken=stack. pop( );//把栈顶符号推出栈
            if (topToken 是终结符){
              if (topToken==token) token=GetNextToken( );//同一个终结符时
              else error( );}
            else{
              if (topToken==token) token=GetNextToken( );//同一个非终结符时
                    else if (token 是非终结符) error( );
                          else {index=M[topToken. val, token. val];
                                X→X1X2…Xk 是序号为 index 的产生式;
                                把 X→X1X2…Xk 倒序推进栈中;}}}}
```

● 构造分析表操作及算法。总控程序中的分析表 M[a, b]是一个二维矩阵，其中 a 是某

个非终结符在非终结符表中的序号，b 是该非终结符所面临的终结符在终结符表中的序号，而元素 M[a, b]中存放的是关于该非终结符的一个产生式序号。对于给定的文法 G，为了构造其分析表 M[a, b]，必须预先定义 FIRST 和 FOLLOW 集。以下是 GSE 中对一个词语 T，产生 FIRST 集合的过程：

```
FIRST Gparser :: GetFirst (Token T)
{if (T是终结符) FIRST. add (T);
    ELSE if (T是非终结符){
        在产生式表中寻找左部符号为 T 的产生式集合 PS;
        for (PS 中的每个产生式)
        {for (产生式右部的每一个符号 S)
            {FST=GetFirst (S);
            FIRST. add (FST 中所有非 EMPTYTOKEN 的 token);
            if (ε∈FST)
            {if(产生式右部的最后一个符号) FIRST. add (EMPTYTOKEN)}
            else 跳出该循环}}}
```

现在就能够对文法 G 的任何符号串 $\alpha=X_1X_2\cdots X_n$ 构造集合 FIRST(α)如下：

```
FIRST Gparser :: GetFirst (TokenT ST)
{for (ST 中的每个 token){
    FIRST. add(First(token)中除去 EMPTYTOKEN 外的所有符号);
    if (ε不属于 First (token)结束;}}
```

对于文法 G 的每个非终结符 A 构造 FOLLOW(A)如下：

```
FOLLOW Gparser:: GetFollow (Token A)
{ if (A==STARTTOKEN) FOLLOW. add (ENDTOKEN);
  for(产生式表中的每个产生式 P){
    if (P 为 αAβ) FOLLOW. add (First(β)中除去 EMPTYTOKEN 外的所有符号);
    if (ε∈First (β) 或 P 为 αA) FOLLOW. add (Follow (A)); }}
```

在对文法 G 的每个非终结符 Z 及其任意候选 α 构造出 FIRST 和 FOLLOW 集后，即可用它们构造 G 的分析表如下：

```
AnalTab *Gparser:: GetAnalTab( )
for(产生式表中的每个产生式 A→α)
{
for (FIRST(α)的每个终结符 a)
    M[A. val, a. val]=index(A→α);
if (ε∈FIRST(α))
    for (FOLLOW(A)的每个终结符 b)
    M[A. val, b. val]=index(A→α);
}
将所有无定义的 M[x, y]置上出错标志;
```

四、总　　结

本文介绍了如何用面向对象的方法来开发 GSE 的过程。由于 OO 方法从问题领域的实体对象入手并以此为基础进行软件开发，因此它比传统的开发方法有更稳定的基础。不

论用户需求如何变化，问题领域的实体对象总是客观存在的，不会因用户而改变。正由于这些OO的内在特征，使得用面向对象方法开发出来的GSE系统具有很好的可重用性和易维护性，很容易对其进行修改和扩充。如文中通用语法分析器对象是用递归下降分析方法对文本进行语法分析，我们也可容易地将其改为用LR(k)文法来分析，而不影响该对象的外在行为。当然，本系统还只是介于原型系统和用户系统之间的半成品，在理论和实践上还有待进一步提高。如在正文编辑时可增加部分静态语义检查功能；在系统定义的符号表基础上检测诸如：表达式中下标变量是否越界；变量使用前是否有初值；变量是否只定义而不使用等等。本系统是国家"八五"攻关项目"青鸟系统"JB2的一个配套子系统。

附　录

GSE的结构化正文语法的语法规则：

语法::=产生式表%%
　　输出格式表 %%

产生式表::=产生式 @|产生式 @ 产生式表

产生式::=开始符 分隔符 符号表

开始符::=〈非终结符〉

分隔符::=::=

符号表::=终结符|〈非终结符〉|终结符 符号表|〈非终结符〉符号表

输出格式表::=ε|格式信息 输出格式表

格式信息::=(数字 数字 数字)(其中第一个数字为产生式序号，第二个数字为产生式中符号序号，第三个数字为缩进字符个数)

参 考 文 献

[1] Teitelbum, T., "The Cornell Program Synthesiser: A Syntax Directed Programming Enviroment", Communication, ACM, 24(9), pp. 563～573, 1981.

[2] Robert A. Ballance, Susan L. Graham etc., "The Pan Language-Based Editing System for Integrated Development Enviroment", ACM, 1990.

[3] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenden, W., "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[4] Peter Coad & Edward Yourdon, Object-Oriented Analysis, Yourdon Press Computing Series, Prentice Hall, 1991.

[5] G. Booch, Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company, 1991.

[6] 陈火旺、钱家骅、孙永强，编译原理，国防工业出版社(北京)，1984.