第七章 遍历语法树及语义分析

上一章我们学习了如何创建语法树。语法树是语法分析的成果。代表了输入字符串的结构信息和符号信息。所以如何使用利用语法树就是一个重要的问题。语法树的结构与语法规则有关语法规则是决定着输入字符串的写法，所以语法树的结构与输入字符串的结构也是相关的。我们做语法分析的工作可能基于两方面的情况。第一：我们在很多时候做语法分析工作并不是要像编译器那样接下来要生成代码（生成可执行文件）。我们只想获得字符串中包含的信息。如解析 XML 文档，这时不需要进行语义分析生成代码。但语法树的结构并不易于使用，我们需要更好的存储结构来更好更直观地表示语法树中的信息，我们可以定义相关的类模型来直观的体现结构和其中的信息。

第二：像编译器一样或类似编译器地进行语义分析并生成代码，或生成另外一种格式的语言。总之这些都需要我们遍历语法树，我们既可以自己编写程序遍历语法树，也可以利用 ANTLR 中的 Tree Parser 分析器来遍历语法树。下面我们对于第一种情况自己编写遍历语法树。

7.1 编写程序遍历语法树

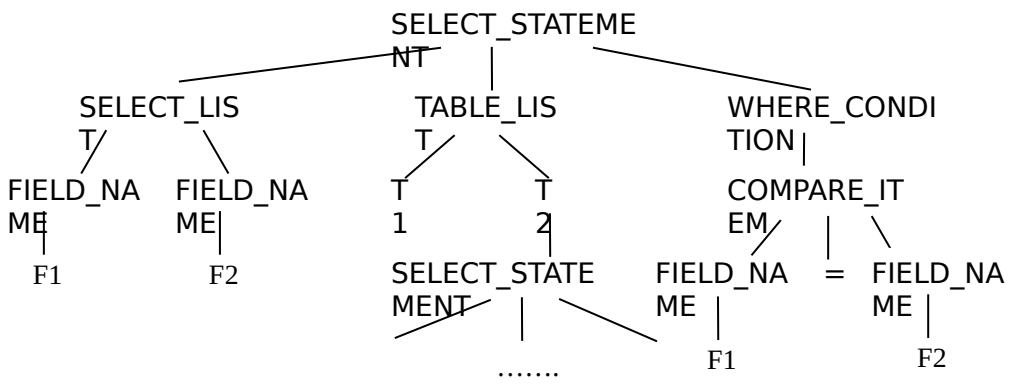
我们用一个简化的 SELECT 语句的例子，编写程序遍历 SELECT 语句的语法树并将获得的信息存储到一个直观的对象结构中。先看下面的文法：

```
grammar Select;
options { language=CSharp; output=AST;}
tokens {
  SELECT_STATEMENT;
  SELECT_LIST;
  TABLE_LIST;
  WHERE_CONDITION;
  FIELD_NAME;
  COMPARE_ITEM;
}
selectStatement : selectClause fromClause whereClause?
        -> ^(SELECT_STATEMENT selectClause fromClause whereClause?);
selectClause : 'SELECT' ('*' -> ^(SELECT_LIST '*')
          | fieldName (',' fieldName)* -> ^(SELECT_LIST fieldName+)
```

```
            );
    fromClause : 'FROM' tableSource (',' tableSource)*
        -> ^(TABLE_LIST tableSource+);
    whereClause : 'WHERE' searchCondition -> ^(WHERE_CONDITION searchCondition);
    searchCondition     : searchItem ('AND' searchItem)*;
    searchItem : expression ((p='=' | p='>' | p='<' | p='<>') expression
                -> ^(COMPARE_ITEM expression $p expression)
               | 'IS' 'NOT'? 'NULL' );
    expression : fieldName | STRING | INT;
    tableSource : tableName | '(' selectStatement ')' 'AS' tableName
                -> ^(tableName selectStatement);
    fieldName : Identifier ('.' Identifier)*
        -> ^(FIELD_NAME Identifier ('.' Identifier)*);
    tableName : Identifier;
    Identifier : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
    STRING : '\'' (~'\'')* '\'';
    INT : '0'..'9'+;
    WS :  ( ' ' | '\t' | '\r' | '\n' )+ { Skip(); } ;
```

这个文法简化了 SELECT 语句的内容，把 SELECT 语句主要分三个部分，一：SELECT 列表（SelectList），认为选择内容只是字段信息,所以在 selectClause 规则中直接使用了 fieldName 字段名做为查询项。二：数据源列表（TableSourceList）数据源有两种，一种是单独的数据表，一种是子查询。三：查询条件（WhereCondition），查询条件也简化成只有 AND 连接的比较表达式。为了使语法树更有意义更容易处理加入了多个虚拟节点 SELECT_STATEMENT 代表整个 SELECT 语句，SELECT_LIST 代表所有查询项的根节点，TABLE_LIST 代表数据源列表的根节点，WHERE_CONDITION 代表查询条件的列表，其中如果查询条件是两个字段的相等比较，则添加 COMPARE_ITEM 节点来标识此条件是表关联的关联条件。expression 表达式规则中有整型、字符型和表字段，如果是表字段则添加 FIELD_NAME 作为标识。另外子查询生成语法树时将子查询的别名作为根节点，这样和其它表会有相同的结构

我们来编写程序遍历语法树获得 SELECT 的信息，这个过程事实上是进行语义分析，



语义分析是语法分析的下一个阶段。这里我们要了解 SELECT 语句查询哪些表的哪些字段以及　表之间的关联情况。首先我们要定义一个模型能够以最合理的方式存放遍历的结果。所以定义了如下的 Select、Table、Relation 类：

```
public class Select
{
  public Dictionary<string, Table> SelectTables
              = new Dictionary<string, Table>();

  public Dictionary<string, Relation> Relations
              = new Dictionary<string, Relation>();

}
public class Table
{
  public List<string> SelectList = new List<string>();
  public string Name;
  public Select SubSelect;
}
public class Relation
{
  public Table LeftTable;
  public Table RightTable;
  public List<string> RelationCondition = new List<string>();
}
```

Select 类代表一个 SELECT 语句。Table 类代表数据表，Table 类中包含了查询字段列表。Relation 类代表之间的关联。一个 SELECT 语句中包括查询表和表关联的信息。表关联包含关联的两个表 LeftTable 属性和 RightTable 属性（本示例中只有二元关联）和关联的条件 RelationCondition 列表。

Dictionary<>泛型类名值列表与 java 中的 Map 属同种数据结构，其它语言请选择自已的方法来实现。下面我们看一下遍历语法树的代码：

```
private void button1_Click(object sender, EventArgs e)
{
  SelectLexer lexer = new SelectLexer(new ANTLRFileStream("Select.sql"));
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  SelectParser parser = new SelectParser(tokens);
  SelectParser.selectStatement_return r = parser.selectStatement();
  System.Console.WriteLine(((BaseTree)r.Tree).ToStringTree());
  printTree((BaseTree)r.Tree);
```

```
  Select select = WalkSelect((BaseTree)r.Tree);
 }
```

这一段是运行代码，分析器生成语法树之后调用 WalkSelect 函数来遍历语法树。在方法之外定义了一个 Select 类的对象，后面讲述的方法会逐步添加信息到 select 对象中。

```
 private Select WalkSelect(BaseTree root)
 {
  Select select = new Select();
  Dictionary<string, List<string>> SelectListDic
                    = new Dictionary<string, List<string>>();
  for (int i = 0; i < root.ChildCount; i++)
  {
   BaseTree currTree = (BaseTree)root.GetChild(i);
   if (currTree.Type == SelectParser.SELECT_LIST)
   {
    GetSelectList(currTree, SelectListDic);
   }
   else if (currTree.Type == SelectParser.TABLE_LIST)
   {
    GetTableList(currTree, select);
    foreach(KeyValuePair<string, List<string>> keyValue in SelectListDic)
    {
select.SelectTables[keyValue.Key].SelectList.AddRange(keyValue.Value);
    }
   }
   else if (currTree.Type == SelectParser.WHERE_CONDITION)
   {
    GetRelation(currTree, select);
   }
  }
  return select;
 }
```

WalkSelect 方法是遍历语法树启始方法，方法中使用一个循环寻找

SELECT_LIST、TABLE_LIST 和 WHERE_CONDITION 三个 SELECT 语句中主要的子树，分别调用 GetSelectList、GetTableList、GetRelation 来处理。在寻找子树时可以使用树节点的 Type 属性值，与分析器中定义的与词法规则符号和语法树虚拟节点的常量进行比较。

GetTableList 方法执行之后对于一个 SelectListDic 字典进行了循环。这段代码我们在后面才可以全面了解。由于 SELECT 语句查询项在前面 FROM 查询表前面，所以在分析查询项（字段）时，并不知道这些字段是哪个表的字段只能暂时保存在 SelectListDic 字典中，然后在分析了 FROM 子句之后再将 SelectListDic 字典中的字段分配到 select 对象的 SelectList 属性中。下面看一下 GetSelectList 方法：

```
private void GetSelectList(BaseTree selectList,
                    Dictionary<string, List<string>> SelectListDic)
{
 for (int i = 0; i < selectList.ChildCount; i++)
 {
  BaseTree selectItem = (BaseTree)selectList.GetChild(i);
  if (selectItem.Type == SelectParser.FIELD_NAME)
  {
        string fieldName = selectItem.GetChild(selectItem.ChildCount -
1).Text;
    string tableName = selectItem.GetChild(0).Text;
    if (!SelectListDic.ContainsKey(tableName))
     SelectListDic.Add(tableName, new List<string>());
    SelectListDic[tableName].Add(fieldName);
  }
 }
}
```

如前所述在 GetSelectList 方法之前作为全局对象定义了一个 SelectListDic 字典，它是表名和其字段名相对应的列表<string, List<string>>。方法中循环遍历 selectList 子树搜索查询项，是以“表名.字段名”形式的查询项的表名和字段名分离（这里为了示例简单我们忽略单表查询所以字段名前都要有表名前缀）。这些表名和字段名按对应关系保存在 SelectListDic 字典中。文法生成语法树时已经确定查询项一定是以 FIELD_NAME 为根节点的子树。

```
private void GetTableList(BaseTree tableList, Select select)
{
 for (int i = 0; i < tableList.ChildCount; i++)
```

```
    {
      BaseTree tableSource = (BaseTree)tableList.GetChild(i);
      Table table = new Table();
      string tableName = tableSource.Text;
      table.Name = tableName;
      select.SelectTables.Add(tableName, table);
      if (tableSource.ChildCount > 0)
      {
        Select subSelect = WalkSelect(tableSource.GetChild(0) as BaseTree);
        table.SubSelect = subSelect;
      }
    }
  }
```

GetTableList 方法用来获得数据表信息。TABLE_LIST 子树的第一个节点都是一个数据表，其中可能会出现子查询的数据表，在语法树中如果是子查询 FIELD_NAME 会有子树，子树为子查询语法树的根节点 SELECT_STATEMENT。这时递归调用 WalkSelect 方法来继续遍历子查询语法树。遍历子查询后返回的 select 对象赋给 table 对象的 SubSelect 属性。

```
  private void GetRelation(BaseTree searchList, Select select)
  {
    for (int i = 0; i < searchList.ChildCount; i++)
    {
      BaseTree searchItem = (BaseTree)searchList.GetChild(i);
      if (searchItem.Type == SelectParser.COMPARE_ITEM)
      {
        BaseTree leftTree = (BaseTree)searchItem.GetChild(0);
        BaseTree opterTree = (BaseTree)searchItem.GetChild(1);
        BaseTree rightTree = (BaseTree)searchItem.GetChild(2);
        if (leftTree.Type == SelectParser.FIELD_NAME
                && rightTree.Type == SelectParser.FIELD_NAME)
        {
          string leftTableName = ((BaseTree)leftTree.GetChild(0)).Text;
          string leftFieldName = ((BaseTree)leftTree.GetChild(2)).Text;
```

```
            string rightTableName = ((BaseTree)rightTree.GetChild(0)).Text;
            string rightFieldName = ((BaseTree)rightTree.GetChild(2)).Text;
                        string key = String.Format("{0}-{1}", leftTableName,
rightTableName);
            string conditionStr = String.Format("{0}.{1} {2} {3}.{4}",
              leftTableName, leftFieldName, opterTree.Text,
              rightTableName, rightFieldName);
            if (!select.Relations.ContainsKey(key))
             select.Relations.Add(key, new Relation());
            Relation relation = select.Relations[key];
            relation.LeftTable = select.SelectTables[leftTableName];
            relation.RightTable = select.SelectTables[rightTableName];
            select.Relations[key].RelationCondition.Add(conditionStr);
          }
        }
      }
    }
```

GetRelation 方法用来获得表之间的关联信息。本示例的文法只接受相等比较和 IS NULL 的条件语句，GetRelation 方法寻找“表名 1.字段 1 = 表名 2.字段”形式的条件并将其收集归纳为表之间的二元关联，并生成 Relation 对象。“表 1.字段 1 = 表 2.字段 2 AND 表 1.字段 3 = 表 2.字段 4”会被生成一个表 1 与表 2 的关联 Relation 对象。为了引用方便表名和字段名组成一个 key 字符串作为键值对应地存放到 Select.Relations 字典中。

运行示例输入：

```
SELECT Users.ID, Users.Name, Order.NO, Order.UserID,
Operators.OperatorName
FROM Users, Order, (SELECT * FROM Operators WHERE IsLive = 1) AS
Operators
WHERE Users.ID = Order.UserID AND Users.Name = Order.UserName
    AND Order.OperatorID = Operators.OperatorID AND Users.Name = '张'
```

示例生成的 Select 在内存中的情况如下图：

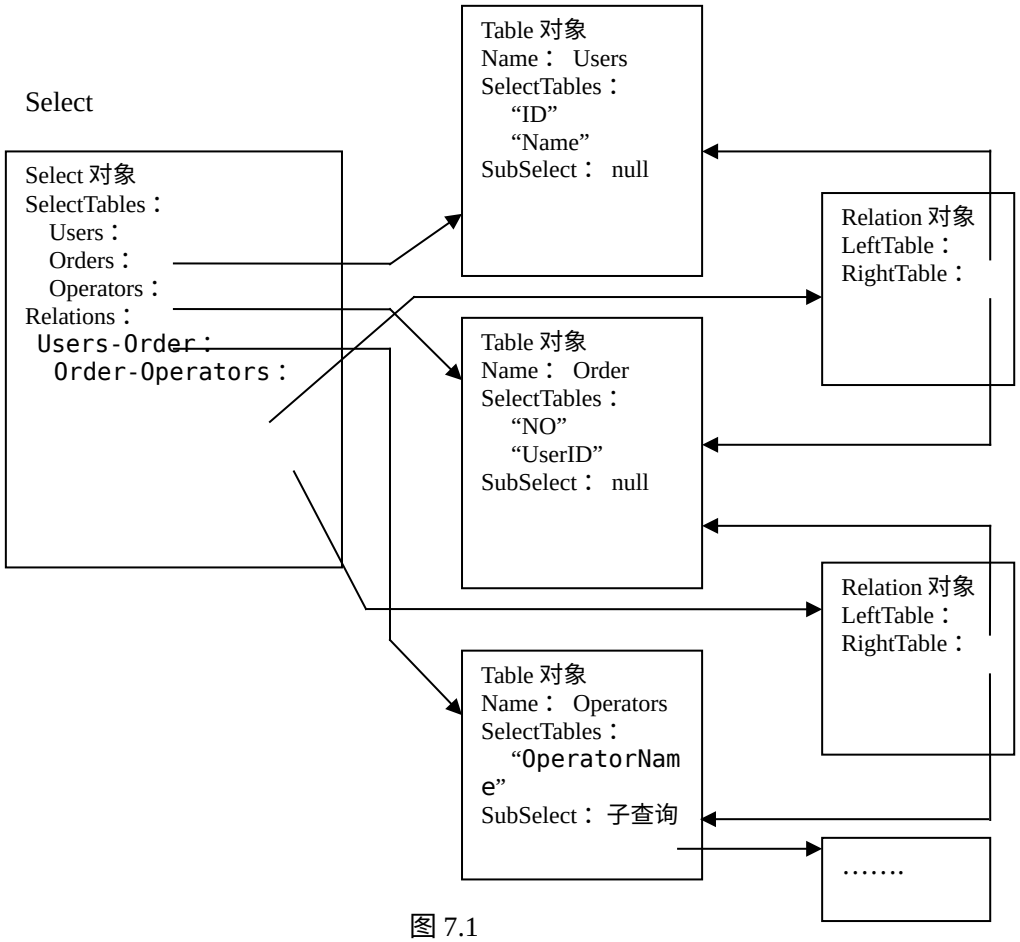


图 7.1

生成了图 7.1 所示的结构之后，一个 SELECT 语句的结构和语义我们已经很好的掌握了。有这样结构的对象再去做一些进一步的工作就容易多了。

7.2 利用 Tree Parser 遍历语法树

ANTLR 中提供的 Tree Parser(语法树分析器)是以语法树为输入对语法树进行遍历的功能模块。之所以叫 Tree Parser 是因为它与词法分析器和语法分析器的后继，词法分析器是对输入的字符串进行分析输出记号流，语法分析器是对记号流进行分析器输出语法树。而 Tree Parser 则是对语法树进行遍历，在遍历的过程中可以嵌入代码（Actions）使用者

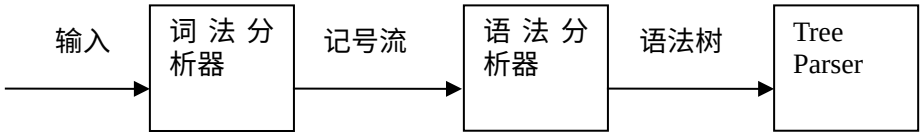


图 7.2

自己来进行后续的工作。Tree Parser 为使用者进行后续工作提供了便利条件。

7.2.1 Tree Parser 与语法分析器（Parser）的相关

ANTLR 中 Tree Parser 要与词法分析器和语法分析器分开写在另一个文件中。文件名也是以“.g”为扩展名。在 Tree Parser 文件中 tree grammar 关键字开头，文件中的文法为 tree 文法。tree 的形式与重写规则的写法一致，但它不是要描述如何建立语法树而是要描述如何遍历语法树。下面我们看一下 Tree Parser 联合工作的例子。

```
grammar TreeParser1;
options { language=CSharp; output=AST;}
vardef : 'int' ID (',' ID)* -> ^('int' ID+);
ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
WS :  ( ' ' | '\t' | '\r' | '\n' )+ { Skip(); } ;
```

这个是一个变量定义的文法 int 类型关键字可以字义 1 到多个整型变量。重写规则把'int'作为根节点并忽略了','字符。下面我们建立“TreeParser1_1.g”文件输入 tree 文件。

```
tree grammar TreeParser1_1;
options {
  language=CSharp;
  tokenVocab=TreeParser1;
}
vardef : ^('int' ID+);
```

在 tree 文法中的 options 设置中有一个新的设置项 tokenVocab，tokenVocab 可以指定此 tree 文法关联着 TreeParser1 文法。实际 tree 文法只关联词法和语法生成的.tokens 文件就可以了，也就是说只将 TreeParser1.tokens 文件与 TreeParser1_1.g 文件同目录就可以了，TreeParser1_1.g 文件并不需要 TreeParser1.g 文件。其中 vardef 规则的也同样的指出了以'int'为根节点 ID 为子节点的结构同，Tree Parser 会按照这个规则遍历语法树。下面看一下运行代码。

```
TreeParser1Lexer lexer
        = new TreeParser1Lexer(new ANTLRFileStream("vardef.txt"));
CommonTokenStream tokens = new CommonTokenStream(lexer);
TreeParser1Parser parser = new TreeParser1Parser(tokens);
TreeParser1Parser.vardef_return r = parser.vardef();
CommonTreeNodeStream treeNodeStream
                = new CommonTreeNodeStream((ITree)r.tree);
treeNodeStream.TokenStream = tokens;
TreeParser1_1 treeParser1_1 = new TreeParser1_1(treeNodeStream);
treeParser1_1.vardef();
```

代码中明显分三个阶段，词法分析到语法分析并生成语法树的代码我们已经很熟悉

了。接下来 CommonTreeNodeStream 类定义树节点流对象，把语法树的节点作为流传入 Tree Parser，treeNodeStream 对象还需要设置词法分析后创建的 CommonTokenStream 对象，这是因为 Tree Parser 需要树节点流和记号流两种信息才能执行。记号就在.tokens 文件中。设置了 treeNodeStream.TokenStream 属性后，将 treeNodeStream 对象作为 TreeParser1_1 构造函数的参数据传入 treeParser1_1 对象中，并调用 treeParser1_1 对象的启始规则 vardef()函数进行遍历。下面我们看一下 Tree Parser 的代码：

```
public class TreeParser1_1 : TreeParser
{
   public static readonly string[] tokenNames = new string[]
    {"<invalid>", "<EOR>", "<DOWN>", "<UP>", "ID", "WS", "'int'",
"','"};
   public const int WS = 5;
   public const int ID = 4;
   public const int EOF = -1;
   public TreeParser1_1(ITreeNodeStream input) : base(input) {
      InitializeCyclicDFAs();
   }
   ......
   public void vardef() // throws RecognitionException [1]
   {
      try {   {
            Match(input,6,FOLLOW_6_in_vardef32);
         Match(input, Token.DOWN, null);
         int cnt1 = 0;
         do {
            int alt1 = 2;
            int LA1_0 = input.LA(1);
            if ( (LA1_0 == ID) ) {
               alt1 = 1;
            }
            switch (alt1)
              {  case 1 : {
                        Match(input,ID,FOLLOW_ID_in_vardef34);
                     } break;
               default:
                     if ( cnt1 >= 1 ) goto loop1;
                      EarlyExitException eee
                         =new EarlyExitException(1, input);
                      throw eee;
            }
            cnt1++;
         } while (true);
```

```
            loop1:
            Match(input, Token.UP, null);
            }
        } catch (RecognitionException re) {
            ReportError(re);
            Recover(input,re);
        }
        finally {}
        return ;
    }
.....
}
```

Tree Parser 从 TreeParser 类继承，代码的结构与语法分析器类似。tokenNames 字符串数组定义了所有的记号。这些记号都是语法树的叶子结点，在遍历语法树时会确认语法树的正确性（语法树的哪些应该位置出现哪些叶子结点进行比对）。vardef()方法的逻辑是先使用 Match(input,6,FOLLOW_6_in_vardef32)匹配'int'节点，'int'的类型值为 6 可以在.tokens 文件中找到'int'的值是 6，ID 的值是 4。然后在 do...while 循环中使用 Match (input,ID,FOLLOW_ID_in_vardef34)匹配 ID 节点，这样就完成了遍历工作。treeParser1_1 在遍历过程中没有做任何事，下面我们在 tree 文法中嵌入 Actions 来实现一些功能。

7.2.2 Tree 文法中嵌入 Actinos

与分析器文法中嵌入 Actions 的规则一样在 Tree 文法内也可以嵌入 Actions。下面我们为 TreeParser1_1 文法嵌入 Actions 来获得定义的变量列表。

```
tree grammar TreeParser1_2;
options {
    language=CSharp;
    tokenVocab=TreeParser1;
    ASTLabelType=CommonTree;
}
@members{
  public List<string> IDList = new List<string>();
}
vardef : ^('int' (ID { IDList.Add($ID.text); })+);
```

在 TreeParser1_2 Tree 文法中定义了一个 IDList 集合属性到来存放遍历语法树后收集的 ID 名称。{ IDList.Add($ID.text); }可以在遍历过程中收集 ID 变量名到 IDList 中。要注意的是这个文法中加入的新的设置项 ASTLabelType=CommonTree 它用来指定树节点的类型。如果没有这个设置项 TreeParser1_2 会把节点当做 object 类型来处理。这样的话$ID.text 写法就会编译不通过因为 object 类型没有 Text 属性。这时对应生成后的代码

为：

```
ID1 = (object)input.LT(1);
Match(input,ID,FOLLOW_ID_in_vardef51);
IDList.Add(ID1.Text);//error
```

设置了 ASTLabelType=CommonTree 选项后代码变成 ID1 = (CommonTree)input.LT(1);，这样就可以使用 CommTree 的属性了。将运行代码换成 treeParser1_2。

```
TreeParser1_2 treeParser1_2 = new TreeParser1_2(treeNodeStream);
treeParser1_2.vardef();
```

7.2.3 Tree 文法

下面我们来研究一下 Tree 文法，Tree 文法作为 ANTLR 中的一种文法用来指示如何遍历语法树，它需要指出语法树的结构。Tree 文法的写法与重写规则类似并且只需描述对应的语法规则就可以了。一般可以把文法中的重规则直接拷贝到 Tree 文法中。Tree 文法中的规则写法并不是一定要与重规则一致。首先语法规则可能用“^”和“！”符号创建语法树。

```
grammar TreeParser1;
……
vardef : 'int'^ ID (','! ID)* ;
……
tree grammar TreeParser1_2;
……
vardef : ^('int' (ID { IDList.Add($ID.text); })+);
```

语法树改成“^”和“！”符号创建之后使用 TreeParser1_2 进行遍历可以得出相同的结果。TreeParser1_2 文法中无需写出','符号，由于 TreeParser1 文法中 ID 是至少出现一次的所以在 TreeParser1_2 中直接使用 ID+来定义。从这个修改的文法中让我们明白的是 Tree 文法规则只用来描述语法树的结构，并不需要受语法规则和重写规则的控制。

```
grammar E;
expression : multExpr (('+'^ |'-'^ ) multExpr)*;
multExpr : atom (('*'^ | '/'^) atom)*;
atom : INT | '('! expression ')'!;
```

对应的 Tree 文法可以写成：

```
grammar E_Walk;
options{
   language=CSharp;
   tokenVocab=E;
   ASTLabelType=CommonTree;
}
expression : ^('+' expression expression)
```

```
| ^('-' expression expression)
| ^('*' expression expression)
| ^('/' expression expression)
| INT;
```

Tree 文法中的符号也可能不与原文法中一一对应，些 Tree 文法会生成一个函数递归来遍历语法树。语法某个规则如果没有特殊指定语法树结构，在 Tree 文法中书写相同的规则就可以了。请看下面的例子：

```
expression : fieldName | STRING | INT;
```

对应 Tree 文法中可以相同书写：

```
expression : fieldName | STRING | INT;
```

如果语法规则中有多余项目，Tree 文法本着描述语法树结构的宗旨可以省去一些不必要的规则，如下面的文法 a，b，c 三规则中 a，b 是多余的，所以在 Tree 文法中可以省略 a 和 b。

```
a    :    b;
b    :    c;
c    :    'c';
```

Tree 文法：

```
a    :    c;
c    :    'c';
```

下面看一个关于 if 语句的 Tree 文法示例：

```
grammar TreeParser2;
……
ifstat
: 'if' '(' expression ')' s1=stat
( 'else' s2=stat -> ^('if' ^(EXPR expression) $s1 $s2)
  | -> ^('if' ^(EXPR expression) $s1)
);
```

对应的 Tree 文法可以写成：

```
tree grammar TreeParser2_1;
...
ifstat : ^('if' ^(EXPR expression) stat stat?) ;
```

TreeParser2 文法中的两个重写规则在 TreeParser2_1 中合并书写 else 子句是可选项所以使用？标识。不过要注意大 Tree 文法中描述语法树结构时一定要与文法的符号名一致，不能缺少节点。Tree 文法应该详细到词法规则符号

如修改 TreeParser1 文法，定义 QID 词法规则，而 ID 规则成为 fragment 词法规则。

```
grammar TreeParser1;
……
```

```
vardef : 'int'^ QID (','! QID)* ;
QID : ID;
fragment ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
WS : ( ' ' | '\t' | '\r' | '\n' )+ { Skip(); } ;
```

这时 tree 文法必须改成 QID，由于语法树中只包含语法规则直接关联的词法规则符号作为叶子节点，不包含对应 fragment 词法规则的节点，所以在 Tree 文法中不可以加入规则“QID：ID;”。甚至如果 ID 不是 fragment 词法规则 Tree 文法中也是不能加入“QID：ID;”规则，因为 ID 对于语法规则来说是间接的。

```
tree grammar TreeParser1_2;
……
vardef : ^('int' (QID { IDList.Add($QID.text); })+);
```

7.2.3 使用 Tree 文法来实现 SELECT 语句的遍历

本章第一个示例 Select 中我们自己编写了遍历语法树的代码，来获得 SELECT 语句的信息并放到我们定义好的类中。现在我们使用 Tree 文法来实现。下面请看 Tree 文法：

```
tree grammar Select_Walk;
options {
   language=CSharp;
   tokenVocab=Select;
   ASTLabelType=CommonTree;
}
selectStatement : ^(SELECT_STATEMENT selectClause fromClause whereClause?);
selectClause : ^(SELECT_LIST '*')
             | ^(SELECT_LIST fieldName+);
fromClause : ^(TABLE_LIST tableSource+);
whereClause : ^(WHERE_CONDITION searchCondition);
searchCondition     : searchItem ('AND' searchItem)*;
searchItem : ^(COMPARE_ITEM expression ('=' | '>' | '<' | '<>') expression)
             | expression 'IS' 'NOT'? 'NULL';
expression : fieldName | STRING | INT;
tableSource : ^(tableName selectStatement?);
fieldName : ^(FIELD_NAME Identifier ('.' Identifier)*);
tableName : Identifier;
```

Select_Walk 文法中 selectStatement 规则与 Select 中的重写规则相同。selectClause 规则是把原重写规则中的两个分支并列地书写。fromClause 和 whereClause 规则也是照搬原文法中的重写规则。searchCondition 规则与原规则一致。searchItem 规则的第一个分支使用了重写规则的写法，第二个分支与中将 expression 符号加入使规则完整。由于在 Tree 文法中不能对应使用文法中的$p 变量，所

以只能使用('=' | '>' | '<' | '<>')符号的写法。Expression，tableSource，fieldName 和 tableName 规则与前几个规则类似。所有的规则写好后可以生成 Select_Walk.cs 文件。不过这时的遍历器还没有任何功能，我们向 Tree 文法加入 Actions 来实现收集信息的功能。

```
tree grammar Select_Walk;
options {
   language=CSharp;
   tokenVocab=Select;
   ASTLabelType=CommonTree;
}
@header {
using System.Collections.Generic;
}
selectStatement returns[Select retSelect]
scope {
 Dictionary<string, List<string>> SelectListDic;
 Select select;
}
@init {
 $selectStatement::select = new Select();
 $selectStatement::SelectListDic =
          new Dictionary<string, List<string>>();
}
@after {
 $retSelect = $selectStatement::select;
}
: ^(SELECT_STATEMENT selectClause fromClause whereClause?);
```

与自己编写遍历程序一样我们要定义收集 SELECT 语句中查询的字段列表信息临时放到 SelectListDic 集合中，还要定义一个 Select 对象来存储 SELECT 语句的信息。SelectListDic 与 Select 对象都是相对于 SELECT 语句的如果有子查询则嵌套 Select 对象。这正好可以用 Scope 属性来实现，属性生命期覆盖 selectStatement 规则，selectStatement 规则函数执行完成后返回分析结果 Select 对象。SelectListDic 和 Select 对象在@init 中进行初始化，在@after 中返回结果。

```
selectClause : ^(SELECT_LIST '*')
 | ^(SELECT_LIST (retFN=fieldName
{if(!$selectStatement::SelectListDic.ContainsKey($retFN.TableName))
    $selectStatement::SelectListDic
                .Add($retFN.TableName, new List<string>());
 $selectStatement::SelectListDic[$retFN.TableName]
                .Add($retFN.RealFieldName);
 } )+ );
```

```
fromClause : ^(TABLE_LIST tableSource+);
whereClause : ^(WHERE_CONDITION searchCondition);
searchCondition   : searchItem ('AND' searchItem)*;


searchItem : ^(COMPARE_ITEM leftE=expression
              (p='=' | p='>' | p='<' | p='<>') rightE=expression)
{ string leftTableName = $leftE.TableName;
  string leftFieldName = $leftE.FieldName;
  string rightTableName = $rightE.TableName;
  string rightFieldName = $rightE.FieldName;
  if(leftTableName != "" && leftTableName != null
     && leftFieldName != "" && leftFieldName != null
     && rightTableName != "" && rightTableName != null
     && rightFieldName != "" && rightFieldName != null)
  { string key = String.Format("{0}-{1}", leftTableName, rightTableName);
    string conditionStr = String.Format("{0}.{1} {2} {3}.{4}",
           leftTableName, leftFieldName, $p.text,
           rightTableName, rightFieldName);
    if (!$selectStatement::select.Relations.ContainsKey(key))
      $selectStatement::select.Relations.Add(key, new Relation());
    Relation relation = $selectStatement::select.Relations[key];
    relation.LeftTable=
           $selectStatement::select.SelectTables[leftTableName];
    relation.RightTable =
           $selectStatement::select.SelectTables[rightTableName];
    $selectStatement::select.Relations[key]
           .RelationCondition.Add(conditionStr);
  }
 }
 | expression 'IS' 'NOT'? 'NULL';

expression returns[string TableName, string FieldName]
: fieldName
  { $TableName=$fieldName.TableName;
    $FieldName=$fieldName.RealFieldName; }
  | STRING
  | INT;

tableSource
: ^(tName=tableName
{ Table table = new Table();
  string temp = $tName.text;
  table.Name = ((Antlr.Runtime.Tree.CommonTree)
```

```
        (((Antlr.Runtime.Tree.TreeRuleReturnScope)(tName)).start)).Text;
    $selectStatement::select.SelectTables.Add(table.Name, table);
    if($selectStatement::SelectListDic.ContainsKey(table.Name))
      $selectStatement::select.SelectTables[table.Name].
          SelectList.AddRange($selectStatement::SelectListDic[table.Name]);
  }
  (retS=selectStatement
  { $selectStatement::select.SelectTables[table.Name].SubSelect
                                        = $retS.retSelect;
  } )? );
```

与自己编写遍历程序一样，主要分析三部分：一 SELECT 查询列表查找出所有要查询的字段。二 FROM 语句部分查询了哪些表，是否有子查询。三 WHERE 条件中有哪些表的相连信息

```
fieldName returns[string TableName, string RealFieldName]
@init{
  string tName = "";
}
: ^(FIELD_NAME fName=Identifier
   (  '.' {tName=$fName.text;} fName=Identifier)*
     {$TableName = tName; $RealFieldName = $fName.text;}
   );
tableName : Identifier;
```

为了操作方便 fieldName 定义两个返回值分别是切割的表名和字段名。本示例认为一个长的标识符中间以“.”分隔的最后的一个名字为字段名，倒数第二个名字为表名。通过这个例子可以看出使用 Tree 文法使我们可以免去很多遍历语法树的工作，使用 Actions 和参数可以很灵活地操作和传递数据。

我们可以利用重写规则来改变树法树的结构从而使遍历程序更方便地处理信息。例如我们可以修改 Select_Walk 示例调换其语法树的先后位置让遍历程序 Select_Walk 先遍历 fromClause 子树再遍历 selectClause 子树，这样可以先收集查询中的表，然后再收集表的字段。所以可以删除掉 SelectListDic 集合，只使用 Select 对象就可以完成操作。下面 Select1 文法是对 Select 文法的修改只是将 selectStatement 规则中 fromClause selectClause 两个符号交换位置。

```
grammar Select1

selectStatement : selectClause fromClause whereClause?
                  -> ^(SELECT_STATEMENT fromClause selectClause
whereClause?);
……
```

下面是 Select_Walk1 Tree 文法适应相对于 Select_Walk 文法的变化，selectStatement 规则中的 fromClause selectClause 交换位置。tableSource 规则中只收集数据表名，而 selectClause 规则中找到的字段直接分配到数据表中

```
tree grammar Select_Walk1;
......
selectStatement returns[Select retSelect]
scope {
  Select select;
}
@init {
  $selectStatement::select = new Select();
}
......
: ^(SELECT_STATEMENT fromClause selectClause whereClause?);

selectClause : ^(SELECT_LIST '*')
             | ^(SELECT_LIST (retFN=fieldName
{                                                                    if(!
$selectStatement::select.SelectTables.ContainsKey($retFN.TableNam
e))
        throw new Exception($retFN.TableName + " is not exist.");
      $selectStatement::select.SelectTables[$retFN.TableName]
                          .SelectList.Add($retFN.RealFieldName);
               } )+ ) ;
......
tableSource
: ^(tName=tableName {
  Table table = new Table();
  string temp = $tName.text;
  table.Name = ((Antlr.Runtime.Tree.CommonTree)
                              (((Antlr.Runtime.Tree.TreeRuleReturnScope)
(tName)).start)).Text;
                $selectStatement::select.SelectTables.Add(table.Name,
table);
       }
      (retS=selectStatement
       { $selectStatement::select.SelectTables[table.Name].SubSelect
                                          = $retS.retSelect;
       } )?
      );
```

7.3 属性文法

前面我们讲了如何遍历语法树，也在某种程度上做了语义分析的工作。下面我们学习一下编译原理中的语义分析方法。在编译原理中最常用的语义分析方法是语法制导翻译方法。语法制导翻译方法在文法的基础上为文法规则加入语义规则形成**属性文法**。属性文法也

就包括**文法**和**语义规则**两部分。我们在前几章一直在讨论文法，语法制导翻译方法是是以文法规则或语法树为导向的语义分析，它的执行依赖文法规则的结构受文法规则形式的影响。

属性文法中的属性可以是语言元素的任意特性，语义规则是属性之间的操作规则。所谓语言元素就是文法中的符号。如一个整型类型在文法定义为符号是 INT，对于整型类型来说它的值是一个重要属性，INT 符号也可能有最大值和最小值的属性用来确定取值范围，还有前面遇到的 SQL 文法中的标识符有 TableName 和 FieldName 两个属性。语义规则正是这些规则之间的操作的集合，如一个符识符的值赋给了一个数据表做为表名 tableName.Name = ID.text 就属性 text 与属性 Name 之间的操作。

在属性文法中一个文法符号 A 对于它的一个属性 a 的写形式为 A.a。对属性的求值并与文法符号的关联过程叫做联编(binding)，联编也就是前面说的属性之间的操作。属性计算求值所用的时间为联编时间（如求一个表达式的值并将值赋予 express.value 属性的过程）。现有的各种语言和各种不同的语言元素的属性计算方式是不同的，有些属性是静态确定有些是动态计算确定的。象 C#，java 这样强类型的语言其变量的类型是在语言中静态定义的，而象 javascript 变量的类型的运行时动态确定的。包含变量表达式的值是在运行时确定的，它也就是所谓的动态联编。

文法制导翻译中属性与文法符号密切关联。对于文法中有的任一规则 Ai 关于它其中一个属性 aj 可写成 Ai.aj，在属性文法的语义规则中关于属性 ai 的操作可以用一个函数 f 来表示。这样对于某一属性 Ai.aj 的一个语义规则可以表示为 Ai.aj= f(Am.ax..Am.ay, … An.ax..An.ay)，Am 到 An 是参与计算的规则，ax 到 ay 是某规则参与计算的属性。看过第五章之后我们知道在 ANTLR 中实现属性很简单我们有参数、返回值、Scope 属性都可以使用。下面看一个整型 INT 的属性文法示例。

INT : INT DIGIT | DIGIT;

DIGIT : ‘0’..’9’+;

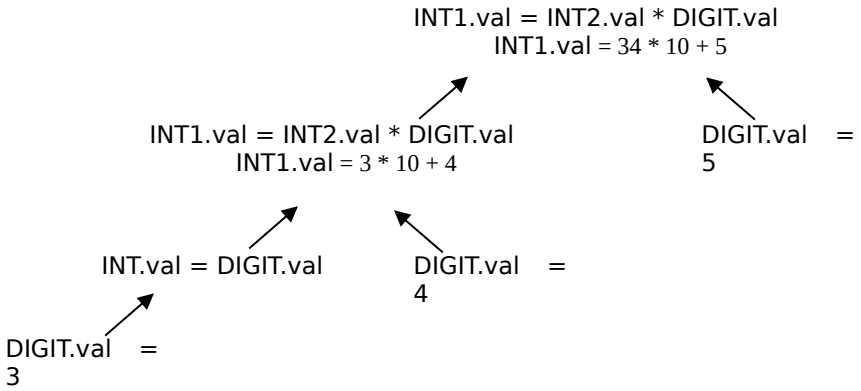
这两个规则定义了一个整型数。对于整型数最重要的属性就是数值，所以首先要实现求值。我们给 INT 和 DIGIT 加上 val 属性，DIGIT 只有一位所以它的值可以直接获取。而 INT 的值要将各位上的值乘以权值 10（只处理十进制数）。下面给出语法规则和语义规则的对应表：

文法规则	语义规则
INT : INT DIGIT	INT1.val = INT2.val * DIGIT.val
INT : DIGIT	INT.val = DIGIT.val
DIGIT : '1'	DIGIT.val = 1
DIGIT : '2'	DIGIT.val = 2
DIGIT : '3'	DIGIT.val = 3
DIGIT : '4'	DIGIT.val = 4
DIGIT : '5'	DIGIT.val = 5
DIGIT : '6'	DIGIT.val = 6
DIGIT : '7'	DIGIT.val = 7

DIGIT : '8'	DIGIT.val = 8
DIGIT : '9'	DIGIT.val = 9

表 7.1

语法规则的每一个分支都定义相应的语义规则。有了这些语义规则就可以在语法分析的过程中计算出 INT 的值。要注意在文法规则 INT : INT DIGIT 中出现两个相同符号时，在语义规则中要用下标区分开。有以前学到的语法分析和语法树的知识很好理解这些。下面给出一个关于此属性文法的结构图，结构图中用箭头属性值的走向。以 345 为例：



结构图中所有 val 属性之间的箭头都是从子节点到父节点，父节点 val 属性的计算依赖子节点 val 属性的值，这样的属性叫做合成属性。一个 X => X1 X2 … Xn 规则对于合成属性 a 有 X.a = f(X1.a, X2.a,… Xn.a)。其中 X1 到 Xn 并不一定全部参与计算。合成属性的求值是父节点依赖子节点的属性值，所以要先计算子节点的属性值，这可以用树的后续遍历来求值。

语义制导方法的语法并不限于 ANTLR 中的语法规则和语法树，在编译原理中这个概念也包括词法规则，上例的 INT 在 ANTLR 中就会被定义成词法规则。INT 示例只是为了说明属性文法是如何工作的，一般在 ANTLR 中对于类似 INT 规则无需使用这样的方法来求值，使用一条(int)INT.text 语句就可以了。

本章第一个示例中我们将语法树中的信息收集后放入了我们定义的能够体现信息的结构和语义的类模型中（如 Select 类），而语法制导翻译方法中有些部分不需要定义模型类，因为可以依靠语法规则或语法树本身的结构来了解信息的结构并同时完成了属性的求值，这样速度会有很大提升。而有些部分则需要收集信息后放到类似于 Select 类这样的符号表中再进行后面的工作。下面我们再举一个具体多个属性的属性文法例子。

一种类型数值有它的取值范围，所以可以使用最大值和最小值两个属性来标记检查数据是否在有效的取值范围内。下面看一下文法规则，一个变量定义的文法定义了 Int32 和 Int16 两种类型，Int16 的取值范围是：-32768 ~ 32767，Int32 的取值范围是：-2147483648~ 2147483647。

```
vardef : type id '=' intValue-> ^(type intValue'=' id);

type : 'Int32' | 'Int16';

id : ID;
```

ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;

intValue : INT;

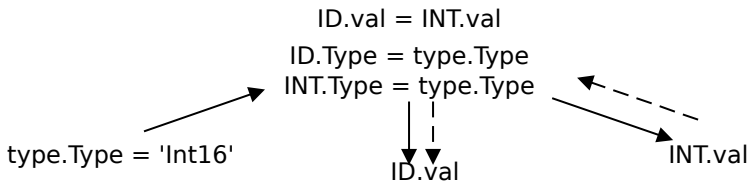
INT : '0'.. '9'+;

由于非 fragment 词法规则不能设置参数和返回值，所以我们把所有的规则都改成语法规则。这个文法我们使用 ANTLR 中的写法其中 ID 有三个属性，val 属性代表变量的值，Type 属性代表变量的类型。INT 也有 val 属性和 Type 属性作用于 ID 的相同。ID 还有一个 Name 属性代表变理名。下面我们看一下属性文法：

文法规则	语义规则
vardef : type ID '=' INT;	ID.val = INT.val ID.Type = type.Type INT.Type = type.Type
type : 'Int32';	type.Type = 'Int32'
type : 'Int16';	type.Type = 'Int16'
ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;	获得 ID 的 Name 属性的语义规则简略
INT: '0'.. '9'+;	获得 DIGIT.val 属性值的语义规则简略

表 7.2

以 Int16 X1 = 23 为例我们看一下属性文法的结构图。实线箭头是 Type 属性的走



向，虚线是 val 属性的走向。

vardef : type ID '=' INT 规则作为一个桥梁 type 规则的 Type 属性分别赋值给 ID 和 INT 的 val 属性将类型信息传递给了数值符号本身，而初始化值 INT.val 赋给了变量 ID 的 val 属性。在这个属性文法中 INT 的 val 属性与之前 INT 示例中不同，我们省略 INT 形成的过程不用去考虑前一位乘 10 加后一位的操作过程，这在 ANTLR 中也确实没有必要。之前 INT 示例中 val 是一个合成属性。本示例中 Type 属性和 val 都属于继承属性，属性文法对继承属性的定义是：除合成属性外其它的属性都是继承属性。

继承属性在符号之间互相传递可能有两种方式，一是从父节点向子节点传递，二是子节点之间的传递。这二种传弟方式对于不同的语法树组织结构也会有不同，第一种：树一般的结构是父节点对每一个子节点都有一个指针。第二种：还有一种树是从父节点有一个指针指向第一个子节点，然后子节点有一个指针指向它的右兄弟以此类推。这两种树结构使属性的传递会有不同。

父节点向子节点传递　　图 1　　子节点之间的传递

图 1 显示了对于第一种树结树继承属性的传递方式，父节点向子节点可以直接传递，子节点之间的传递要通过父节点。第二种树结构父节点向子节点传递时要经过第一个子节点然后向右传递。节点之间可以从左向右传递。ANTLR3 的语法树属于第一种情况，继承属性的传递要通用父节点。

父节点向子节点传递　　图 2　　子节点之间的传递

7.3.1 ANTLR 中如何传递属性

属性的传递在 ANTLR 中有多种方法，第一种是使用参数和返回值。以前面 INT 示例为例我们写出一个 ANTLR 文法。

```
tree grammar AttrGrammar1;
options {
    language=CSharp;
    tokenVocab=Attr;
    ASTLabelType=CommonTree;
}
vardef : type intValue [$type.Type] '=' id[$type.Type, $ intValue.val];
type returns[string Type] : a='Int32' | a='Int16'{ $Type = $a.text };
id [string Type, int val] returns[string Name]
: ID { $Name = $ ID.text; };
intValue [string Type] returns[int val] : INT
{  $val = (int)$ INT.text;
   if($Type == "int16" && ($val < -32768 || $val > 32767))
      throw new Exception("数值不在有效的范围内。");
};
```

我们看到 vardef 规则 INT 与 ID 的顺序调换了，这需要使用重写规则来改变一下符号顺序。因为要先知道 INT 的值才能赋给 ID。type 的 Type 属性作为 type 规则的返回值来定义，可以返回类型名。ID 规则有两个参类分种是 type 规则提供的 Type 属性和 INT 提供的 val 属性。ID 规则的 Name 属性作为返回值定义。INT 规则接收 Type 参数在 INT 规则中可

以判断一下值的有效范围。INT 规则作为返回值也定义了 val 属性。使用参数和返回值可以较好的实现属性文法，但是正如实例中要用到重写规则来改变语法树的节点次序。参数和返回值很受遍历顺序的限制，有时也不完全能用重写规则来解决，因为实际情况是很复杂的。有时 y 就需要一遍以上地遍历语法树。但这时我们也可以考虑在语法分析阶段用 Actions 将部分的属性先计算出来，然后在遍历语法树的时候直接取值来避免多次遍历语法树。就下面我们用这种方法来实现 AttrGrammar1 文法，先看一下修改后的文法。

```
grammar TowStep;
options {
   language=CSharp;
   output=AST;
}
tokens {VAL;}
vardef : type id a='=' intValue
          -> ^(type VAL[$a, $intValue.text] id '=' intValue);
type : 'Int32' | 'Int16';
id : ID;
ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
intValue : INT;
INT: '0'.. '9'+;
WS : ( ' ' | '\t' | '\r' | '\n' )+ { Skip(); };
```

我们在文法中定义一个 VAL 虚拟节点，利用重写规则将 intValue 求值然后将值作为 VAL 节点的 text 属性，VAL 节点的位置在 id 之前这样在遍历语法树时 id 就可以从 VAL 中获得 intValue 的属性值了。下面是看一下 tree 文法 TowStep_Walk。

```
tree grammar TowStep_Walk;
options {
   language=CSharp;
   tokenVocab=TowStep;
   ASTLabelType=CommonTree;
}
vardef : ^(type VAL id[$type.Type, Convert.ToInt32($VAL.text)]
       '=' intValue[$type.Type]);
type returns[string Type] : (a='Int32' | a='Int16') { $Type = $a.text; };

id[string Type, int val] returns[string Name] : ID
```

```
{ $Name = $ID.text;
  if($Type == "Int16" && ($val < -32768 || $val > 32767))
    throw new Exception("数值不在有效的范围内。");
};
intValue [string Type] returns[int val] : INT
{  $val = Convert.ToInt32($INT.text);
  if($Type == "Int16" && ($val < -32768 || $val > 32767))
    throw new Exception("数值不在有效的范围内。");
};
```

前面两个示例说明了用 ANTLR 如何实现属性文法。大多数情况使用参数和返回值可以很好的实现属性文法，当然在有些情况下可以使用 scope 属性和 members 类员，这些就看具体的情况而定。在语义规则的赋值方向与遍历的顺序相反时我们可以利用重写规则在文法中改变节点的顺序也可以在语法分析时先计算出一些局部的结果，在以后的遍历语法树时再利用。　甚至将计算结果保存到外部的我们自己定义的数据结构中。

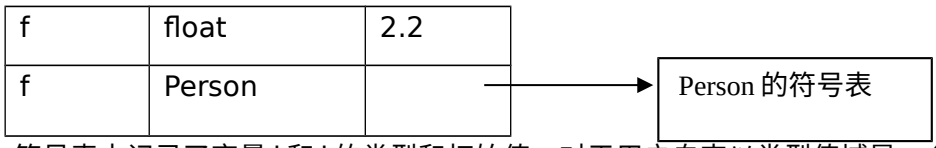
7.4 符号表

本章第一个示例 Select 中我们定义了一组类来记录遍历语法中的收集的信息，TowStep 示例中我们也可以不增加虚拟节点，而将语法分析阶段计算的值放到分析器以外的存储结构中，然后在 TowStep_Walk 中获取这个值。在这些情况下分析器以外的数据结构实际上是用到了符号表的功能。编译过程中的多个阶段都会用到符号表。符号表主要有三个主要作用：第一是记录符号的属性。符号表是以名称为索引的集合，集合中可以保存符号的属性值。如 TowStep 示例中将 intValue 的值可以保存到符号表中。第二是语法检查的依据。在上下文无关文法由于无法知道上下文的信息。所以语法分析时可以在变量的声明时将其名称，类型入到符号中，在后面分析时如果遇到这个变量可以在符号表的查找这个变量是否定义以及类型是否正确等类似的语法检查。第三符号表是代码生成地址分配的依据。请看下面的代码和它的符号表。

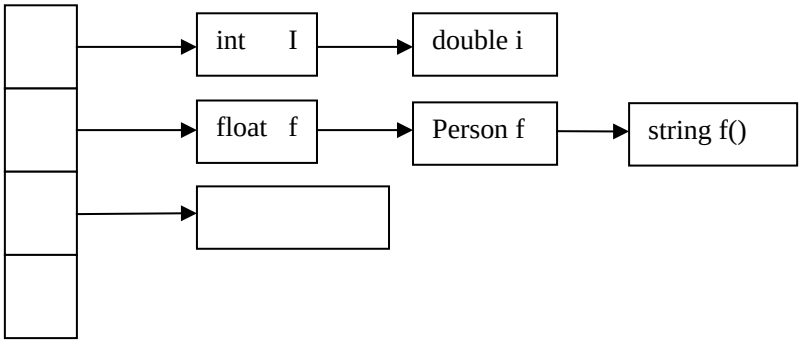
符号名可能是原代码中一个变量名，一个方法名，一个类名或一个属性名。符号的长度根据语言不同有不同的最长限制。符号的类型可能是简单类型，也可能是复杂类型（如结构，类等）这些也可能是用户自定义的类型。这些类型中有些类型是语法分析时静态确定的，有些类型是在运行时动态确定的。如表达式的计算过程中不同类型运算会根据操作数和计算结果才能确定。面向对象语言中的多态性代码要在运行时才能检查类型的正确性。

```
int i = 100;
float f = 2.2;
Person f = new Person;
```

名称	类型	值
i	int	100



符号表中记录了变量 i 和 j 的类型和初始值，对于用户自定义类型值域是一个指针指向自定义类型的符号表。为了便于在符号表中查找我们可以在建立符号表时将符号表按照符号名排序。这样可以利用二分法等算法来快速查找到符号以提高分析器的执行效率。也可以用符号的首字母来把符号分类提高查询效率，但是按标识符的首字母区分会不平均有些字母开头的标识符很多而有些很少。所以许多编译器使用 hash 表来实现符号表的入口，hash 表使用特定的算法来将符号名计算出一个在一定范围内的随机数，数值相同的变量放在一起。下面是使用 hash 表符号表的示意图。



java 与.net 中有许多集合类可以

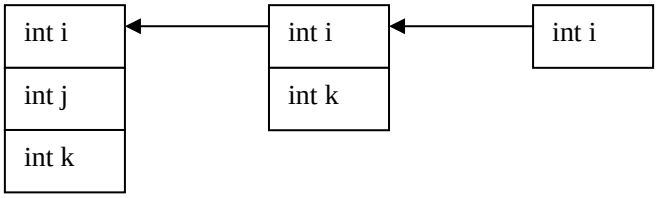
7.4.1 符号表的作用域及可见性

一个符号变量在程序中的可用的范围是它的作用域。一般符号在源程序中的位置决定了符号的作用域。比如在 java 类中定义的私有变量作用域在整个类中，在类的方法里定义的变量的作用域只在这个方法中，在方法内部的代码块{}内定义的变量的作用域只在代码块内。很多语言在不同的作用域中是可以用相同的变量名的。相同名称的变量名在符号表存放里要加上作用域的前缀以区分，或以作用域分成多个符号表来存储。

```
class cla {
   int i,j,k
   void f() {
      int i,k;
      {
            int i;//位置 1
            i = 5;
         k = 4;
         j = 6;
      }
   }
}
```

上面程序段中在嵌套的作用域内，有多个同名的变量 i 这时变量的引用里以内层优先，在位置 1 引用变量 i 时是最内层的变量 i 起作用，这时外两层的 i 被隐藏。根据不同的作用域将创建多个符号表，符号表之间用指针由内层指向外层。当内层找不到引用的变量时由指针向外层查找直到最外层。

在面向对象语言中作用域变得更加的复杂，在一个代码块中引用的符号可以是本方法



的，也可能是类的成员，也可能是基类的成员。所以不但要在本类中查找，还要延着继承链向在基类中查找，直到最顶层的基类。SQL 中 SELECT 语句嵌套子查询的情况也存在这种嵌套关系的符号表。在子查询里的引用的是父查询的符号。

虽然符号表不属于 ANTLR 支持的范围内，ANTLR 也为创建符号表提拱了一些便利条件。init 和 after 可以很方便地在规则的开始和结束时创建和外理符号表。也可以直接利用 Scope 属　　性来实现嵌套作用域的符号表。

有时在分析源代码时会出现引用在前声明在后的情况，例如在代码开始处调用后一个后面才定义的函数。这时要求分析器能够在正式分析前扫描一遍所有的符号建立符号表。这样就要多遍分析。有些语言如 C++和 pascal 使用向前引用声明来避免多遍的扫描。

在 java 和.net 中我们可以利用系统提拱的强大的集合类来实现符号表。现代化的集合类使我们可以节省很多代码。Java 中泛型的 Map 集合与.net 中泛型的 Dictionary 集合都可以用来实现符号表，做插入、查找等操作都比较方便。

向前引用

面向对象的符号查找的复杂性

参数返回值

ANTLR 功能的利用