Constructing Reduced Derivation Trees

Adrian Johnstone Elizabeth Scott

Technical Report

CSD – TR – 98 – 09

October 12, 1998

Royal Holloway
University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

Abstract

The primary role of a parser is to check that its input is syntactically correct. Effectively, it does this by constructing a derivation of the input from the grammar rules. However, 'parsing is not translation' [Par96]: syntax analysis usually involves more than just this correctness check. If the input is correct then the syntax analyser may produce a 'parse tree' which is the first step in the semantic analysis of the input. This reflects the fact that the syntax of a programming language is *chosen* by the designer to allow constructs with particular meanings to be built into the language. Thus in a real sense the syntax of a language is not separate from the semantics. The syntax is the first step in defining the semantics, which makes it natural for the output of the syntax analysis stage to be a structure from which much of the semantics of the input can be constructed.

The purposes of this report are to establish the motivation for using something other than a full derivation tree in the translation process, to propose that grammars should be defined so that trees appropriate for translation purposes are produced as a result of the parsing stage, to give a formal definition of a *Reduced Derivation Tree* (RDT) as a tree which can be obtained from a derivation tree in specified ways, to give a formal definition of a canonical reduced derivation tree, and to describe methods of generating the required reduced derivation trees in cases where the chosen parsing technique does not allow the use of a grammar which would generate such trees 'naturally'.

Our RDTs are defined independently of any particular parsing technologies, and we describe how to build them in conjunction with both standard top-down and bottom-up parsers.

Contents

1 Introduction **1**
1.1 Basic definitions 1
1.2 The operation of a grammar 2

2 Syntax and semantics **4**
2.1 Parsing and tree construction with BNF grammars 4
2.2 Attribute evaluation 4
2.3 Changing the shape of the trees 5

3 Reducing the resolution in parse trees **7**
3.1 Unnecessary terminals 7
3.2 Unnecessary non-terminals 7
3.3 Terminals that provide structure 8

4 Folding **10**
4.1 Convergent Folds 10
4.2 Promoting nodes 11

5 Annotations Based on Promotion Operators **12**
5.1 Initial example 13
5.2 Generating RDTs from annotated BNF grammars 14
5.3 Limitations of the promotion operator notation 15
5.3.1 Example – promotion priorities 15
5.3.2 Example – conditional promotions 16
5.3.3 Example – redundancy 17

6 Formal description of RDTs using syntax-directed definitions **20**
6.1 Example 21
6.2 General form of syntax-directed definitions for RDTs 22
6.3 The function `build()` 22
6.4 Example 23
6.5 Overcoming limitations of annotated grammars 25
6.5.1 Example – promotion priorities revisited 25
6.5.2 Example – conditional promotions revisited 25
6.6 Using promotion operators and syntax-directed definitions 26
6.6.1 Example 26

7 Canonical RDTs for BNF grammars **28**
7.1 Algorithm for generating a syntax-directed definition from a BNF grammar 28
7.2 Example 28
7.3 Example 29
7.4 Remarks about the canonical form 31

8 Canonical RDTs from edited BNF grammars **33**
8.1 Algorithm for annotating an edited BNF grammar 33
8.2 Example 34

9 Building trees as the input is read **36**
9.1 Example 36
9.2 Using translation schemes and a single promotion operator to build TDRs 37
9.3 Example 38
9.4 Example 38
9.5 Algorithm for tree construction when parsing with an annotated translation scheme 38
9.6 Example 40

10 Canonical RDTs via translation schemes **43**
10.1 Algorithm to generate an annotated translation scheme from an edited BNF grammar 43
10.2 Example 44
10.3 Example 44

11 Changing the shape of the RDT **46**
11.1 Left associative operators 46
11.2 Promotion of a child node above its parent 47
11.3 Specifying 'promote above' in a translation scheme 48
11.4 Algorithm for tree construction from a general translation scheme 48
11.5 Example 50
11.6 Left-associative operators from right-recursive translation schemes 52

12 `rdp`'s annotation operators **55**
12.1 `rdp` input format 55
12.2 Promotion operators in terms of general translation schemes 56
12.3 Example 57

1 Introduction

The primary role of a parser is to check that its input is syntactically correct. Effectively, it does this by constructing a derivation of the input from the grammar rules. However, 'parsing is not translation' [Par96]: syntax analysis usually involves more than just this correctness check. If the input is correct then the syntax analyser may produce a 'parse tree' which is the first step in the semantic analysis of the input. This reflects the fact that the syntax of a programming language is *chosen* by the designer to allow constructs with particular meanings to be built into the language. Thus in a real sense the syntax of a language is not separate from the semantics. The syntax is the first step in defining the semantics, which makes it natural for the output of the syntax analysis stage to be a structure from which much of the semantics of the input can be constructed.

It is our belief that the concept of an abstract syntax tree is not well defined in the literature. The purposes of this report are to establish the motivation for using something other than a full derivation tree in the translation process, to propose that grammars should be defined so that trees appropriate for translation purposes are produced as a result of the parsing stage, to give a formal definition of a *Reduced Derivation Tree* (RDT) as a tree which can be obtained from a derivation tree in specified ways, to give a formal definition of a canonical reduced derivation tree, and to describe methods of generating the required reduced derivation trees in cases where the chosen parsing technique does not allow the use of a grammar which would generate such trees 'naturally'.

1.1 Basic definitions

Formal BNF grammars are used to specify languages and traditionally the 'operation' of a grammar is given by describing how it is used to derive (generate) sentences in the language.

A *BNF grammar* has a set T of terminal symbols, a set N of non-terminal symbols (here T and N are disjoint), a distinguished element $S \in N$ called the start symbol, and a set of production rules of the form $X ::= \alpha$, where $X \in N$ and α is a string of elements from $T \cup N$.

Each terminal $t \in T$ has a corresponding *pattern* which is the set of strings of input characters corresponding to t. An element of the pattern of t is a *lexeme* of t.

A *derivation step* is of the form

$$\delta X \gamma => \delta \alpha \gamma$$

where δ, γ are elements of $(T \cup N)^*$ and $X ::= \alpha$ is a production in the grammar.

A *derivation* is a sequence of derivation steps which begins with the start symbol and ends with a string of terminals

$$S => \beta_1 => \beta_2 => \ldots => u \in T^*$$

We say that $\beta \in (N \cup T)^*$ can be *derived* from $X \in N$ if there is a sequence of derivation steps

$$X => \beta_1 => \beta_2 => \ldots => \beta$$

A *sentence* of a grammar is a string $u \in T^*$ which can be derived from the start symbol, and the *language* generated by a grammar is the set of sentences of that grammar.

1.2 The operation of a grammar

The 'operation' of a grammar is often described in terms of the derivations which can be generated from it. In this we report describe the operation of a BNF grammar by describing how it is used to generate derivation trees. We also use this approach to describe the operation of 'annotated BNF grammars' by describing how they are used to generate reduced derivation trees.

A *derivation tree* is an ordered tree whose interior nodes are associated with non-terminals in the grammar, and whose leaf nodes are associated with terminals. The root node is associated with the start symbol, and the children of an interior node associated with X are associated with the symbols which correspond to the RHS of some production rule for X. The order of the children corresponds to the order that the symbols appear in the production rule. The *yield* of the tree is the sentence obtained by reading the terminals associated with its leaf nodes, in left-to-right order.

There is a direct association between derivation trees and derivations, in which the yield of the tree is the sentence constructed at the end of the derivation. For example in the grammar

```
S ::= S + S | a
```

the derivation

```
S => S + S => a + S => a + S + S => a + a + S => a + a + a
```

is associated with the tree

```
        S
      / | \
    S   +   S
    |     / | \
    a    S  +  S
         |     |
         a     a
```

Technically, a derivation tree is defined in terms of a grammar and a derivation. Two derivations are said to be *equivalent* if they are associated with the same derivation tree, and in this way a derivation tree corresponds to an equivalence class of derivations. Equivalent derivations derive the same sentence but it is possible for the same sentence to correspond to two or more distinct equivalence classes of derivations. If there is a sentence which corresponds to two or more equivalence classes of derivation then the grammar is *ambiguous.* Equivalently, a grammar is ambiguous if there are two or more distinct derivation trees which have the same yield.

If a grammar is not ambiguous then the derivation tree produced from a given input sentence will be unique, and we can define a derivation tree in terms of a grammar and a sentence. The intention is that the algorithms described below will be applied to non-ambiguous grammars, and we shall refer to the derivation tree defined or generated by a grammar and a given input sentence.

2 Syntax and semantics

2.1 Parsing and tree construction with BNF grammars

Derivation trees allow a derivation to be reconstructed (up to order of non-terminal replacement). They can also be used for semantic evaluation. It makes sense for this evaluation to be 'local'. In particular it is compositional, the meaning of a construct is defined in terms of the meanings of its components. It is difficult to retain unstructured information. We can remember the meaning of a small number of individual items independently. But if we wish to know the meaning of a vast number of sentences then it is natural to learn the meanings of basic elements (words) and to learn a few rules for constructing the meaning of complex sentences from the meanings of their components. This is perhaps the essential feature of a non-ambiguous context free language, the meanings of the sentences can be discovered from the meanings of their components by applying one of a finite set of rules. The structure of a derivation tree reflects this, components are descendants of the parent construct.

It is often said that the structure or 'shape' of a derivation tree gives the semantics of a sentence, but it is necessary to be careful here. The semantics which can be obtained is also a function of the tree walker that we use when evaluating the semantics. For example, the tree

```
        S
      / | \
     S  -  I
   / | \   |
  S  -  I  2
  |     |
  I     1
  |
  3
```

is usually thought of as indicating a value of (3-1)-2=0 for the sentence `3-1-2`. In principle there is nothing to stop us defining the result of a leaf node to be its value minus the result of the leaf node to the left. Then, if we use a right-most walk of the tree we can evaluate the nodes and end up with a result of 3-(1-2) = 4. The sense in which the above tree 'gives left associativity' to subtraction is on the assumption that the result of a node will be a function of the results of its descendants and its siblings, and that a depth-first tree walker will be used for the evaluation.

2.2 Attribute evaluation

Calculations can be carried out at any stage during the derivation tree construction; such a process is often called *attribute evaluation* and the objects whose 'values' are calculated are called *attributes*. In a recursive descent based parser, each time a symbol becomes the current symbol a function associated with that symbol is called. Using normal function parameter passing and return mechanisms it is possible to pass the values of attributes already calculated up and down the derivation tree as it is constructed. A node in the derivation tree becomes the current node when it is created and again after each of the subtrees corresponding to its children has been constructed. Thus any attribute of a node can depend on any attributes of its descendants and of nodes to the left of that node in the tree, and also on any attributes of ancestor nodes which have already been calculated when the node is created. Thus calculations, usually called *semantic actions* in this context, can be placed at the being, end, and between any two symbols in a production and such actions will be executed at the node corresponding to the LHS of the production rule.

The aim is to design the grammar and hence the derivation trees, so that the semantic evaluation (translation) of the parser input can be carried out as the derivation tree is constructed. If we assume that the tree *construction* will proceed from left to right, either using a 'top-down' parser and translation scheme as outlined in the previous paragraph, or by placing semantic rules at the end of production rules and using a 'bottom-up' parser, then a grammar which generates derivation tree of the form

```
        S
      / | \
     S  -  I
   / | \   |
  S  -  I  3
  |     |
  I     1
  |
  4
```

can be thought of as giving '-' left associative semantics.

We shall assume from now on that the intention when using trees for semantic evaluation is that the meaning of a construct, and hence the result of evaluating its node, is dependent only on the results of evaluating its children, siblings and parent, i.e. the dependency is local in the tree.

There is an analogy with maps here. In planning a route, deciding which turning should be taken next is usually a function of where we have just come from and where we have to get to next. Maps come in different levels of resolution, planning a route from one city to another can often be done using a map with lower resolution, it is not necessary to show all the roads and all their names. In some sense derivation trees are like ordinance survey maps, they are necessary for reconstructing a full derivations but usually contain too much detail for efficient semantic evaluation. In the next section we consider a method of producing lower resolution *reduced* derivation trees by leaving out some nodes and promoting others.

2.3 Changing the shape of the trees

Because of the limitations of particular parsing techniques it may not be possible to use a grammar which gives the required structure to the resulting derivation trees. For example, the obvious way to ensure that the derivation trees for expressions involving subtraction have the 'left associative' structure is to use a left recursive grammar

```
S ::= S - INTEGER | INTEGER
```

however, such a grammar cannot be parsed correctly using a standard recursive descent based parser. We shall look at an approach which allows derivation trees to be reordered so that 'left associative' trees can be produced from right recursive grammars.

In [JS98a] we look at another approach which allows 'balanced binary' subtrees to be constructed for associative operators such as '+'. This will enable the language designer to ensure, for example, that 1-2-3-4 has semantics (((1-2)-3)-4 and that 1+2+3+4+5+6 has semantics ((1+2)+3)+(4+(5+6)).

These modifications are not simply reductions in resolution, they are redrawings of the 'map'. Both redrawings are carried out locally as the parse tree is being constructed. We shall see that in the first case this is efficient and no rebuilding of the tree is required, and in the second case, although some tree rebuilding is carried out, making the modifications at parse time allows the user to over-ride the associative semantics and impose a particular order of calculation.

3 Reducing the resolution in parse trees

The derivation tree may contain more information than is needed to allow the semantics to be determined, it may be of higher than necessary *resolution*. The language may contain more symbols than is necessary to convey the required meaning, to make things easier for the user, and the grammar usually contains more structure than is necessary to define the syntax of the language, to allow the language to be described compositionally and to resolve possible ambiguities and hierarchies.

3.1 Unnecessary terminals

The language may contain 'syntactic sugar' to make it easier for humans to use – the opening parenthesis in a while construct `while( ...` is redundant.

3.2 Unnecessary non-terminals

The grammar may also contain extra non-terminals and rules either to ensure that the derivation trees are of particular shapes or to allow a particular parsing technique to be used. For example, with the grammar

```
S ::= S + S | a
```

no particular shape of derivation tree can be guaranteed. Either of the trees

```
     S                                S
   / | \                            / | \
  S  +   S                       S   +  S
  |    / | \                   / | \    |
  a   S  +  S                 S  +  S   a
      |     |                 |     |
      a     a                 a     a
```

may be produced. With the grammar

```
S ::= a + S | a
```

the derivation trees are guaranteed to 'grow' to the right

```
    S
  / | \
 a  +  S
     / | \
    a  +  S
          |
          a
```

but the grammar is not left factored and so cannot be parsed using a standard recursive descent parser. The grammar

```
S ::= a T
T ::= + S | epsilon
```

admits a recursive descent parser but at the cost of increasing the size and depth of the derivation trees.

```
  S
 / \
a   T
   / \
  +   S
     / \
    a   T
       / \
      +   S
         / \
        a   T
            |
          epsilon
```

The additional non-terminals, and in general any non-terminal node in the tree, are just intermediate stages which are not *necessary* to allow the semantics of a sentence to be constructed but allow the language to be defined compositionally.

3.3 Terminals that provide structure

Another issue is the fact that sentences are 'flat' so some extra syntax is used to distinguish the components. Parenthesis, punctuation and keywords can all be used in this way. For example, statements are often separated by semi-colons to allow us to determine where one statement ends and the next begins and the **else** keyword in a Pascal **if** statement separates two possible alternative statements.

```
if a<b then a:=b; b:=c; else c:=a;
if a<b then a:=b; b:=c; c:=a;
```

Once a derivation tree is constructed these separating components are not needed because the two dimensional nature of a tree allows the components to be distinguished.

```
                  S
  /   |     |     |       |      \
if    B   then    S     else      S
    / | \      /  |  \          / | \
   a  <  b   S    ;   S        S  ;  S
            /|\     / | \     /|\    |
           a:= b   S  ;  S   c:= a  epsilon
                  /|\    |
                 b:= c  epsilon
```

The keyword **if** indicates which rule should be used to evaluate the node and the keywords **then** and **else** are not needed. The lower resolution tree

```
            if
   /         |         \
  B          S          S
 /|\       /   \       / \
a < b     S     S     S  epsilon
         /|\   / \   /|\
        a:= b S   S b:= c
             /|\  |
            b:= c epsilon
```

contains all the required information, as does the even lower resolution tree

```
            if
   /         |         \
  <          ;          :=
 / \       /   \       / \
a   b     :=    :=     b   c
         / \   / \
        a   b b   c
```

Structure is also needed in arithmetic expressions to allow their order of evaluation to be determined. Parentheses are used to identify the operands of an operator, for example `3-(1-2)`

```
   S
 / | \
S  -  I
|   / | \
I  (  S  )
|   / | \
3  S  -  I
   |     |
   I     2
   |
   1
```

but once the tree is constructed the operator/operand relationships are recorded in the subtree structure.

In the rest of this report we shall consider the generation of lower resolution *reduced trees* in which some of the unnecessary information is omitted.

4 Folding

We informally define a reduced derivation tree (RDT) as a tree obtained from a derivation tree by applying one or more *edge folds*. The idea is to remove an unnecessary node by folding the edge which connects it to its parent. An edge to be folded connects two nodes, the *source* and the *destination*, and we indicate an edge to be folded as an arrow.

$$\begin{array}{c} \text{source} \\ \downarrow \\ \text{destination} \end{array} \qquad\qquad \begin{array}{c} \text{destination} \\ \uparrow \\ \text{source} \end{array}$$

When the edge is folded, the source node replaces the destination node by removing the folded edge, attaching the 'in-edge' of the destination node to the source node, inserting the destination node's children to the source node and then deleting the destination node. The node orderings are preserved: the left (right) siblings of the source node become left (right) siblings of the source node's children. The two possibilities are on the left and right in the figure below, in either case the result is the tree in the middle.

```
     parent              parent                  parent
       |                   |                       |
   destination   ->      source        <-     ___source___
  /    |     \         /   |    \            /     |      \
left source right   left children right   left destination right
       |                                           |
     children                                   children
```

4.1 Convergent Folds

Real RDT's are the result of applying many edge folds to the derivation tree. In many cases, the order in which the folds are applied does not affect the final tree, but there is one case which we call a *convergent folding scheme* in which the order of application is important. Where a node acts as both the source in one fold and the destination in another, then two resultant trees are possible. Consider a chain of nodes A-B-C-D in which A is the parent, B is a child of A, C is a child of B and D is a child of C, and two folds (source B to destination C) and (source D to destination C).

$$\begin{array}{c} A \\ | \\ B \\ \downarrow \\ C \\ \uparrow \\ D \end{array}$$

The two folds *converge* on the node C. If we apply the down fold first (from B to C) then the final tree is

```
A
|
D
```

If we apply the up fold first (from D to C) then the final tree is

```
A
|
B
```

Our syntax directed definitions described below allow individual control over the order of evaluation. Our promotion annotations described in the next section and implemented in our **rdp** [JS98b] tool apply the rule that folds are executed in a top-down left-to-right order (giving the first of the possibilities above) because this is a natural implementation for a top-down parser.

4.2 Promoting nodes

We mark to be folded certain edges in the example from the previous section

```
        S
   /    ↑   \
 S      —    I
 ↑      ↙  ↑  ↘
 I      (   S   )
 ↑        / ↑ \
 3      S   —  I
        ↑      ↑
        I      2
        ↑
        1
```

These folds remove the now redundant parenthesis node, promote 'only' children on to their parents, and promote the - nodes.

Applying these folds gives

```
      S                    S                  S
  /   ↑   \            /   ↑   \          /   ↑   \
 S    —    I          I    —    S        3    —    S
 ↑         ↑          ↑       / ↑ \              / ↑ \
 I         S          3      I  —  2            1  —  2
 ↑       / ↑ \               ↑
 3     S   —  I              1
       ↑      ↑
       I      2
       ↑
       1
```

and finally generates

```
      -
    /   \
   4     -
       /   \
      1     3
```

from which it is still possible to construct the semantics using a left-most depth-first tree walk.

The idea is to remove unnecessary bits of the derivation tree as the tree is constructed, producing a less complex tree which still allows the semantics of the sentence to be determined.

5 Annotations Based on Promotion Operators

Edge folding is a convenient notion to use in a graphical environment when one is presented with a fully constructed tree to manipulate, but does not sit easily with the definition of languages using BNF grammars in which the shape of final tree is implicit in the grammar. We shall now give a description of the use of a grammar decorated with *promotion operators* to indicate that the edge joining that terminal or non-terminal is to be folded with its parent. We think of parsing in terms of constructing a derivation tree and we think of promotion operators as changing this construction method to effectively 'hoist' a node up into its parent's position, dragging its children with it. If a symbol has a promotion operator attached then the edge which attaches the corresponding node to its parent is to be folded. A single promotion, `x^`, indicates that the node corresponding to `x` is the destination of the fold. A double promotion, `x^^`, indicates that the node corresponding to `x` is the source of the fold. We refer to the former case as a 'promote-under' and to the latter case as a 'promote-over'.

Later we shall formally describe how to use syntax-directed definitions and then translation schemes to generate a reduced form of the derivation tree. It is possible to describe the effects of the promotion operators using either a top-down or a bottom-up model of tree construction, and we shall do both formally in later sections. Both of these construction methods require a derivation of the input to be supplied.

The definition of an RDT is in terms of a grammar and a derivation, and is independent of any particular parsing technique. But our ultimate goal is to do the tree construction at the same time as the derivation is constructed, i.e. during the parse, and it is natural to use bottom-up tree construction in conjunction with bottom-up parsing techniques and top-down tree construction with top-down parsers. Since standard bottom-up parsing techniques can be applied to a wider range of grammars, we shall describe tree construction as a bottom-up process first. The equivalent top-down descriptions are given in Section 9.

We assume that we have, or are constructing, a derivation of a sentence u from a BNF grammar, some of whose productions rules contain symbols which have been annotated with single or double *promotion operators*, `^` and `^^`. For each step in the derivation, starting from the right hand end, we construct a tree corresponding to the production rule processed at that step. The root of the tree corresponds to the LHS of the production rule, and its children are trees which correspond to the symbols on the RHS of the rule. The tree constructed at the left-most, start symbol, derivation step is the *reduced derivation tree* (RDT) corresponding to u defined by this annotated grammar.

The reader can imagine that we have a standard shift/reduce parser which reads the input from left to right, attempting to recognise a handle, at which point the corresponding rule is processed. As this processing is carried out, the corresponding part of the RDT is constructed. At each stage, trees corresponding the non-terminal symbols on the RHS of the production rule will already exist. These will be joined together as subtrees of a new tree.

5.1 Initial example

Consider the following grammar, in which `int` is a terminal whose pattern is the set of integers,

```
S ::= E + S   | E
E ::= int * E   | int
```

The input `int * int + int` generates the following (reverse) derivation

```
int*int+int <= int*E1+int <= E2+int <= E2+E3 <= E2+S1 <= S
```

Each derivation step corresponds to a step in the parse where a rule is processed (i.e. a reduction step). We show the tree constructed each such step.

```
rule                    tree
----                    ----
E1::=int                  E
                          |
                         int

E2::=int*E1               E
                        / | \
                     int *  E
                            |
                           int

E3::=int                  E
                          |
                         int

S1::=E3                   S
                          |
                          E
                          |
                         int

S::=E2+S1                 S
                        / | \
                       E  +  S
                     / | \   |
                  int *  E   E
                         |   |
                        int int
```

We can annotate the above grammar to get

```
S ::= E +^^ S   | E^
E ::= int *^^ E   | int^^
```

in which a ^ causes the corresponding root node to be deleted when the tree is constructed, and a ^^ causes the corresponding root node to be promoted. For example, The input `int * int + int` and derivation above will then generate the following trees

```
rule                    tree
----                    ----
E1::=int                 int:2

E2::=int*E1               *:2
                         / \
                       int int

E3::=int                 int:2

S1::=E3                  int:2

S::=E2+S1                 +:2
                         / \
                        *  int
                       / \
                     int int
```

Notice that effectively the single promotions have been performed before the double ones, thus the `^^` on `int` overrides the `^` on `E` in the step `S1::=E3`. To achieve this result we attach an integer to the root node of each subtree which records whether there is a double promotion to be carried up from lower in the tree.

5.2 Generating RDTs from annotated BNF grammars

We now give a general procedure for constructing an RDT from an annotated grammar and a (reverse) derivation.

We assume that we have a derivation of an input string, or at least that we have a tail (right-most portion) of a derivation that we are constructing, and that at the current (or most recently constructed) step in the derivation the production rule $X ::= x1^{\nu 1}...xn^{\nu n}$, where each νi is a sequence of 0,1, or 2 promotion operators, has been used to replace `X`

```
u <= ... <= y1...x1..xn...ym <= y1...X...ym <= ...
```

We also assume that we have already constructed subtrees associated with each `xi` that is a non-terminal.

We construct a subtree associated with this instance of the non-terminal `X` by creating a root node, initially labelled `X:0`, and considering each of the symbols `x1,...,xn` in turn.

Suppose that we have already considered `x1,...,xi-1` and the tree being constructed is currently of the form

```
root:m
/  \
......
```

If `xi` is a terminal, and if νi is empty then create a node labelled `xi` as the right-most current child of the root node.

```
      root:m
     /  | \
    .....  xi
```

If $\nu i =$^ do nothing. If $\nu i =$^^ then relabel the root node `xi:2`.

```
       xi:2
      /  \
     ......
```

If `xi` is a non-terminal then we already have a tree

```
       y:r
      / \
    b1...bm
```

If νi is empty add this tree to the current tree by making its root the right-most current child of root node.

```
      root:m
     /  | \
    .....  y
          / \
        b1...bm
```

Otherwise, merge the two root nodes, making the children of subtree the right-most children of the current tree, preserving their relative ordering. If $\nu i =$^ then leave the root node as it is

```
       root:m
    /   | |   \
  ..... b1..bn
```

If $\nu i =$^^ then relabel the root node `y:r`

```
         y:r
    /   | |   \
  ..... b1..bn
```

5.3 Limitations of the promotion operator notation

In Section 6 we shall give formal equivalents to the promotion operators using syntax-directed definitions. This formal description is more cumbersome, but it allows the language designer more control over the shape of RDTs. In this section we discuss some limitations of the promotion operator notation.

5.3.1 Example – promotion priorities

We have defined the tree construction process from annotated grammars in such a way that if a node at the bottom of a chain of promoted nodes has a double promotion, then this node dominates in the folding process. For example, the production rules

```
E ::=  C^ | +^^ C^     C ::= D^^    D ::= int
```

and the derivation fragment

```
E => C => D => a
```

generate a subtree of the form

```
 D
 |
int
```

However, it may be that the language designer would prefer the subtree

```
 E
 |
int
```

to be generated in this case. This effect could be achieved either by changing the tree construction process or by changing the annotation of the grammar. If we change the tree construction process then it will no longer be possible to get the first behaviour, so the language designer is now constrained in an equivalent way. If we change the annotation so that `D^^` becomes `D^`, this will affect other uses of the production `C::=D`, for example, the derivation fragment

```
E => +C => +D => +a
```

would then also generate the first subtree below rather than the second

```
 +           D
 |           |
int         int
```

Syntax-directed definitions provide the flexibility to generate different promotion behaviours for different alternates in a production rule.

5.3.2 Example – conditional promotions

Another limitation of the annotated grammar approach is that it is not possible to 'optionally' promote a node depending on the promotion outcomes of the node's siblings. This conditional promotion can arise in situations where we have a production rule which contains only non-terminals, which may typically have been introduced to 'left-factor' a grammar. For example, the annotated grammar (in which **eps** is a terminal whose pattern contains just the empty string)

```
S ::= E D^
D ::= -^^ S | eps^
E ::= int^^   | (^ S^^ )^
```

and the derivation

```
int-int <= E1-int <= E1-E2 <= E1-E2D1 <= E1-S1 <= E1D2 <= S
```

generate the RDT

```
      -
     / \
   int  S
        |
       int
```

To remove the extra `S` node from the tree we need to promote the node corresponding to `E` in the case where `D` is expanded using the rule `D::=eps`. This conditionality can be expressed in the syntax-directed definitions described below. For an annotated grammar the 'solution', and the approach used in the implementation of the promotion operators in **rdp**, is to promote the empty, `eps`, node and then have a separate empty-node removal routine which removes empty nodes from the RDT after its construction is complete. In this case, the annotated grammar

```
S ::= E D^
D ::= -^^ S | eps^^
E ::= int^^   | (^ S^^ )^
```

and the derivation above generate the RDT

```
      -
     / \
   int eps
        |
       int
```

5.3.3 Example – redundancy

We conclude this discussion on limitations by remarking that there is also some redundancy in the promotion operator notation, and correspondingly in the syntax-directed definitions. The grammars

```
S ::= N^ =^^ T X^
N ::= a
T ::= I^ E^^
E ::= +^^ T | eps^^
I ::= int
X ::= Y^
Y ::= ; | .

S ::= N^^ =^^ T X^
N ::= a
T ::= I^ E^^
E ::= +^^ T | eps^^
I ::= int
X ::= Y^
Y ::= ; | .
```

```
S ::= N^ =^^ T X
N ::= a
T ::= I^ E^^
E ::= +^^ T | eps^^
I ::= int
X ::= Y^
Y ::= ;^^ | .^^
```

```
S ::= N^ =^^ T X^
N ::= a
T ::= I^ E^^
E ::= +^^ T | eps^^
I ::= int
X ::= Y
Y ::= ;^^ | .^^
```

all produce the same RDT for a given input string, although some of the subtrees used in the construction will be different in each case. On input string `a = int+int+int;` the following tree will be generated

```
       =
     / | \
    a  +   ;
      / \
    int   +
         / \
       int  eps
             |
            int
```

This is because the order of intermediate folds does not affect the final outcome except at convergent points. If the bottom node in a chain is the source of fold then in doesn't matter whether its parent is the source or destination of a fold above, the final outcome is that the bottom node dominates. The last three grammars above generate the following three folding sequences, all of which have the same final outcome.

```
S ⟼ S ⟼ S        S ⟼ S ⟼ S        S ⟼ S ⟼ S
↓    ↓    |        |    |    |        ↓    |    |
X    Y    ;        X    X    ;        X    Y    ;
↓    |             ↓    ↑             |    ↑
Y    ;             Y    ;             Y    ;
|                  ↑                  ↑
;                  ;                  ;
```

It is also worth noting that the grammar

```
S ::= N^ =^^ T X^
N ::= a
T ::= I^ E^^
E ::= +^^ T | eps^^
I ::= int
X ::= Y^
Y ::= ;^^ | .^^
```

does not produce the same trees as the others above; the double promotion operators on the symbols ; and . override the one on =, changing the name of the root node.

```
        ;
      /   \
     a      +
          /   \
        int     +
              /   \
            int   eps
                   |
                  int
```

6 Formal description of RDTs using syntax-directed definitions

To address the limitation issues discussed in Section 5.3, and to provide a formal definition of RDTs in terms of established theoretical principles, we now consider tree construction for languages defined by a particular form of syntax-directed definition [ASU86].

Syntax-directed definitions are grammars that have semantic rules at the end of the production rules which are used to calculate the values of attributes of symbols in the production rule. It is usual for the semantic rules to be applied after the corresponding production rule has been processed by the parser, thus it is natural for a tree builder whose input is a syntax-directed definition to build the tree from the leaves up. In our syntax-directed definitions each non-terminal, `X`, has a synthesised attribute, `X.tree` which holds (the label of) the subtree corresponding to particular instance of `X`.

We want to be able to label the nodes in the RDT, so each non-terminal `X` has a synthesised attribute `X.name`. (We shall always label a node corresponding to a terminal, `a`, with the letter `"a"` but if the node were required to have a different label then an attribute `a.name` could be used.)

We don't have to explicitly distinguish between the concepts of single and double promotions because the new root node is labelled using the `name` attribute. However, consider the annotated grammar

```
S ::= a^^ X^            X::= d^^ | h
```

We expect the root node of an RDT for `ad` to be labelled `"d"` not `"a"`, the double promotion on `d` overriding the single promotion on `X`. To achieve this, when determining `Y.name` in a production rule of the form `Y::=...X...`, the promotion value of `X` which is derived from the symbols on the RHS of a production rule needs to be passed back up the grammar. To allow this, non-terminals have a synthesised attribute `X .tprom`.

As each subtree is added to the new tree we have to decide whether to make the root of the subtree a child of the new root or whether to merge the new and old roots together. We might expect each symbol `x` to have an attribute `x.del` which is 1 if the root of the corresponding subtree is to be merged with the root of the tree being constructed, and which is 0 otherwise. However, to decide the value of the `name` attribute we would first have to check whether `x.del=1`, and if so, then whether `x.tprom=2`. This process can be carried out more compactly by defining an attribute whose value is 0 in cases where `x.del=0`, and whose value is the maximum of `x.del` and `x.tprom` otherwise. Thus we don't actually have an attribute `x.del`, in fact each grammar symbol, `x`, has an inherited attribute, `x.prom`, whose value is set to be 0, 2, or the maximum of `x.tprom` and 1. (This corresponds to having 0, 2, or 1 promotion operators on `x`.) If `x.prom`$>$ 0 then there is a fold between `x` and its parent. If `x.prom=1` the parent is the source node, if `x.prom=2` then parent is the destination node and in this case the parent's `name` attribute is updated.

Finally, we have a function `build()` which takes as input the symbols `x1,...,xn` of a given grammar rule `X::=x1...xn`, the attribute `X.name`, the values of each `xi.prom` and `xi.tree`, if `xi` is a non-terminal, and which returns (the label of) a new tree which is the subtree corresponding to `X`. Rather than listing this full set of attributes, we indicate the dependency via the associated production rule, so we write `build(X::=x1...xn)`.

6.1 Example

The syntax-directed definition

```
S ::= E+S1   [* E.prom=0  +.prom=2  S1.prom=0  S.tprom=2
                S.name="+"  S.tree=build(S::=E+S1) *]
      | E    [* E.prom=max(E.tprom,1)  S.tprom=E.prom
                S.name=E.name  S.tree=build(S::=E) *]

E ::= int*E1 [* int.prom=0  *.prom=2  E1.prom=0  E.tprom=2
                E.name="*"  E.tree=build(E::=int*E1) *]
      | int  [* int.prom=2  E.tprom=2  E.name="int"
                E.tree=build(E::=int) *]
```

essentially corresponds to the annotated grammar

```
S ::= E +^^ S  | E^
E ::= int *^^ E  | int^^
```

The input `int * int + int` generates the following (reverse) derivation

```
int*int+int <= int*E1+int <= E2+int <= E2+E3 <= E2+S1 <= S
```

Each derivation step corresponds to a step in the parse where a rule is processed (i.e. a reduction step). We show the synthesised attributes, including the tree, constructed at such step in the parse.

```
E1::=int        E1.tprom=2
                E1.name ="int"
                E1.tree = int

E2::=int*E1     E2.tprom=0
                E2.name ="*"
                E2.tree = *
                         / \
                       int int

E3::=int        E3.tprom=2
                E3.name ="int"
                E3.tree = int

S1::=E3         S1.tprom=2
                S1.name ="int"
                S1.tree = int

S::=E2+S1       S.tprom =2
                S.name  ="+"
```

```
S.tree  = +
           / \
          *  int
         / \
       int int
```

6.2 General form of syntax-directed definitions for RDTs

We now describe how to write semantic rules to define the attributes `prom`, `tprom` and `name`.

For a terminal `a`, to remove the corresponding node from the RDT set `a.prom` =1 or, to overwrite the parent node, set `a.prom`=2. Otherwise set `a.prom`=0.

For a non-terminal, `X`, to remove the corresponding node from the RDT set `X.prom=max(X.tprom,1)`, or, to overwrite the parent node, set `X.prom`=2. Otherwise set `X.prom`=0. Set `X.tprom` to be the maximum value of the `prom` values on the RHS of its production rule and set `X.name` to be the `name` of the right-most symbol with `prom`=2, if one exists, or else set it to be `"X"`.

```
X::=x1 ... xn [* x1.prom = k1 or max(x1.tprom,1)
                 ...
                 xn.prom = kn or max(xn.tprom,1)
                 X.tprom = max(x1.prom,...,xn.prom)
                 X.name  = if(xn.prom=2){xn.name or "xn"} else
                             ...
                             if(x1.prom=2){x1.name or "x1"} else "X"
                 X.tree  = build(X::=x1...xn) *]
```

Note: to guarantee that the attributes associated with a rule in a cycle-free syntax-directed definition can be evaluated immediately after that rule has been processed these attributes must be defined only in terms of synthesized attributes. This is not the case for the syntax-directed definitions we use to define RDTs. In fact, these definitions will not in general be L-attributed either. The attributes `X.tprom` and `X.tree` are defined in terms of the inherited attributes `xi.prom`, and `xi.prom` may depend on the attribute `xi.tprom`. However, this is not actually a problem because the values of the attributes `xi.tprom` are known when it is necessary to calculate `xi.prom`, `X.tprom` and `X.tree`.

6.3 The function `build()`

The function `build()` takes as input a string, a sequence of grammar symbols, a sequence of integers, and a sequence of trees. The intention is that there will be an associated production rule, `X::=x1...xn` say, and that the string will be the value of `X.name`, the symbols will be `x1,...,xn`, the integers will be the values of `x1.prom`, ... `xn.prom` and that the trees will be the trees `xi.tree` for each `xi` which is a non-terminal. Rather than listing this set of parameters, though, we write `build(X::=x1...xn)` to indicate the dependency.

In Section 8 we shall use a slightly extended syntax-directed definition in which the attributes `X.name` can have a special value `"delete"`. We shall describe `build()` allowing for this special value, so that it is available when we eventually need it.

`build(X::=x1...xn)` starts by setting `X.tree` to be a single, root, node labelled `X.name` and then considers the symbols `x1,...,xn` in turn.

If `xi` is a terminal, if `xi.prom`=0 create a node labelled `xi` as the right-most current child of the root node of `X.tree`. Otherwise, do nothing.

```
          X.name
          /  | \
         .....  xi
```

If `xi` is a non-terminal then we already have a tree

```
xi.tree  =  xi.name
             /  \
           b1 .. bm
```

If `xi.prom`=0 and `xi.name` is not `"delete"`, add this tree to `X.tree` by making its root the right-most current child of root node of `X.tree`.

```
          X.name
          /  | \
         ..... xi.name
                /  \
              b1 .. bm
```

Otherwise, merge the two root nodes, making the children of `xi.name` the right-most children of the root of `X.tree`, preserving their relative ordering.

```
          X.name
        /    |  |   \
       .....  b1 .. bn
```

6.4 Example

The syntax-directed definition

```
S ::= aXb [* a.prom=0  X.prom=max(X.tprom,1)  b.prom=0
             S.tprom = if(X.prom=2){2}else{0}
             S.name = if(X.prom=2){X.name}else{"S"}
             S.tree = build(S::=aXb)  *]
    | Xg  [* g.prom=0  X.prom=0  S.tprom=0
             S.name = "S"  S.tree=build(S::=Xg) *]

X ::= cYe [* c.prom=0  Y.prom=max(Y.tprom,1)  e.prom=0
             X.tprom=if(Y.prom=2){2}else{0}
             X.name=if(Y.prom=2){Y.name}else{"X"}
             X.tree=build(X::=cYe) *]
```

```
Y ::= d   [* d.prom=2  Y.tprom=2  Y.name="d"  Y.tree=build(Y::=d) *]
    | h   [* h.prom=0  Y.tprom=0  Y.name="Y"  Y.tree=build(Y::=h) *]
```

corresponds to

```
S ::= a X^ b  |  X g
X ::= c Y^ e
Y ::= d^^ | h
```

The input `a c d e b` generates the (right-most, reverse) derivation

```
a c d e b <= a c Y1 e b <= a X1 e <= S
```

and the corresponding attributes at each step are:

```
Y1::= d        d.prom=2         Y1.tree= d
               Y1.tprom=2
               Y1.name="d"

X1::= c Y1 e   c.prom=0         X1.tree= d
               Y1.prom=2                / \
               e.prom=0                c   e
               X1.tprom=2
               X1.name="d"

S ::= a X1 b   a.prom=0         S.tree = d
               X1.prom=2               / /\ \
               b.prom=0               a c  e b
               S.tprom=2
               S.name="d"
```

The input `c h e g` generates the derivation

```
c h e g <= c Y1 e g <= X1 g <= S
```

and the corresponding attributes are

```
Y1::= d        h.prom=0         Y1.tree= Y
               Y1.tprom=0                |
               Y1.name="Y"               h

X1::= c Y1 e   c.prom=0         X1.tree= X
               Y1.prom=1                /|\
               e.prom=0                c h e
               X1.tprom=2
               X1.name="X"

S ::= X1 g     X1.prom=0        S.tree = S
               g.prom=0                 / \
               b.prom=0                X   g
               S.tprom=0              /|\
               S.name="S"            c h e
```

6.5 Overcoming limitations of annotated grammars

Syntax-directed definitions are more powerful than the annotated grammars. Each annotated grammar has a corresponding syntax-directed definition which generates the same RDTs (see Section 12). In this section we give syntax-directed definitions which overcome the limitations discussed in Examples 5.3.1 and 5.3.2 above.

6.5.1 Example – promotion priorities revisited

If we wish a single promotion to take priority over double promotions lower in the tree we can set the `prom` attribute to be 1. For example, the syntax-directed definition

```
E ::= C   [* C.prom=1  E.tprom=0  E.name="E"  E.tree=build(E::=C)
    | +C  [* +.prom=2  C.prom=max(C.tprom,1)  E.tprom=2
             E.name=if(C.prom=2){C.name}else{"+"}
             E.tree=build(E::=+C)

C ::= D   [* D.prom=2  C.tprom=2  C.name=D.name
             C.tree=build(C::=D)   *]

D ::= int [* int.prom=0  D.tprom=0  D.name="D"
             D.tree=build(D::=int)  *]
```

and the derivations

```
int <= D <= C <= E    +int <= +D <= +C <= E
```

generate trees of the form

```
   E                  D
   |                  |
  int                int
```

respectively. In this first case the double promotion is not passed up the tree but in the second case it is.

6.5.2 Example – conditional promotions revisited

It is possible, with a syntax-directed definition, to 'optionally' promote a node corresponding to x depending on the promotion outcomes of its siblings by using an if statement to define `x.prom`. This allows us to remove nodes introduced, for example, to 'left-factor' a grammar without modifying the RDT once it has been constructed.

The syntax-directed definition

```
S ::= ED  [* D.prom=max(D.tprom,1)
             E.prom=if(D.prom=2){0}else{E.tprom}
             S.tprom=if(D.prom=2 or E.prom=2){2}else{0}
             S.name=if(D.prom=2){D.name}else
```

```
                       if(E.prom=2){E.name}else"S"
             S.tree=build(S::=ED)  *]

D ::= -S  [* -.prom=2  S.prom=0  D.tprom=2  D.name="-"
             D.tree=build(D::=-S)  *]
    | eps [* eps.prom=1  D.tprom=0  D.name="D"
             D.tree=build(D::=eps) *]

E ::= int [* int.prom=2  E.tprom=2  E.name="int"
             E.tree=build(E::=int) *]
    | (S) [* (.prom=1  S.prom=2  ).prom=1  E.tprom=2
             E.name=S.name  E.tree=build(E::=(S)) *]
```

and the derivation

```
int-int <= E1-int <= E1-E2 <= E1-E2D1 <= E1-S1 <= E1D2 <= S
```

generate the RDT

```
       -
      / \
    int int
```

6.6 Using promotion operators and syntax-directed definitions

The scheme that we have outlined for building RDTs based on syntax-directed definitions can be made less cumbersome if we use promotion operators. We annotate the symbols on the RHS of a production rules with a promotion operator ^ which replaces the attribute `prom` in the syntax-directed definition. A single operator ^ on a symbol `x` is equivalent to setting `x.prom` to be greater than 0 and no operator is equivalent to setting `x.prom` to 0. The attribute `tprom` is still needed to determine `X.name`, but now its value will either be 0 or 2.

The function `build()` is essentially unchanged, except that we substitute the phrase "if x has no promotion operator" for the phrase "if `x.prom`=0".

6.6.1 Example

The annotated syntax-directed definition

```
S ::= b S1^ c^ X^ [* S.tprom=2  S.tree=build(S::=bS1^c^x^)
                     S.name=if(X.tprom=2){X.name}else{"c"} *]
    | e S1^ X^    [* S.tprom=if(X.tprom=2 or S.tprom=2){2}else{0}
                     S.name=if(X.tprom=2){X.name}else{
                              if(S1.tprom=2){S1.name}else{"S"}}
                     S.tree=build(S::=eS1^X^) *]
    | g           [* S.tprom=0  S.name="S"  S.tree=build(S::=g) *]
```

```
X ::= d^          [* X.tprom=2  X.name="d"  X.tree=build(X::=d) *]
    | h           [* X.tprom=0  X.name="X"  X.tree=build(X::=h) *]
```

is equivalent to the syntax-directed definition

```
S ::= bS1cX [* b.prom=0 S1.prom=max(S1.tprom,1)
               X.prom=max(X.tprom,1)  c.prom=2
               S.tprom=2  S.tree=build(S::=bS1cX)
               S.name=if(X.prom=2){X.name}else{"c"} *]
    | eS1X  [* e.prom=0  S1.prom=max(S1.tprom,1)
               X.prom=max(X.tprom,1)  S.tree=build(S::=eS1X)
               S.tprom=if(X.prom=2 or S1.prom=2){2}else{0}
               S.name=if(X.prom=2){X.name}else{
                              if(S1.prom=2){S1.name}else{"S"}} *]
    | g     [* g.prom=0  S.tprom=0
               S.name="S"  S.tree=build(S::=g) *]

X ::= d     [* d.prom=2  X.tprom=2
               X.name="d"  X.tree=build(X::=d) *]
    | h     [* h.prom=0  X.tprom=0
               X.name="X"  X.tree=build(X::=h) *]
```

and to the annotated BNF grammar

```
S ::= b S^ c^^ X^  |  e S^ X^  |  g
X ::= d^^  |  h
```

7 Canonical RDTs for BNF grammars

The underlying idea is that the RHSs of production rules should be thought of as functions whose parameters are the non-terminals and the terminals with a pattern containing more than one element. The other terminals on the RHS are considered to be the 'name' of the rule.

To produce a *canonical RDT* a standard non-annotated BNF grammar is taken. This is then preprocessed using an algorithm described below to produce a syntax-directed definition of the type described in Section 6.2.

7.1 Algorithm for generating a syntax-directed definition from a BNF grammar

Consider each of the original production rules `X::=`γ in turn. Essentially semantic rules are defined so that if a node has a single child then this child is promoted over its parent, otherwise a parent is given a specified name and all of its children that correspond to terminals with a single lexeme are deleted. All of these promotions will be of the same type, the child is promoted over the parent, so we shall not need the attribute `X.tprom`.

1. If γ is the empty string or a single terminal, `X::=a`, define semantic rules for this production rule that set `a.prom` to be 2 and `X.name` to be `"a"`:

```
X ::= a [* a.prom=2  X.name="a"  X.tree=build(X::=a) *]
```

2. If γ is a single non-terminal, `X::=Y`, then define semantic rules that set `Y.prom` to be 2 and `X.name` to be `Y.name`:

```
X ::= Y [* Y.prom=2  X.name=Y.name   X.tree=build(X::=Y) *]
```

3. If γ contains two or more symbols `X::=x1...xk`, then define semantic rules to set `xi.prom=2` if `xi` is a terminal with only one lexeme, to set `x1.prom=0` otherwise, and to set `X.name` to be the concatenation of the terminal symbols whose pattern contains only one element, or `"X"` if there are no such non-terminals.

7.2 Example

If we begin with the grammar

```
S ::= E+S | E
E ::= T*E | T
T ::= int
```

where `int` is a terminal whose pattern is the set of integers, and apply the above algorithm we get

```
S ::= E+S1 [* E.prom=0  +.prom=2  S1.prom=0
              S.name="+"  S.tree=build(S::=E+S) *]
    | E    [* E.prom=2  S.name=E.name  S.tree=build(S::=E) *]

E ::= T*E1 [* T.prom=0  *.prom=2  E1.prom=0
              E.name="*"  E.tree=build(E::=T*E) *]
    | T    [* T.prom=2  E.name=T.name  E.tree=build(E::=T) *]

T ::= int  [* int.prom=2  T.name="int"  T.tree=build(T::=int) *]
```

Using this annotated syntax-directed definition and input string `int+int*int+int` generates the derivation

```
int+int*int+int<= T1+int*int+int <= E1+int*int+int <= E1+T2*int+int
               <= E1*T2*T3+int <= E1+T2*E2+int <= E1+E3+int
               <= E1+E3+T4 <= E1+E3+E4 <= E1+E3+S1 <= E1+S2 <= S
```

and the following sequence of corresponding trees, culminating in the canonical RDT.

```
T1.tree = int   E1.tree = int   T2.tree = int   T3.tree = int

E2.tree = int   E3.tree = *     T4.tree = int   E4.tree = int
                         / \
                       int int

S1.tree = int   S2.tree = +
                         / \
                        *  int
                       / \
                     int int

Canonical RDT:   S.tree = +
                         / \
                       int  +
                           / \
                          *  int
                         / \
                       int int
```

7.3 Example

If we begin with the following grammar, in which `ID` is a terminal whose pattern is the set of C-style identifiers,

```
S ::= A B | X ID
B ::= ; | G
G ::= .
A ::= C + A | C
C ::= D E
E ::= * C | eps
```

```
D ::= F | - F
F ::= ID | ( A )
X ::= A O A
O ::= < | <= | ==
```

and apply the above algorithm we get

```
S ::= A B  [* A.prom=0  B.prom=0  S.name="S"
              S.tree=build(S::=AB) *]
    | X ID [* X.prom=0  ID.prom=0  S.name="S"
              S.tree=build(S::=X ID) *]

B ::= ;    [* ;.prom=2  B.name=";"  B.tree=build(B::=;) *]
    | G    [* G.prom=2  B.name=G.name  B.tree=build(B::=G) *]

G ::= .    [* ..prom=2  G.name="."  G.tree=build(G::=.) *]

A ::= C+A1 [* C.prom=0  +.prom=2  A1.prom=0
              A.name="+"  A.tree=build(A::=C+A) *]
    | C    [* C.prom=2  A.name=C.name  A.tree=build(A::=C) *]

C ::= D E  [* D.prom=0  E.prom=0  C.name="C"
              C.tree=build(C::=DE) *]

E ::= *C   [* *.prom=2  C.prom=0  E.name="*"
              E.tree=build(E::=*C) *]
    | eps  [* eps.prom=2  E.name="eps"  E.tree=build(E::=eps) *]

D ::= F    [* F.prom=2  D.name=F.name  D.tree=build(D::=F) *]
    | -F   [* -.prom=2  F.prom=0  D.name="-"
              D.tree=build(D::=-F) *]

F ::= ID   [* ID.prom=2  F.name="ID"  F.tree=build(F::=ID) *]
    | (A)  [* (.prom=2  A.prom=0  ).prom=2
              F.name="(_)"  F.tree=build(F::=(A)) *]

X ::= A1OA2[* A1.prom=0  O.prom=0  A2.prom=0
              X.name="X"  X.tree=build(X::=AOA) *]

O ::= <    [* <.prom=2  O.name="<"  O.tree=build(O::=<) *]
    | <=   [* <=.prom=2  O.name="<="  O.tree=build(O::=<=) *]
    | ==   [* ==.prom=2  O.name="=="  O.tree=build(O::===) *]
```

Using this translation scheme to parse the string

```
ID * ID + ID * ( ID + ID ) ;
```

generates the canonical RDT

```
         S
       /   \
      +     ;
   /     \
  C       C
 / \     / \
ID  *   ID  *
    |        |
    C        C
   / \      / \
 ID  eps  (_) eps
           |
           +
         /   \
       C       C
      / \     / \
     ID  eps ID  eps
```

7.4 Remarks about the canonical form

The canonical RDT produced in Example 7.3 is close to the abstract syntax tree that a human might produce. The additional changes a human would probably make would be to somehow remove the `C`, `S`, `;`, and `(_)` nodes.

The C nodes correspond to the production `C::=DE` which doesn't contain any terminals, reflecting the fact that it doesn't really correspond to any syntactic part of the language being defined. It is purely there to 'left-factor' the grammar, necessary if a standard recursive descent based parser is to be used. The 'obvious' thing to do here is put a promotion operator on one of the two non-terminals, causing the `C` nodes to be overwritten. In this case we need `E^` which would result in the following tree

```
           S
         /    \
        +      ;
      /    \
    *        *
  /   \     /  \
ID    eps ID   eps
       |        |
       ID      (_)
                |
                +
             /    \
           eps    eps
            |      |
            ID     ID
```

The problem is that for other productions, for example ones that have been 'right factored', it could be that the first non-terminal is the one that should be promoted, and it is not possible to tell, simply from the original BNF, which non-terminal to promote. Thus in the canonical form we have been forced to leave in the essentially redundant intermediate node.

Similar remarks apply to the `S` node.

The `(_)` node arises from the production `F::=(A)`. Here again a human would simply add a promotion operator to the non-terminal node, `F::=(^A^)^`, which would cause the following tree to be constructed

```
            S
          /   \
         +      ;
      /     \
    C        C
  /   \    /   \
ID     *  ID     *
       |        |
       C        C
      / \      / \
    ID  eps   +   eps
            /   \
          C      C
         / \    / \
        ID eps ID eps
```

This would be correct because the parentheses in the original production rule were redundant once the tree structure has been established. The problem is that any algorithmic rule which inserted such a promotion operator would also insert one in the production `D::=-F` giving `D::=-^F^`. In this case the intermediate - node would be lost. Not all terminals with single element patterns play the same role in the grammar; some, such as parentheses, are there for structural reasons, whereas others, such as -, are really unary operators. Differences such as these cannot be determined from the BNF and hence cannot be incorporated in an algorithm which simply analyses the BNF.

The solution is to have a canonical RDT which is conservative in the sense that it retains redundant nodes in certain cases, and to allow humans, as we have done above, to construct non-canonical RDTs directly by specifying a syntax-directed definition.

A more general, but perhaps more user daunting approach, is to have an annotation algorithm which takes a partially annotated grammar and completes the annotation. In this case a user could specify the values of **`X.name`** and **`x.prom`** where they see fit and the algorithm would add missing names and promotions according to the scheme describe above. We now describe a slightly different approach in which the user supplies an edited BNF grammar in which some terminal symbols are followed by an exclamation mark, indicating that they should be ignored in the tree construction process.

8 Canonical RDTs from edited BNF grammars

The user provides a BNF grammar in which some of the symbols have been marked with a trailing exclamation mark, indicating that the symbol is not needed for semantic analysis and should be ignored. We call such grammars *edited BNF grammars*. For example, we might place an exclamation mark after a statement separating semicolon, `;!`, and after parentheses, `(!` and `)!` in Example 7.3. We do not expect users to add exclamation marks to non-terminal symbols, but the algorithm specifies the behaviour which results in this case for completeness.

As before, the grammar is preprocessed and turned in to a syntax-directed definition. However, initially symbols with a trailing exclamation mark are ignored, even to the extent that a production rule whose RHS has only one symbol with no exclamation mark is treated as though it is a unit rule. Once the grammar has been processed the exclamation marks are removed.

8.1 Algorithm for annotating an edited BNF grammar

Consider each of the production rules $X ::= \gamma$ in turn. Essentially the grammar is annotated so that if a node has only one child without an exclamation mark then this child is promoted over its parent, if the node has no children without exclamation marks then the node is removed, otherwise a parent is given a specified name and all its children that correspond to terminals with only one lexeme are deleted.

1. If γ is the empty string or a single terminal apart from exclamation marked symbols, `X ::= x1! ...a xi! ...xn!`, then set `X.name` to be `"a"` and set all the `prom` attributes to be 2.

```
X ::= x1..a..xn [*a.prom=2  x1.prom=2 ... xn.prom=2
                 X.name="a"  X.tree=build(X::=x1..a..xn)*]
```

2. If γ is the a single non-terminal apart from exclamation marked symbols, `X ::= x1! ...Y xi! ...xn!`, then set `X.name` to be `Y.name`, and set all the `prom` attributes to be 2.

```
X ::= x1..Y..xn [*Y.prom=2 x1.prom=2 ... xn.prom=2
                 X.name=Y.name  X.tree=build(X::=x1..Y..xn)*]
```

3. If γ contains only exclamation marked symbols, `X ::= x1! ...xn!`, then set `X.name` to be `"delete"`, and set all the `prom` attributes to be 2.

```
X ::= x1...xn [* x1.prom=2 ... xn.prom=2  X.name="delete"
                X.tree=build(X::=x1...xn) *]
```

4. If γ contains two or more non-exclamation marked symbols, $X ::= x_1 \dots x_k$, then annotate this production rule by setting `X.name` to be the concatenation of the non-exclamation marked terminal symbols whose pattern

contains only one element, or "X" if there are no such non-terminals, set `xi.prom=2` if `xi` has a trailing exclamation mark, or if `xi` is a terminal with only one lexeme, and set `xi.prom=0` otherwise.

8.2 Example

If we begin with the edited grammar

```
S ::= A B | X ID
B ::= ;! | G
G ::= .!
A ::= C + A | C
C ::= D E
E ::= * C | eps!
D ::= F | - F
F ::= ID | (! A )!
X ::= A O A
O ::= < | <= | ==
```

and apply the above annotating algorithm we get

```
S ::= A B  [* A.prom=0  B.prom=0
              S.name="S"  S.tree=build(S::=AB) *]
    | X ID [* X.prom=0  ID.prom=0
              S.name="S" S.tree=build(S::=X ID) *]

B ::= ;    [* ;.prom=2 B.name="delete" B.tree=build(B::=;) *]
    | G    [* G.prom=2 B.name=G.name B.tree=build(B::=G) *]

G ::= .    [* ..prom=2 G.name="delete" G.tree=build(G::=.) *]

A ::= C+A1 [* C.prom=0  +.prom=2  A1.prom=0
              A.name="+"  A.tree=build(A::=C+A) *]
    | C    [* C.prom=2 A.name=C.name A.tree=build(A::=C) *]

C ::= D E  [* D.prom=0  E.prom=0
              C.name="C"  C.tree=build(C::=DE) *]

E ::= *C   [* *.prom=2  C.prom=0
              E.name="*"  E.tree=build(E::=*C) *]
    | eps  [* eps.prom=2  E.name="delete"
              E.tree=build(E::=eps) *]

D ::= F    [* F.prom=2  D.name=F.name  D.tree=build(D::=F) *]
    | -F   [* -.prom=2  F.prom=0
              D.name="-"  D.tree=build(D::=-F) *]

F ::= ID   [* ID.prom=2  F.name="ID"  F.tree=build(F::=ID) *]
```

```
    | (A)  [* (.prom=2  A.prom=2  ).prom=2
              F.name=A.name  F.tree=build(F::=(A)) *]

X ::= A1OA2[* A1.prom=0  O.prom=0  A2.prom=0
              X.name="X"  X.tree=build(X::=AOA) *]

O ::= <     [* <.prom=2  O.name="<"  O.tree=build(O::=<) *]
    | <=    [* <=.prom=2  O.name="<="  O.tree=build(O::=<=) *]
    | ==    [* ==.prom=2  O.name="=="  O.tree=build(O::===) *]
```

Using this syntax-direct definition and with input

```
ID * ID + ID * (ID + ID) ;
```

results in the following tree

```
     S
     |
     +
   /   \
  C     C
 / \   / \
ID  * ID  *
    |     |
    C     C
    |     |
   ID     +
         / \
        C   C
        |   |
       ID  ID
```

9 Building trees as the input is read

In the methods we have described so far, the RDT is constructed as the processing of rules is completed. If a grammar is suitable for deterministic top-down parsing the tree can be built as the input symbols are read. We shall now describe the use of translation schemes to build RDTs top down, assuming that the grammar rules are processed as they are encountered, in the traditional recursive descent style. We 'traverse' the grammar, beginning with the start symbol, constructing a left-most derivation of the input string. At each stage in the construction there is a *current production*, a *current symbol* in that production and a *current node* in the tree being constructed. If the current symbol is a terminal then a node for that symbol is created as the right-most child of the current node. If the current symbol is a non-terminal then a production for that symbol is chosen, the new production becomes the current production and the first symbol on its RHS becomes the current symbol. A new node is created as a child of the current node, and this node becomes the current node.

9.1 Example

```
S ::= I | S - I
I ::= 1 | 2 | 3 | (S)
```

This grammar is not LL(1) so cannot be parsed using a standard predictive recursive descent parser, but there is a left-most derivation

```
S => S-I => I-I => 3-I => 3-(S) => 3-(S-I) => 3-(I-I)
  => 3-(1-I) => 3-(1-2)
```

and at each step in this derivation there is a current symbol and current production. The corresponding tree construction steps are

```
S  S  S   S    S     S         S         S         S         S
  /  /   /   / |   / | \     / | \     / | \     / | \     / | \
 S  S   S   S  -  S  -  I   S  -  I   S  -  I   S  -  I   S  -  I
    |   |   |     |         |    /    |    /|   |    /|   |    /|
    I   I   I     I         I   (     I   ( S   I   ( S   I   ( S
        |   |     |         |         |         |    /    |    /
        3   3     3         3         3         3   S     3   S
                                                              |
                                                              I

   S           S           S             S             S
 / | \       / | \       / | \         / | \         / | \
S  -  I     S  -  I     S  -  I       S  -  I       S  -  I
|    /|     |    /|     |    /|       |    /|       |    /|\
I   ( S     I   ( S     I   ( S       I   ( S       I   ( S )
|    /      |    /|     |    /|\      |    /|\      |    /|\
3   S       3   S -     3   S - I     3   S - I     3   S - I
    |           |           |             |   |         |   |
    I           I           I             I   2         I   2
    |           |           |             |             |
    1           1           1             1             1
```

Using a translation scheme we can direct the tree constructing process so that some nodes are not put in the tree, effectively executing edge folds on the fly. By passing a variable into and out of each production rule call, we can continually update the names of nodes which have been constructed and by either using or ignoring a name passed up we can achieve the effects of both double and single promotion operators.

9.2 Using translation schemes and a single promotion operator to build TDRs

Syntax-directed definitions lend themselves to classical bottom-up parsing techniques. However, translation schemes are perhaps easier to write and to understand, and RDTs can be constructed from translation schemes as the input symbols are read. The disadvantage is that if a translation scheme is to be implemented using a standard deterministic parser, the grammar needs to be LL(1). Of course, if we are *given* a left-most derivation, then we can use the translation scheme to build the parse tree by following the derivation steps. So translation schemes can be used to define canonical RDTs for any non-ambiguous grammar, and we envisage the use of a 'back-tracking' top-down parser which first finds the left-most derivation and then re-runs this derivation to build the RDT.

A translation scheme is a modification of a syntax-directed definition in which semantic rules for evaluating attributes can be placed anywhere in the RHS of a production rule. In our case, each non-terminal symbol can take and return a string parameter. The input string will be the current name of the current parent node, and the return string will contain the current name of the node associated with the symbol just processed.

When a terminal symbol with no promotion operator is dealt with, a new node labelled with the name of the terminal is created. If the terminal has a promotion operator then no new node is created but a following semantic action can reassign the current name of the parent node to be the name of the terminal (i.e. mimicking a double promotion operator).

When a non-terminal symbol with no promotion operator is dealt with a new node is created with current name the name of the non-terminal. This is then passed down into the production rule for that non-terminal. When the production rule has been processed a string is returned which is the final name for this node.

When a non-terminal symbol `Y^` with a promotion operator is dealt with no new node is created but the current name of the node associated with the LHS of the production is passed down into the production rule for `Y`. The value ultimately returned by the production rule for `Y` is assigned the current name of the parent node.

In the following we shall assume that the value of the attribute called `name` is the value that is returned when a production rule has been processed, and that the name of the variable in which that value is returned is specified after a colon following the corresponding symbol in the grammar.

9.3 Example

The annotated translation scheme

```
S(name) ::= a X^(name):name b | X("X") g
X(name) ::= c Y^(name):name e
Y(name) ::= d^   [* name = "d" *]   | h
```

corresponds to the syntax-directed definition in Example 6.4. For the production `Y::=h` the value of `name` is unaltered and the input value is returned. For the production `S::=aXb` the value returned by `X` is the value returned by this production rule.

Given the input `c h e g`, this translation scheme generates the derivation

```
S => X1 g => c Y1 e g => c h e g
```

and the corresponding tree construction steps are

```
 S    S      S      S        S        S         S
     /      /      /        /        /         / \
   X1     X1     X1       X1       X1         X   g
         /      /        / |      / | \      /|\
        c      c        c  h     c  h  e    c h e
```

9.4 Example

The translation scheme

```
S(name) ::= b S^(name) c^ [* name = "c" *] X^(name):name
          | e S^(name):name X^(name):name | g

X(name) ::= d^ [* name = "d" *] | h
```

corresponds to the syntax-directed definitions in Example 6.6.1.

9.5 Algorithm for tree construction when parsing with an annotated translation scheme

The parse begins with the start production, which is called with input `name="S"` and correspondingly the root node of the tree is created, initially labelled with the start symbol. At each stage in the parse there is a *current production*, a *current symbol* in that production, and a *current node*. Initially these are the start production, `S::=x1...xn` say, the symbol `x1`, and the root node respectively. It will always be the case that the current symbol appears in the current production and that the current node corresponds to the LHS of the current production.

1. If the current symbol is a non-terminal, `E(fj)` say, with no promotion operator then a production for `E` is chosen, `E::=y1...yk` say, and a new node initially labelled with the instance of this non-terminal, `Em` say, is created as a child of the current node, currently labelled `fi` say. Then

the symbol `y1` becomes the current symbol, `E::=y1...yk` becomes the current production, and the newly constructed node becomes the current node.

```
      .      current production   E ::= y1...yk
     .           current symbol   y1
    fi             current node   Em
   / |
... Em
```

2. If the current symbol is a non-terminal with a promotion operator, `E^(fj)` say, then a production, `E::=y1...yk` say, is chosen, the symbol `y1` becomes the current symbol, `E::=y1...yk` becomes the current production, and the current node is unchanged.

```
      .      current production   E ::= y1...yk
     .           current symbol   y1
    fi             current node   fi
   /
...
```

3. If the current symbol, `a`, is the empty string or a terminal with no promotion operator then we have a current production of the form `B ::= y1... yj a...yk` and a current node labelled `fi`. In this case a new node labelled `a` is created as a child of the current node.

```
     .       current production   get_next_symbol.prod
    .            current symbol   get_next_symbol.sym
   fi              current node   get_next_symbol.node
  / | \
....  a
```

Then a routine is called to get the next symbol, which will reassign the current symbol and may change the current node and production. We think of `get_next_symbol` as returning a symbol, `get_next_symbol.sym`, a node, `get_next_symbol.node`, and a production rule, `get_next_symbol.prod`. The routine `get_next_symbol` simply makes the current symbol the next symbol in the current production rule, if there is one, and leaves the current node and production unchanged. If current symbol is the last symbol in the current rule, or the empty symbol, then

- ◇ the current node is re-named to be the value of `name` calculated and returned at the end of the current production
- ◇ `get_next_symbol.sym`, `get_next_symbol.node` and `get_next_symbol.prod` are set back to be the current symbol, node and production, respectively, at the time the current production was called, and `get_next_symbol` is called again.
- ◇ If `get_next_symbol.sym` is last symbol of the start production then the root node is labelled with the value returned by the start production, and the tree construction process is terminated.

4. If the current symbol, `a^`, is the empty string or terminal with a promotion operator, then `get_next_symbol` is called immediately. So no new nodes are created in the tree but a new current symbol, and possibly new current node and production rule, are selected, and some nodes maybe re-labelled.

```
      .        current production   get_next_symbol.prod
     .             current symbol   get_next_symbol.sym
    fi               current node   get_next_symbol.node
  / |
....
```

5. If the translation scheme being used has been generated from an edited BNF grammar, see Section 10.1, then some of the nodes may be labelled `"delete"`. In this case, when the tree construction is complete, all nodes labelled `delete` are removed from the tree and their children, if they have any, become children of the deleted node's parent, at the position where the deleted node was and in their existing order.

9.6 Example

The translation scheme

```
S(name) ::= E("E") D^(name):name

D(name) ::= -^ [* name="-" *] S("S")
          | eps^ [* name="eps" *]

E(name) ::= int^ [* name="int" *]  | (^ S^(name):name )^
```

is equivalent to the second annotated grammar in Example 5.3.2.

Using this translation scheme and the algorithm in Section 9.5 on the input string `int - int - int` results in the following construction sequence

```
S          current production   S ::= E1 D1^
               current symbol   E1
                 current node   S

S          current production   E1 ::= int^
|              current symbol   int^
E1               current node   E1
```

Now, the current symbol `int` is the last in the current rule, `name` gets the value `"int"` which is used to label the current node, `E1`. Then the current node and production return to the previous instances, and the next symbol, `D1`, is got.

```
 S        current production   S ::= E1 D1^
 |            current symbol   D1^
int             current node   S

 S        current production   D1 ::= -^ S1("S")
 |            current symbol   -^
int             current node   S
```

```
   S           current production   D1 ::= -^ S1("S")
   |               current symbol   S1
  int                current node   S

   S           current production   S1 ::= E2("E") D2^("S")
  / \              current symbol   E2
int  S1              current node   S1

   S           current production   E2 ::= int^
  / \              current symbol   int^
int  S1              current node   E2
     |
     E2

   S           current production   S1 ::= E2("E") D2^("S")
  / \              current symbol   D2^
int  S1              current node   S1
     |
    int

   S           current production   D2 ::= -^ S2("S")
  / \              current symbol   -^
int  S1              current node   S1
     |
    int

   S           current production   D2 ::= -^ S2("S")
  / \              current symbol   S2
int  S1              current node   S1
     |
    int

   S           current production   S2 ::= E3("E") D3^("S")
  / \              current symbol   E3
int  S1              current node   S2
    / \
  int  S2

   S           current production   E3 ::= int^
  / \              current symbol   int^
int  S1              current node   E3
    / \
  int  S2
       |
       E3
```

```
    S          current production   S2 ::= E3("E") D3^("S")
   / \             current symbol   D3^
 int  S1             current node   S2
     / \
   int  S2
         |
        int

    S          current production   D3 ::= eps^
   / \             current symbol   eps^
 int  S1             current node   S2
     / \
   int  S2
         |
        int

    S          current production   S ::= E1("E") D1^("S")
   / \             current symbol   D1
 int  -              current node   S
     / \
   int eps
         |
        int
```

The symbol returned by `get_next_symbol` is `D1`, the last symbol in the start production, so we re-label the root node with the returned value of **`name`**, which in this case is `-`, and the tree construction process is complete.

```
    -
   / \
 int  -
     / \
   int eps
         |
        int
```

10 Canonical RDTs via translation schemes

In this section we give an algorithm which turns an edited BNF grammar in to a translation scheme with promotion operators in such a way that, if the grammar in non-ambiguous, the trees produced are the canonical RDTs defined in Section 7. We only describe the process for an edited grammar, but an edited grammar with no exclamation marks is just a grammar, so the following description includes non-edited grammars as a special case.

As for the bottom-up construction, an edited BNF grammar is preprocessed using the algorithm described below to produce an annotated translation scheme in which a semantic rule assigns a name to the LHS of each production, and in which some of the symbols have single associated promotion operators. Because in all the promotions the child is promoted over the parent, we don't need to pass the current node names down the tree. Thus our translation scheme uses only return values (synthesised attributes).

10.1 Algorithm to generate an annotated translation scheme from an edited BNF grammar

Consider each of the production rules `X::=`γ in turn.

1. If γ is the empty string or a single terminal apart from exclamation marked symbols, `X ::= x1!...a xi!...xn!`, then annotate this production rule to return `"a"`, by setting `name` to be `"a"`, and put a single promotion operator on all the symbols:

   ```
   X ::= x1^ ... a^ [* name="a" *] xi^ ... xn^
   ```

2. If γ is the a single non-terminal apart from exclamation marked symbols, `X ::= x1!...Y xi!...xn!`, then annotate this production rule by returning the value returned by `Y`, and put a single promotion operator on all the symbols:

   ```
   X ::= x1^ ... Y^:name xi^ ... xn^
   ```

3. If γ contains only exclamation marked symbols, `X::=x1! ... xn!`, then return `"delete"`, and put a single promotion operator on all the symbols:

   ```
   X ::= x1^ ... xn^ [* name="delete" *]
   ```

4. If γ contains two or more symbols non-exclamation marked symbols, `X::=x1...xk`, then set `name` to be the concatenation of the non-exclamation marked terminal symbols whose pattern contains only one element, or `"X"` if there are no such non-terminals, and put a single promotion operator on all the exclamation marked symbols and all the terminal symbols whose pattern contains only one element.

10.2 Example

If we begin with the grammar from Example 7.2

```
S ::= E+S | E
E ::= T*E | T
T ::= int
```

and apply the above translation scheme generating algorithm we get

```
S ::= E +^ S [* name="+" *]  | E^:name
E ::= T *^ E [* name="*" *]  | T^:name
T ::= int^   [* name="int" *]
```

The derivation

```
S => E1+S1 => T1+S1 => int+S1 => int+E2+S2 => int+T2*E3+S2
  => int+int*E3+S2 => int+int*T3+S2 => int+int*int+S2
  => int+int*int+E4 => int+int*int+T4 => int+int*int+int
```

corresponds to the following sequence of trees, culminating in the same canonical RDT as we get with the syntax-directed definition in Section 7.

```
S    S     S     S       S       S       S        S
    /     /     / \     / \     / \     / \      / \
   E1   int   int  S1 int  S1 int  S1 int  S1  int  S1
                          /       /       /        /
                         E2      E2      E2       E2
                                /       /        / \
                               T2     int      int  E3

   S          S           S           +
  / \        / \         / \         / \
int  S1    int  S1     int  +      int  +
    /          / \         / \         / \
   *          *   S2      *  int      *  int
  / \        / \         / \         / \
int int    int int     int int     int int
```

10.3 Example

If we begin with the grammar from Example 8.2 above

```
S ::= A B | X ID
B ::= ;! | G
G ::= .!
A ::= C + A | C
C ::= D E
E ::= * C | eps!
D ::= F | - F
F ::= ID | (! A )!
X ::= A O A
O ::= < | <= | ==
```

and apply the above annotating algorithm we get

```
S ::= A B [* name="S" *]  |  X ID [* name="S" *]
B ::= ;^ [* name="delete" *]  |  G^:name
G ::= .^ [* name="delete" *]
A ::= C +^ A [* name="+" *]  |  C^:name
C ::= D E  [* name="C" *]
E ::= *^ C  [* name="*" *]   |  eps^  [* name="delete" *]
D ::= F^:name  |  -^ F  [* name="-" *]
F ::= ID^  [* name="ID" *]  | (^ A^:name )^
X ::= A O A  [* name="X" *]
O ::= <^  [* name="<" *]  |  <=^  [* name="<=" *]
                          |  ==^  [* name="==" *]
```

Using the above translation scheme, the algorithm in Section 10, and the derivation

```
S => AB => CB => DEB => FEB => ID EB => ID*CB
  => ID*DEB => ID*FEB => ID*(A)EB => ID*(C+A)EB => ID*(DE+A)EB
  => ID*(FE+A)EB => ID*(ID E+A)EB => ID*(ID+A)EB => ID*(ID+C)EB
  => ID*(ID+DE)EB => ID*(ID+FE)EB => ID*(ID+ID E)EB
  => ID*(ID+ID)EB => ID*(ID+ID)B => ID*(ID+ID)G => ID*(ID+ID).
```

results in the following RDT

```
            S
            |
            C
           / \
         ID   *
              |
              C
              |
              +
             / \
            C   C
            |   |
           ID  ID
```

11 Changing the shape of the RDT

Our basic thesis is that language designers should define the grammars so that the trees used as intermediate forms automatically have the 'shape' required for semantic evaluation, and that the promotion operators should be used to remove redundancy from such trees. However, we also advocate that these trees should be built automatically by the parser, and all currently known linear time parsing techniques have restrictions on the grammars to which they can be applied. Thus it is possible that the designer will be unable to write a grammar which generates trees of the required shape if a particular parsing technique is also required. A classic example is the standard recursive descent technique which can not be used on left-recursive grammars.

In this section we shall introduce a third type of promotion operator which can be used to change the local shape of an RDT, and we shall demonstrate how this operator can be used on a right-recursive translation scheme in such a way that left-recursive RDTs are constructed.

11.1 Left associative operators

As we remarked in Section 2, if we are to walk an RDT in left-most depth-first fashion, to ensure left-associative semantics is achieved for operators such a `'-'` the RDT must 'grow to the left'

```
      -
     / \
    -  int
   / \
 int int
```

The natural way to achieve this is to use a left-recursive translation scheme

```
S ::= S -^ [* name="-" *] int  | int^ [* name="int" *]
```

However, standard recursive descent based parsers cannot be based on left-recursive grammars. The same language can be generated by an equivalent right-recursive grammar

```
S ::= int -^ [* name="-" *] S | int^ [* name="int" *]
```

which solves the problem from a parsing point of view, but it doesn't generate the required RDTs. For input `int-int-int` the RDT

```
      -
     / \
   int  -
       / \
     int int
```

is generated.

Another solution is to use an extended BNF which has a Kleene closure operation, often denoted by `{}`, and implement this with iteration semantics. For example,

```
S ::= int {-^ [* name="-" *] int}
```

In this case the RDT generated from `int - int - int` is

```
     -
   / | \
int int int
```

Using a left-most tree walk, it is natural to evaluate the result of this tree in a left-associative fashion. In another report, [JS98a], we discuss a particular extended BNF, which we call *iterator BNF* (IBNF), which contains structures of this form. In that report we discuss the generation of derivation trees for IBNF grammars. However, formally `'-'` is a binary operator, its node in the tree should only have two children, and some applications may require a tree walker which exploits this property, i.e. the designer may require the RDT which is generated naturally by a left-recursive grammar. We now discuss the use of a third promotion operator using which we can construct a left-most tree from a right-recursive grammar. (A description of a neater construction of such a tree from an iterator grammar can be found in [JS98a].)

11.2 Promotion of a child node above its parent

It turns out that we can construct a left recursive RDT from from a right-recursive grammar if we introduce a third type of promotion, *promote above*, in which the edge from a parent to a child is folded backwards so that the child becomes the parent of its former parent and of all its former siblings to the right. The following three trees fold to give the RDTs below in each case.

```
      X                   X                       X
 /  | ⇑ | \          /  | ⇑ | ⇑ \            /  | ⇑ | \
a...b c d..e        a...b c h d..e          a...b c Y..e
                                                    ⇑ \
                                                    f   h

       c                    d                        c
    / | \                  /|\                    /  |   \
   X  d..e                c ..e                  X   f  ..e
  / \                    / \                    / \  |\
 a...b                  X   h                  a...b Y h
                       / \
                      a...b
```

As usual, we define the resultant tree on the basis of applying the folds in depth-first, left-most order.

```
        X                                  d
  /  | ⇑ | ↑ \                            / | \
 a...b c h d..e        becomes            X h..e
                                         / \
                                        a...b
```

11.3 Specifying 'promote above' in a translation scheme

In this discussion we shall revert to the basic definition of translation schemes in which grammar symbols have associated attributes and there are semantic actions placed throughout the production rules which evaluate these attributes [ASU86]. This is because we now need a fourth 'promotion' value. There is an extension of the annotated grammar definition given in Section 5.2 which implements a restricted version of the promote-above operation, and we shall discuss this in Section 12.

Each grammar symbol `x` will have an inherited attribute `x.hat`, whose value is either 0,1, or 3, which determines whether the corresponding node is left as it is, deleted, or promoted above its parent respectively. To make the translation schemes slightly easier to read these attributes will have their values defined at the front of the production rule. (These attributes perform almost the same role as the `prom` attributes used in the syntax-directed definitions above. The difference is that the `prom` attributes depended on the `tprom` attributes which are not known until the associated production rule has been processed. If we used these attributes then we would not be able to put the actions at the front of the production and the values would not be known when the associated node is constructed.)

Each non-terminal symbol, `X`, will also have a synthesised attribute `X.name` which will be used to name the current node (i.e. the node corresponding to `X`) once the production for `X` has been processed, and a synthesised attribute `X.that` which will be used in the calculation of the `name` attribute of its parent node.

Example

```
S ::= [* int.hat=2  D.hat=1 *] int D
      [* S.that=2  S.name=if(D.that=2){D.name}else{"int"} *]

D ::= [* -.hat=3  int.hat=0  D1.hat=1 *] - int D1
      [* D.that=D1.that D.name=if(D1.that=2){D1.name}else{"D"} *]
     | [* eps.hat=1 *] eps
      [* D.that=0  D.name="D" *]
```

We call a translation scheme of this form a *general translation scheme*, and we now give an algorithm for constructing an RDT from such a scheme.

11.4 Algorithm for tree construction from a general translation scheme

This algorithm is a modification of the algorithm given in Section 9.5. The parse begins with the start production and correspondingly the root node of the tree is created, initially labelled with the start symbol. At each stage in the parse there is a *current production*, a *current symbol* in that production, and a *current node*. Initially these are the start production, `S::=x1...xn` say, the symbol `x1`, and the root node respectively. It will always be the case that the current symbol appears in the current production and that the current node corresponds to the LHS of the current production.

1. If the current symbol is a non-terminal, `E` say, and if `E.hat=0` then a production for `E` is chosen, `E::=y1...yk` say, and a new node initially labelled with the instance of this non-terminal, `Em` say, is created as a child of the current node, labelled `fi` say. Then the symbol `y1` becomes the current symbol, `E::=y1...yk` becomes the current production, and the newly constructed node becomes the current node.

```
       .      current production   E ::= y1...yk
      .           current symbol   y1
    fi              current node   Em
  / |
... Em
```

2. If the current symbol is a non-terminal, `E`, and if `E.hat=1`, then a production, `E::=y1...yk` say, is chosen, the symbol `y1` becomes the current symbol, `E::=y1...yk` becomes the current production, and the current node is unchanged.

```
       .      current production   E ::= y1...yk
      .           current symbol   y1
    fi              current node   fi
  /
...
```

3. If the current symbol is a non-terminal, `E`, and if `E.hat=3`, then a production, `E::=y1...yk` say, is chosen and a new node initially labelled with the instance of this non-terminal, `Em` say, is created as the parent of the current node, `fi` say. Then the symbol `y1` becomes the current symbol, `E::=y1...yk` becomes the current production, and the newly constructed node becomes the current node.

```
       .      current production   E ::= y1...yk
      .           current symbol   y1
    Em              current node   Em
    |
    fi
  /
...
```

4. If the current symbol is the empty string or a token, `a` say, with `a.hat=0` then we have a current production of the form `B ::= y1... yj a...yk` and a current node labelled `fi`. In this case a new node labelled `a` is created as a child of the current node.

```
      .       current production   get_next_symbol.prod
    .             current symbol   get_next_symbol.sym
   fi               current node   get_next_symbol.node
 / | \
....  a
```

Then a routine is called to get the next symbol, which will reassign the current symbol and may change the current node and production. We think of `get_next_symbol` as returning a symbol, `get_next_symbol.sym`, a node, `get_next_symbol.node`, and a production rule, `get_next_symbol.prod`.

The routine `get_next_symbol` simply makes the current symbol the next symbol in the current production rule, if there is one, and leaves the current node and production unchanged. If current symbol is the last symbol in the current rule, or the empty symbol, then

- ◇ if there is a node labelled `"Xk"`, where `Xk` is the LHS of the current production, then this node is relabelled to be the value of `Xk.name`.
- ◇ `get_next_symbol.sym`, `get_next_symbol.node` and `get_next_symbol.prod` are set back to be the current symbol, node and production, respectively, at the time the current production was called and `get_next_symbol` is called again.
- ◇ If `get_next_symbol.sym` is last symbol of the start production then the root node is labelled with `S.name` and the tree construction process is terminated.

5. If the current symbol, `a`, is the empty string or a terminal with `a.hat=1` then `get_next_symbol` is called immediately. So no new nodes are created in the tree but a new current symbol, and possibly new current node and production rule, are selected, and some nodes maybe re-labelled.

```
      .        current production   get_next_symbol.prod
     .             current symbol   get_next_symbol.sym
    fi               current node   get_next_symbol.node
  / |
 ....
```

6. If the current symbol, `a`, is the empty string or a terminal with `a.hat=3` then we have a current production of the form `B ::= y1... yj a...yk` and a current node labelled `fi`. In this case a new node labelled `a` is created as the parent of the current node, and this new node becomes the current node. Then `get_next_symbol` is called.

```
      .        current production   get_next_symbol.prod
     .             current symbol   get_next_symbol.sym
    a                current node   get_next_symbol.node
    |
    fi
  / |
 ....
```

11.5 Example

Using the above algorithm with the general translation scheme from Section 11.3, and input `int:1 - int:2 - int:3` we get the following tree construction sequence (here we have distinguished between the different instances of `int` to make the tree structure clearer)

```
   S:0       current production  S ::= int:1 D1
                 current symbol  int:1
                   current node  S

   S:0       current production  S ::= int:1 D1
                 current symbol  D1
                   current node  S

   S:0       current production  D1 ::= - int:2 D2
                 current symbol  -
                   current node  S

   -:1       current production  D1 ::= - int:2 D2
   |             current symbol  int:2
   S               current node  -:1

   -:1       current production  D1 ::= - int:2 D2
  / \            current symbol  D2
 S int:2           current node  -:1

   -:1       current production  D2 ::= - int:3 D3
  / \            current symbol  -
 S int:2           current node  -:1

   -:2       current production  D2 ::= - int:3 D3
   |             current symbol  int:3
   -:1             current node  -:2
  / \
 S int:2

    -:2      current production  D2 ::= - int:3 D3
   / \           current symbol  D3
  -:1 int:3        current node  -:2
 / \
S int:2

    -:2      current production  D3 ::= eps
   / \           current symbol  eps
  -:1 int:3        current node  -:2
 / \
S int:2

    -:2      current production  S ::= int:1 D1
   / \           current symbol  D1
  -:1 int:3        current node  S
 / \
S int:2

      -         final tree
     / \
    -  int:3
  /   \
int:1 int:2
```

11.6 Left-associative operators from right-recursive translation schemes

```
S ::= int D
D ::= - int D  | eps
```

is an LL(1) grammar which generates expressions containing subtractions. On input `int:1 - int:2 - int:3` the following derivation tree is generated

```
          S
       /     \
    int:1    D1
           /   |   \
          -  int:2  D2
                  /  |  \
                 - int:3 D3
                          |
                         eps
```

If we place double folds on the edges connecting the `D` and `-` nodes to their parents, and then apply these folds we get

```
              D3
             / \
            -  eps
          /   \
         D2  int:3
         |
         -
       /   \
      D1  int:2
      |
      S
      |
     int:1
```

This is equivalent to using the translation scheme

```
S ::= [* int.hat=0  D.hat=3 *] int D [* S.that=0  S.name="S" *]

D ::= [* -.hat=3  int.hat=0  D1.hat=3 *] - int D1
      [* D.that=0  D.name="D" *]
    | [* eps.hat=0 *] eps [* D.that=0  D.name="D" *]
```

If alternatively we place single folds of the edges entering `D` and `eps` nodes and the first `int` node, and double folds on `-` nodes, then applying these folds in left-most depth first fashion gives the tree in Section 11.5, and is equivalent to using the translation scheme

```
S ::= [* int.hat=2  D.hat=1 *] int D
      [* S.that=2  S.name=if(D.that=2){D.name}else{"int"} *]

D ::= [* -.hat=3  int.hat=0  D1.hat=1 *] - int D1
      [* D.that=D1.that  D.name=if(D1.that=2){D1.name}else{"D"} *]
    | [* eps.hat=1 *] eps [* D.that=0  D.name="D" *]
```

In order to add extra operators such `*`, we need to address issues of operator priority. For example, we want `2-3*4` to be have the same tree as `2-(3*4)`. To do this with a left-factored non-left recursive grammar we can write

```
S ::= E D
D ::= + S  | - S | eps
E ::= T M
M ::= * E | / E | eps
T ::= int | ( S )
```

which, on input `int:1-(int:2+int:3)*int:4-int:5` generates the tree

```
                 S
         /                 \
       E                    D
     /   \           /         \
    T     M         -           S1
    |     |               /            \
  int:1 eps              E               D
                    /         \        /   \
                   T            M     -     S1
                /  |  \        / \         /   \
               (   S1  )      *   E1      E     D
                 /    \          /   \   /   \   |
               E       D       T     M  T     M eps
             /   \    / \      |     |  |     |
            T     M  +   S1   int:4 eps int:5 eps
            |     |    /   \
          int:2 eps  E      D
                   /   \    |
                  T     M eps
                  |     |
                 int:3 eps
```

If we put single folds on the edges in to the `M`, `D`, `(`, and `)` nodes and the `E1` and `S` nodes, and if we put double folds on the edges in to the `+`,`-`,`*`,`/` nodes, this corresponds to the translation scheme

```
S ::= [* E.hat=0  D.hat=1 *] E D
      [* S.that=D.that  S.name=if(D.that=2){D.name}else{"S"} *]

D ::= [* +.hat=3  S.hat=1 *] + S
      [* D.that=S.that  D.name=if(S.that=2){S.name}else{"D"} *]
   |  [* -.hat=3  S.hat=1 *] - S
      [* D.that=S.that  D.name=if(S.that=2){S.name}else{"D"} *]
   |  [* eps.hat=1 *] eps [* D.that=0  D.name="D" *]

E ::= [* T.hat=0  M.hat=1 *] T M
      [* E.that=M.that  E.name=if(M.that=2){M.name}else{"E"} *]

M ::= [* *.hat=3  E.hat=1 *] * E
```

```
      [* M.that=E.that  M.name=if(E.that=2){E.name}else{"M"} *]
   |  [* /.hat=3  E.hat=1 *] / E
      [* M.that=E.that  M.name=if(E.that=2){E.name}else{"M"} *]
   |  [* eps.hat=1 *] eps [* M.that=0  M.name="M" *]

T ::= [* int.hat=2 *] int [* T.that=2  T.name="int" *]
   |  [* (.hat=1  S.hat=2  ).hat=1 *] ( S )
      [* T.that=2  T.name=S.name *]
```

which on input

```
int:1-(int:2+int:3)*int:4-int:5
```

generates the tree

```
             -
          /     \
         -       E
      /    \     |
     S      *  int:6
     |     / \
     E    E int:4
     |    |
   int:1  +
         / \
        S   E
        |   |
        E int:3
        |
      int:2
```

It is tempting to put folds on all edges in to all the E nodes, i.e. writing the first production rule as

```
S ::= [* E.hat=1  D.hat=1 *] E D
      [* S.that=D.that  S.name=if(D.that=2){D.name} else
                               if(E.that=2){E.name}else{"S"} *]
```

However, in this case the subsequent multiplication and division operators are promoted above the wrong node and end up too high in the tree.

```
               -
             /   \
            *   int:5
          /   \
         -   int:4
       /   \
      S      -
      |     / \
    int:1  S  int:3
           |
         int:2
```

This is because the folds are applied in a left-most depth-first order.

12 rdp's annotation operators

We conclude this report with a brief discussion of a concrete implementation of the promotion ideas discussed above.

rdp [JS98b] is a program which takes an IBNF (an extended form of BNF) specification of a language and, provided the specification is LL(1), generates a recursive descent based parser written in C. In particular, **rdp** can be given an annotated grammar in which some of the symbols have a single, double or triple attached promotion operator, and from this it generates a parser which in turn automatically generates the RDT corresponding to any given input string. **rdp** produces the trees in a representation which is suitable for visualisation using the VCG [San95] tool.

We shall only discuss the BNF inputs to **rdp** in this report, and since **rdp** generates recursive descent based compilers, we shall describe its implementation of the promotion operators in terms of translation schemes not syntax-directed definitions.

12.1 rdp input format

Only the grammar rules are actually input to **rdp**, and each such rule must be terminated by a period.

The following conventions are used by **rdp** to deduce the remaining aspects of the grammar: the left hand side of the first grammar rule is the start symbol, the terminal symbols are enclosed in single quotes, and each grammar rule is terminated by a full stop. In addition, there is only allowed to be one production rule for each non-terminal; all the rules required for a given non-terminal must be combined together using the alternation symbol $|$.

We also note that **rdp** does not directly accept 'epsilon' rules (i.e. rules of the form $A ::= \epsilon$, where ϵ denotes the empty string). The notation `[`α`]` is used to represent $\alpha|\epsilon$. So, for example,

`Y ::= ['+' S].` corresponds to `Y ::= '+' S |` ϵ.

We can annotate the symbols on the RHS of a production rules with promotion operators `^`, `^^` and `^^^`. The behaviour of the operators `^` and `^^` is as described in Section 5, and `^^^` essentially corresponds to the 'promote above' operation described in Section 11.3.

For example, **rdp** will accept and generate a parser for the annotated grammar

```
S ::= E D^ .
D ::= '+'^^^ S^  | [ '-'^^^ S^]:^ .
E ::= T M^ .
M ::= '*'^^^ E^ | [ '/'^^^ E^]:^ .
T ::= INTEGER^^ | '('^ S^^ ')'^ .
```

To put a promotion operator on the ϵ string we append it to the corresponding bracket using a colon. So

`Y ::= ['+' S]:^ .` corresponds to `Y ::= '+' S |` ϵ`^`.

12.2 Promotion operators in terms of general translation schemes

We formally define the tree construction behaviour of `rdp` on input of an annotated grammar in terms by defining a corresponding general translation scheme.

For each production rule in the `rdp` grammar we describe a corresponding translation scheme rule. The union of these rules constitutes a translation scheme corresponding to the original annotated grammar. As we are only considering annotated BNF here we can assume that each production rule in a `rdp` input grammar is of the form

```
X ::= d1 | ... | dn .
```

where each `di` is of one of the forms

$\texttt{x1}_1^{\nu}\ \ldots\ \texttt{xn}_n^{\nu}\qquad [\texttt{x1}_1^{\nu}\ \ldots\ \texttt{xn}_n^{\nu}]\qquad [\texttt{x1}_1^{\nu}\ \ldots\ \texttt{xn}_n^{\nu}]:^{\beta}$

where each `xi` is a grammar symbol, each ν_i is a string of 0,1,2 or 3 promotion operators, and β is a string of 1,2, or 3 promotion operators.

At most one `di` can have brackets, because the grammar must be LL(1). So, if `di=[s]` or `di=[s]`$:^{\beta}$, first we re-write the production rule in the form

```
X ::= d1 | ... | s | ... | dn
    | [* eps.hat=0 *] eps [* X.that=0  X.name="X" *]
```

or, if $\beta =$`^`,

```
X ::= d1 | ... | s | ... | dn
    | [* eps.hat=1 *] eps [* X.that=0  X.name="X" *]
```

or, if $\beta =$`^^`,

```
X ::= d1 | ... | s | ... | dn
    | [* eps.hat=2 *] eps [* X.that=2  X.name="eps" *]
```

or, if $\beta =$`^^^`,

```
X ::= d1 | ... | s | ... | dn
    | [* eps.hat=3 *] eps [* X.that=0  X.name="X" *]
```

Now we can assume that

```
X ::= d1 | ... | dn .
```

where each `di` is of the form

$\texttt{di} = \texttt{x1}_1^{\nu}\ \ldots\ \texttt{xn}_n^{\nu}$

We deal with each production `X::=di` in turn, so that we end up with a rule of the form

```
X ::= [* x1.hat=..   *] x1 ... xn [* X.that=.. X.name=.. *]
```

First we add semantic actions at the front of the production to set the **hat** attributes using the promotion operators ν_j. We set

$$\texttt{xj.hat} = \begin{cases} 0, & \text{if } \nu_j \text{ is empty} \\ 1, & \text{if } \nu_j{=}\wedge \text{ or } \wedge\wedge \\ 3, & \text{if } \nu_j = \wedge\wedge\wedge \end{cases}$$

At the end of the production we add another semantic action to define `X.that` and `X.name`. For notational purposes we define an intermediate value, `x.val`, corresponding to an incidence x^{ν} as follows

$$\texttt{x.val} = \begin{cases} \texttt{x.that}, & \text{if x is a non-terminal and } \nu = \wedge \\ 2, & \text{if } \nu = \wedge\wedge \\ 0, & \text{otherwise.} \end{cases}$$

(The attribute `val` is just used here for definition purposes, it is never written in a translation scheme, where the actual integer or `that` attribute is always used.) We set

```
X.that = max(xi.val)
```

and

```
X.name = if(xn.val=2){xn.name or "xn"} else ...
           if(x1.val=2){x1.name or "x1"} else {"X"}
```

12.3 Example

Applying the above correspondence to the grammar

```
S ::= N^ '='^^ T X^ .
N ::= 'a' .
T ::= I^ E^^ .
E ::= ['+'^^ T ]:^^ .
I ::= 'int' .
X ::= Y^ .
Y ::= ';' | '.' .
```

which is essentially the grammar from Section 5.3.3, results in the following translation scheme

```
S ::= [* N.hat=1  =.hat=1  T.hat=0  X.hat=1 *] N=TX
      [* S.that=2  S.name=if(X.that=2){X.name}else{=} *]

N ::= [* a.hat=0 *] a [* N.that=0  N.name="N" *]

T ::= [* I.hat=1  E.hat=1 *] IE [* T.that=2  T.name=E.name *]

E ::= [* +.hat=1  T.hat=0 *] +T [* E.that=2 E.name="+" *]
   |  [* eps.hat=1 *] eps [* E.that=2 E.name="eps" *]
```

```
I ::= [* int.hat=0 *] int [* I.that=0 I.name="I" *]

X ::= [* Y.hat=1 *] Y
      [* I.that=Y.that  Y.name=if(Y.that=2){Y.name}else{"X"} *]

Y ::= [* ;.hat=0 *] ; [* Y.that=0 Y.name="Y" *]
   |  [* ..hat=0 *] . [* Y.that=0 Y.name="Y" *]
```

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles techniques and tools.* Addison-Wesley, 1986.

[JS98a] Adrian Johnstone and Elizabeth Scott. Iterator bnf. Technical Report TR-98-??, Royal Holloway, University of London, Computer Science Department, November 1998.

[JS98b] Adrian Johnstone and Elizabeth Scott. **rdp** – an iterator based recursive descent parser generator with tree promotion operators. *SIGPLAN notices*, 33(9), September 1998.

[Par96] Terence John Parr. *Language translation using PCCTS and C++.* Automata Publishing Company, 1996.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.