Tree Parsing

$Revision: 1.4 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

This document describes a language for defining tree parsers. Tree parsers can be used to transform, interpret and print tree-structured data. They are particularly useful for problems in which the action at a node depends strongly on the context in which that node appears. Code selection is a common example of this kind of problem: The code selected for an operation is largely determined by that operation's context.

Consider the problem of selecting an instruction to implement an integer addition operation on a typical RISC machine. The machine has two integer add instructions, one taking two register operands and the other taking a register and a constant operand. Both of these instructions leave their result in a register. Load and store instructions also take a register and a constant operand, which are integers added to obtain the memory address. Integer addition is a commutative operation, so the instructions involving constant operands can be used regardless of *which* operand is constant. The code selector must perform a distinct action in each of the five possible situations resulting from these conditions.

The language described in this document allows the user to specify the possible situations and required actions in an intuitive way as a set of pattern/action rules:

```
IntReg ::= Plus(IntReg,IntReg) :  IntRR_add
IntReg ::= Plus(IntReg,IntLit) :: IntRI_add
MemAdr ::= Plus(IntReg,IntLit) :: RI_address
```

Here the first rule specifies that one possible situation is to compute a value into an integer register by adding the contents of two integer registers. The required action in that situation is the `IntRR_add` action. Two situations are specified by the second rule. In one the left operand is in an integer register and the right operand is an integer constant, and in the other the operands are reversed. Both situations can be handled by the `IntRI_add` action, provided that the operands are always presented to it in the order stated (first the register operand, then the constant operand).

In addition to supplying the set of rules as a type-'`tp`' file, a user must make certain that implementations of the actions (like `IntRR_add`, `IntRI_add` and `RI_address` in the above example) are available. The actions might be written by the user or produced by other components of the system like PTG (see Pattern-Based Text Generator).

Actions are arbitrary functions. They may or may not have results, and may or may not have side effects. The effect of the whole process is nothing more than the sum total of the effects of the actions.

To get the specified actions executed, the user must first call functions that are generated from the set of rules. These functions create a tree embodying the contextual relationships among the operators (like `Plus` above). Once the complete tree has been established, another generated function is called to parse it (e.g. to determine whether a given node is an `IntReg` or `IntLit` in the example above). Action routines are invoked as a side effect of the parse.

1 The Tree To Be Parsed

Problems amenable to solution by tree parsing involve hierarchical relationships among entities. Each entity is represented by a node in a tree, and the structure of the tree represents the hierarchical relationship among the entities represented by its nodes.

The relationships are such that nodes corresponding to entities of a particular kind always have the same number of children. No constraint is placed on the *kinds* of children a particular kind of node can have; only the *number* of children is fixed. This tree parser accepts only trees in which each node has no more than two children.

An entity like an integer addition operator is completely characterized by the kind of node representing it. Integer constants, on the other hand, are not completely characterized by the fact that they are represented by `IntDenotation` nodes. Each `IntDenotation` node must therefore carry the constant's value as an *attribute*. This tree parser allows an arbitrary number of attributes of arbitrary type to be attached to each node.

A user builds the tree describing the hierarchical relationships among the entities of interest by invoking specific constructor functions. The constructor used to build a particular node depends on the number of children and the number and type of attributes required by that node.

This section begins by formalizing the structure of a tree to be parsed. It then characterizes the attributes, and finally explains the naming conventions for the constructors.

1.1 Tree Structure

The tree structure is defined in terms of a set of symbols that constitute a *ranked alphabet*: Each symbol has an associated *arity* that determines the number of children a node representing the symbol will have. Each node of the tree represents a symbol of the ranked alphabet, and the number of children of a node is the arity of the symbol it represents. Any such tree is legal; there is no constraint on the symbols represented by the children of a node, only on their number.

The ranked alphabet is extracted from the specification supplied by the user (see Chapter 2 [The Tree Patterns], page 8). The translator verifies that the arity of each symbol is consistent over the specification.

Each symbol of the ranked alphabet denotes a particular kind of entity. For example, here is a set of symbols forming a ranked alphabet that could be the basis of a tree describing simple arithmetic expressions:

```
IntegerVal   FloatingVal   IntegerVar   FloatingVar
Negative
Plus         Minus         Star         Slash
```

The symbols in the first row have arity 0, and are therefore represented by leaves of the tree. `Negative` has arity 1, and the symbols in the third row all have arity 2. Each symbol has the obvious meaning when describing an expression:

'`3.1415`' `FloatingVal`

'`-3`' `Negative(IntegerVal)`

'`k-3`' `Minus(IntegerVar,IntegerVal)`

'`(a*7)/(j+2)`'

`Slash(Star(FloatingVar,IntegerVal),Plus(IntegerVar,IntegerVal))`

The notation here the normal algebraic one: A term is either a symbol of arity 0, or it is a symbol of arity k followed by a parenthesized list of k terms. Each term corresponds to a node of the tree.

A tree describing the expression in the first line has one node, representing the symbol `FloatingVal`. Because `FloatingVal` has arity 0, that node has no children. (The value '`3.1415`' would appear as an attribute of the node, see Section 1.2 [Decorating Nodes], page 5.)

A tree describing the expression in the last line has seven nodes. Four are leaves because the symbols they represent have arity 0; each of the remaining three has two children because the symbol it represents has arity 2.

A tree is not acceptable to the tree parser described in this document if any node has more than two children. Thus no symbol of the ranked alphabet may have arity greater than 2. That is not a significant restriction, since any tree can be represented as a binary tree.

Suppose that we want to use trees to describe the following C expressions:

```
'i>j ? i-j : j-i'
'(i=1, j=3, k=5, l+3) + 7'
```

Although `?:` is usually thought of as a ternary operator, its semantics provide a natural decomposition into a condition and two alternatives:

```
Conditional(
  Greater(IntegerVar,IntegerVar),
  Alternatives(Minus(IntegerVar,IntegerVar),Minus(IntegerVar,IntegerVar)))
```

The comma expression might have any number of components, but they can simply be accumulated from left to right:

```
Plus(
  Comma(
    Comma(
      Comma(Assign(IntegerVar,IntegerVal),Assign(IntegerVar,IntegerVal)),
      Assign(IntegerVar,IntegerVal)),
    Plus(IntegerVar,IntegerVal)),
  IntegerVal)
```

1.2 Decorating Nodes

In addition to its arity, each symbol in the ranked alphabet may be associated with a fixed number of attributes. Each attribute has a specific type. The attributes decorate the nodes of the tree, but they do not contribute any structural information.

In the examples of the previous section, the symbols of arity 0 did not provide all of the necessary information about the leaves. Each symbol of arity 0 specified *what* the leaf was, but not *which* value of that kind it represented. This is often the case with leaves, so a leaf usually has an associated attribute. Interior nodes, on the other hand, seldom need attributes.

Each attribute must be given a value of the proper type when the node corresponding to the symbol is created. This value will not affect the tree parse in any way, but will be passed unchanged to the function implementing the action associated with the rule used in the derivation of the node. Thus attributes are a mechanism for passing information through the tree parse.

1.3 Node Construction Functions

Each node of the tree to be parsed is constructed by invoking a function whose name and parameters depend on the number of children and attributes of the node. The name always begins with the characters `TP_`, followed by the digit representing the number of children. If there are attributes, the attribute types follow. Each attribute type is preceded by an underscore.

The set of constructors is determined from the specification supplied by the user (see Chapter 2 [The Tree Patterns], page 8). The translator verifies that each occurrence of a symbol is consistent with respect to the number of children and types of attributes.

Consider the simple expression trees discussed above (see Section 1.1 [Tree Structure], page 3):

```
IntegerVal    FloatingVal    IntegerVar    FloatingVar
Negative
Plus          Minus          Star          Slash
```

Suppose that integer and floating-point values are represented by the integer indexes of their denotations in the string table (see Section 'Character String Storage' in Library Reference), and variables are represented by definition table keys (see Section 'The Definition Table Module' in Property Definition Language). In that case each tree node representing either `IntegerVal` or `FloatingVal` would be decorated with an `int`-valued attribute; each tree node representing either `IntegerVar` or `FloatingVar` would be decorated with a `DefTableKey`-valued attribute. No other node would have attributes, and four tree construction functions would be created by the translator:

`TPNode` **TP_0_int** (`int` *symbol*, `int` *attr*)
: Return a *symbol* leaf decorated with *attr*, of type `int`

`TPNode` **TP_0_DefTableKey** (`int` *symbol*, `DefTableKey` *attr*)
: Return a *symbol* leaf decorated with *attr*, of type `DefTableKey`

`TPNode` **TP_1** (`int` *symbol*, `TPNode` *child*)
: Return an undecorated *symbol* node with one child

`TPNode` **TP_2** (`int` *symbol*, `TPNode` *left*, `TPNode` *right*)
: Return an undecorated *symbol* node with two children

Here's how the tree describing the expression '`-i+1`' could be constructed:

```
TP_2(
  Plus,
  TP_1(Negative, TP_0_DefTableKey(IntegerVar, keyOfi)),
```

```
TP_0_int(IntegerVal, indexOf1))
```

Here `keyOfi` is a variable holding the definition table key associated with variable `i` and `indexOf1` is a variable holding the string table index of the denotation for `1`.

All tree construction functions return values of type `TPNode`. Attributes can be attached to nodes with children, although there are no such nodes in the example above. Here's the constructor invocation for a node with two children and two integer attributes:

```
TP_2_int_int(Symbol, child1, child2, attr1, attr2);
```

2 The Tree Patterns

The tree patterns describe a set of derivations for trees. They are based on the ranked alphabet of symbols represented by tree nodes and also on a finite set of *nonterminals*. The ranked alphabet and the set of nonterminals are disjoint.

Each nonterminal represents a relevant interpretation of a node. For example, if the tree parser was intended to select machine instructions to implement expression evaluation, the nonterminal `IntReg` might be used to represent the interpretation "an integer value in a register". A derivation could interpret either a leaf describing an integer constant or a node describing an addition operation in that way. Another derivation could interpret the same addition node as "a floating-point value in a register" (possibly represented by the nonterminal `FltReg`).

Each rule characterizes a context in which a specific action is to be performed. For code selection there might be one rule characterizing an integer addition instruction and another characterizing a floating-point addition instruction. An integer addition instruction that required both of its operands to be in registers and delivered its result to a register would be characterized by a rule involving only `IntReg` nonterminals.

Most rules characterize contexts consisting of single tree nodes. Some contexts, however, do not involve any tree nodes at all. Suppose that a node is interpreted as leaving an integer value in a register, and there is an instruction that converts an integer value in a register to a floating-point value in a register. If the original node is the child of a node demanding a floating-point value in a register, the tree parser can supply the implied conversion instruction by using the rule characterizing its context in the derivation.

It is also possible to write a rule characterizing a context consisting of several nodes. Some machines have complex addressing functions that involve summing the contents of two registers and a constant and then accessing a value at the resulting address. In this case, a single rule with a pattern containing two addition operations and placing appropriate interpretations on the operands would characterize the context in which the addressing function action was performed.

The set of patterns is generally ambiguous. In order to disambiguate them, each rule has an associated *cost*. Costs are non-negative integer values, and default to 1 if left unspecified. The tree parser selects the derivation having the lowest total cost. We will ignore the cost in this chapter (see Chapter 4 [Summary of the Specification Language], page 18).

2.1 Rules Describing Tree Nodes

A rule describing a single tree node has the following general form:

```
NO ::= s(Ni,aj)
```

Here `NO` is a nonterminal, `s` an element of the ranked alphabet, `Ni` a (possibly empty) list of nonterminals, and `aj` a (possibly empty) list of attribute types. If one of `Ni` and `aj` is empty then the comma separating them is omitted; if both are empty both the comma and parentheses are omitted.

Recall that trees describing simple arithmetic expressions could be based upon the following ranked alphabet:

```
IntegerVal   FloatingVal   IntegerVar   FloatingVar
Negative
Plus         Minus         Star         Slash
```

Suppose that the tree parser is to select machine instructions that evaluate the expression described by the tree being parsed. Assume that the target machine has a simple RISC architecture, in which all operands must be loaded into registers and every operation leaves its result in a register.

One context relevant to instruction selection is that of an `IntegerVal` leaf. This context corresponds to the selection of an instruction to load an integer constant operand into a register. It could be characterized by the following rule:

```
IntReg ::= IntegerVal(int)
```

This rule describes a single node, and has the form `NO ::= s(a1)`. `NO` is the nonterminal `IntReg`, which places the interpretation "an integer value in a register" on the node. `IntegerVal` is the element `s` of the ranked alphabet. Since `IntegerVal` has arity 0, no nonterminals may appear between the parentheses. As discussed above (see Section 1.2 [Decorating Nodes], page 5), the leaf has a single associated attribute to specify the value it represents. This value is a string table index of type `int`, so the rule contains the type identifier `int`.

Another context related to instruction selection is that of a `Plus` node. This context corresponds to the selection of an instruction to add the contents of two registers, leaving the result in a register. It could be characterized by the following rule:

```
IntReg ::= Plus(IntReg,IntReg)
```

This rule describes a single node, and has the form `N0 ::= s(N1,N2)`. `N0` is the nonterminal `IntReg`, which places the interpretation "an integer value in a register" on the node. `Plus` is the element `s` of the ranked alphabet. Since `Plus` has arity 2, two nonterminals must appear between the parentheses. `IntReg` is the appropriate nonterminal in this case, because it places the interpretation "an integer value in a register" on both children and the machine's integer addition instruction requires both of its operands in registers.

If the target machine had floating-point operations as well as integer operations, a complete set of rules characterizing the relevant contexts in trees describing simple arithmetic expressions might be:

```
IntReg ::= IntegerVal(int)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= Negative(IntReg)
IntReg ::= Plus(IntReg,IntReg)
IntReg ::= Minus(IntReg,IntReg)
IntReg ::= Star(IntReg,IntReg)
IntReg ::= Slash(IntReg,IntReg)

FltReg ::= FloatingVal(int)
FltReg ::= FloatingVar(DefTableKey)
FltReg ::= Negative(FltReg)
FltReg ::= Plus(FltReg,FltReg)
FltReg ::= Minus(FltReg,FltReg)
FltReg ::= Star(FltReg,FltReg)
FltReg ::= Slash(FltReg,FltReg)
```

It is important to remember that the tree to be parsed involves only the nodes representing the symbols of the ranked alphabet (`IntegerVal`, `Plus`, etc.) The tree parser constructs a derivation of that tree in terms of the tree patterns. That derivation consists of applications of the rules, and those rules must be applied consistently with respect to the nonterminals. For example, recall the tree describing '`k-3`':

```
'k-3'      Minus(IntegerVar,IntegerVal)
```

This tree could be derived by applying the following rules:

```
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= IntegerVal(int)
IntReg ::= Minus(IntReg,IntReg)
```

2.2 Chain Rules

A chain rule has the following general form:

```
N0 ::= N1
```

Here `N0` and `N1` are both nonterminals.

A chain rule is used in the derivation of a tree when the interpretation of a node differs from the interpretation required by its parent. It does not describe any tree node, but simply indicates that the difference in interpretations is allowed.

The patterns in the last section (see Section 2.1 [Rules Describing Tree Nodes], page 9) cannot derive the tree for the expression '`k-2.3`':

```
IntReg ::= IntegerVar(DefTableKey)
FltReg ::= FloatingVal(int)
IntReg ::= Minus(IntReg,IntReg)   /* Fails */
FltReg ::= Minus(FltReg,FltReg)   /* Fails also */
```

Both rules describing the `Minus` node demand operands of the same interpretation, and in this tree the operands have different interpretations.

Suppose that it is possible to convert an `IntReg` to a `FltReg` without loss of information. If this is true, then the value of '`k`' could be converted to a floating-point value and the result used as the first child of the `Minus` node. The possibility of such a conversion is indicated by adding the following chain rule to the patterns given in the last section:

```
FltReg ::= IntReg
```

If this chain rule is one of the patterns then the derivation of '`k-2.3`' would be:

```
IntReg ::= IntegerVar(DefTableKey)
FltReg ::= IntReg
FltReg ::= FloatingVal(int)
FltReg ::= Minus(FltReg,FltReg)
```

Now consider the expression '`k-3`' from the last section. With the addition of the chain rule, *two* derivations are possible:

```
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= IntegerVal(int)
IntReg ::= Minus(IntReg,IntReg)

IntReg ::= IntegerVar(DefTableKey)
FltReg ::= IntReg
IntReg ::= IntegerVal(int)
FltReg ::= IntReg
FltReg ::= Minus(FltReg,FltReg)
```

Remember, however, that each rule has an associated cost. That cost defaults to 1 when it isn't specified, so each of the rules in this example has cost 1. The cost of a derivation is simply the sum of the costs of the rules from which it is constituted. Thus the cost of the first derivation above is 3 and the cost of the second is 5. The tree parser always selects the derivation with the lowest cost, so the derivation of '`k-3`' will be the first of the two given.

2.3 Rules Describing Tree Fragments

The right-hand side of a rule describing a tree fragment defines that fragment with nonterminal leaves. Some examples are:

```
N0 ::= s(t(N1),N2)
N0 ::= s(N1,t(N2))
N0 ::= s(t(N1),u(N2))
N0 ::= s(t(s(N1,N2)),N3)
```

Here `N0` is a nonterminal, `s`, `t` and `u` are elements of the ranked alphabet, and `N1`, `N2` and `N3` are nonterminals. No attribute types are allowed in in a rule describing a tree fragment.

Recall the tree used to describe a C conditional expression:

'`i>j ? i-j : j-i`'

```
Conditional(
  Greater(IntegerVar,IntegerVar),
  Alternatives(
    Minus(IntegerVar,IntegerVar),
    Minus(IntegerVar,IntegerVar)))
```

The following rules might be used to describe the tree fragment resulting from the conditional:

```
IntReg ::= Conditional(IntReg,Alternatives(IntReg,IntReg))
FltReg ::= Conditional(IntReg,Alternatives(FltReg,FltReg))
```

If these tree fragment rules (and appropriate rules for `Greater`) are part of the specification then the derivation of '`i>j ? i-j : j-i`' would be:

```
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= Greater(IntReg,IntReg)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= Minus(IntReg,IntReg)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= IntegerVar(DefTableKey)
IntReg ::= Minus(IntReg,IntReg)
IntReg ::= Conditional(IntReg,Alternatives(IntReg,IntReg))
```

Notice that there are no derivation steps corresponding to the components of the tree fragment resulting from the conditional; there is only a single derivation step corresponding to the entire fragment.

3 Actions Carried Out During Parsing

Each rule has an associated action, written as an identifier:

```
N0 ::= s(Ni,aj)    : Action1
N0 ::= N1          : Action2
N0 ::= s(t(Ni),Nj) : Action3
```

The action associated with a rule is carried out each time the rule is used in a derivation. Each rule may be associated with a distinct action, or a single action may be associated with several rules.

3.1 Actions and Values

The action carried out for each use of a rule in a derivation is a function application. The action identifier is the name of the function, and the arguments to which it is applied are the values of the nonterminals and attributes appearing on the right-hand side of the pattern. These values are taken in order from left to right. The result of the function becomes the value of the nonterminal appearing on the left-hand side of the pattern.

For example, consider one of the rules of the specification introduced above (see Section 2.1 [Rules Describing Tree Nodes], page 9), augmented by an action called `IntR_loadconst`:

```
IntReg ::= IntegerVal(int) : IntR_loadconst
```

For each use of the rule `IntReg ::= IntegerVal(int)` in some derivation, the function named `IntR_loadconst` will be applied to the integer-valued attribute of the leaf. The result of this function application will become the value of the `IntReg` nonterminal.

The types of the attribute values are stated explicitly in the rules. A type is also associated with each nonterminal by means of a declaration (see Chapter 4 [Summary of the Specification Language], page 18). For example, the type associated with the nonterminal `IntReg` might be structure of type `reg` defined as follows:

```
typedef struct { int register; PTGNode code; } reg;
```

Here the `register` field would be the number of the register holding the result and the `code` field would be a representation of the assembly language instructions producing the result in that register (see Section 'Introduction' in Pattern-Based Text Generator).

In this case, execution of `IntR_loadconst` would allocate a register and create a PTG node representing the assembly language instruction loading the integer constant specified by the integer-valued attribute of the leaf into that register. It would return a type-`reg` structure containing that information. This structure would then become the value of the `IntReg` nonterminal.

Here's another example of a rule, this time augmented by an action called `IntRR_sub`:

```
IntReg ::= Minus(IntReg,IntReg) : IntRR_sub
```

The function named `IntRR_sub` will be applied to the values returned by the two children of the `Minus` node for each use of `IntReg ::= Minus(IntReg,IntReg)` in some derivation, and the result will become the value of the `IntReg` nonterminal on the left-hand side of the rule. The first argument of `IntRR_sub` would be the value returned by the action associated with the left child of the `Minus` node, and the second would be the value returned by the right child.

Execution of `IntRR_sub` might allocate a register to hold the result of the subtraction and create a PTG node representing the sequence consisting of the PTG nodes passed to it as operands followed by the assembly language instruction that computes the difference of two integer values in registers and leaves the result in a register. `IntRR_sub` would return a type-`reg` structure, which would become the value of the `IntReg` nonterminal on the left-hand side of the rule.

Each nonterminal is associated with a function whose name is `TP_` followed by the name of the nonterminal. This function takes as its only argument a tree (of type `TPNode`, see Section 1.3 [Node Construction Functions], page 5), and returns a value of the type associated with the nonterminal. Whenever one of these functions is applied to the root of a tree, it parses that tree. The parse finds the cheapest derivation of the function's nonterminal at the root of the tree. All of the actions implied by the derivation are executed, and the result of the function is the result delivered by the action executed at the root of the tree. The only guarantee one can make about the order in which the actions are executed is that it respects the data flow constraints implied by the function applications.

A specification with all of the rules described so far has only two nonterminals (`IntReg` and `FltReg`). The translator will generate two parsing functions that can be applied to the root of a tree:

`reg` **TP_IntReg** (`TPNode` *tree*)

A derivation for the tree rooted in *tree* in which the root is interpreted as an `IntReg` will be sought. If such a derivation is possible, the actions associated with the steps for the cheapest will be executed. The result of the action associated with the derivation step at the root will be returned. Otherwise the program will terminate abnormally.

`reg` **TP_FltReg** (`TPNode` *tree*)

A derivation for the tree rooted in *tree* in which the root is interpreted as a `FltReg` will be sought. If such a derivation is possible, the actions associated with the steps for the cheapest will be executed. The result of the action associated with the derivation step at the root will be returned. Otherwise the program will terminate abnormally.

The program will terminate abnormally when a requested derivation is not possible. This condition always arises from a design fault; either the patterns are incomplete, or the tree to be parsed is malformed.

3.2 Implementing Actions

An action is a function application, and the name of the action is the function to be invoked. The rule with which the action is associated determines the signature of the function: Recall that the arguments of the function are the nonterminals and attributes appearing on the right-hand side of the associated rule, in order from left to right. The result of the function becomes the value of the nonterminal appearing on the left-hand side of the associated rule. Each nonterminal and attribute has a fixed type.

Function application can be implemented either by calling a macro or by invoking a routine. If the action requires a routine invocation, and the signature of the routine to be invoked matches the signature determined by the rule, then the routine name can be used directly as the action. Often, however, there is a mismatch between the signatures. In that case, the action can be made the name of a macro that rearranges arguments, inserts constants, or does whatever else is needed to correct the mismatch.

3.3 Commutative Actions

Many computers have instruction sets that are asymmetric in their treatment of operands. For example, a machine with two-operand instructions may allow only the second of these operands to be a literal value. If two values in registers are being added, the "add register" instruction is used, but if a literal value were being added to a value in a register the "add immediate" instruction would be necessary. One rule characterizing an integer addition operation for such a machine, with an action to generate the "add immediate" instruction, might be the following:

```
IntReg ::= Plus(IntReg,IntLit) : IntRI_add
```

(Here the nonterminal `IntLit` represents the interpretation "a literal integer".)

Notice that the children of the `Plus` node in this rule have different interpretations; this rule cannot be used in a derivation that interprets the left child of the `Plus` node as an `IntLit` and the right child as an `IntReg`.

Because addition is commutative, however, it is possible to interchange the children of the `Plus` node without changing the resulting value. Therefore if a derivation interprets the left child of the `Plus` node as an `IntLit` and the right child as an `IntReg`, the tree parser should be able to simply invoke the `IntRI_add` action with the two operands reversed.

This possibility is indicated by using `::` instead of `:` between the rule and its associated action:

```
IntReg ::= Plus(IntReg,IntLit) :: IntRI_add
```

4 Summary of the Specification Language

The phrase structure of the specification language is described by the following ambiguous grammar:

```
Source: (Include / Declaration / Rule)+ .

Declaration: (Nonterm // ',') ':' Type ';' .
Nonterm: Identifier .
Type: Identifier .

Rule:
  Nonterm '::=' Node     (':' / '::') Action ['COST' Integer] ';' /
  Nonterm '::=' Nonterm  ':'          Action ['COST' Integer] ';' /
  Nonterm '::=' Fragment ':'          Action ['COST' Integer] ';' .
Action: Identifier .

Node:
  Terminal /
  Terminal '(' Nonterm [',' Nonterm] ')' /
  Terminal '(' (Type // ',') ')' /
  Terminal '(' Nonterm [',' Nonterm] ',' (Type // ',') ')' .
Terminal: Identifier .

Fragment: Terminal '(' Child [',' Child] ')' .
Child: Nonterm / Fragment .
```

Declarations and rules are the main components of a specification. Includes are simply names of files that are needed to define the identifiers representing types and actions.

An `Include` is a sequence of characters delimited by quotation marks ("). It is used unchanged in an `#include` directive output by the specification language translator. Only one `#include` directive is output for each distinct `Include`, regardless of how many times that `Include` appears in the specification.

An `Identifier` is a sequence of letters and digits, the first of which is a letter. As in C, the underscore (_) is considered a letter.

4.1 Declarations

Each nonterminal symbol must be declared, stating the type of the value associated with it:

```
Declaration: (Nonterm // ',') ':' Type ';' .
Nonterm: Identifier .
Type: Identifier .
```

Types are always represented by identifiers. If the type is a C basic type, no further declaration is necessary. Other types must be defined by a `typedef` construct that appears in some file named by an `Include` (see Chapter 4 [Summary of the Specification Language], page 18).

Here is a set of declarations that is appropriate for the examples given earlier in this document:

```
IntLit: int;
IntReg, FltReg: reg;  "mydefs.h"
```

Because `int` is a C basic type, no further information is necessary. `reg`, on the other hand, is declared by a `typedef` construct that appears in file '`mydefs.h`'. Thus the `Include` '`"mydefs.h"`' is used to provide access to that information.

4.2 Rules

As discussed earlier, there are three kinds of rules: node rules (see Section 2.1 [Rules Describing Tree Nodes], page 9), chain rules (see Section 2.2 [Chain Rules], page 11), and fragment rules (see Section 2.3 [Rules Describing Tree Fragments], page 12):

```
Rule:
  Nonterm '::=' Node    (':' / '::') Action ['COST' Integer] ';' /
  Nonterm '::=' Nonterm  ':'         Action ['COST' Integer] ';' /
  Nonterm '::=' Fragment ':'         Action ['COST' Integer] ';' .
Action: Identifier .
```

Each rule has an associated action and an optional cost (which defaults to 1 if not specified). The action is defined by an identifier, which must be defined in the file described by one of the `Include` components of the specification. That definition might be an `extern` statement or a `#define` directive (see Section 3.2 [Implementing Actions], page 16). The signature of the action is determined by the type of the left-hand-side `Nonterm` and the right-hand side as discussed above (see Section 3.1 [Actions and Values], page 14).

A node rule describes a single, possibly decorated, node of the tree being parsed (see Section 2.1 [Rules Describing Tree Nodes], page 9):

```
Node:
  Terminal /
  Terminal '(' Nonterm [',' Nonterm] ')' /
  Terminal '(' (Type // ',') ')' /
  Terminal '(' Nonterm [',' Nonterm] ',' (Type // ',') ')' .
Terminal: Identifier .
```

`Terminal` is the symbol of the ranked alphabet that is represented by the node. The `Node` must have k `Nonterm` children if `Terminal` has arity k.

Each `Terminal` is also associated with a specific set (possibly empty) of attributes. There is no limit to the number or types of the attributes decorating a node. Each attribute is denoted by a `Type`, which must be either a C basic type or an identifier defined by a `typedef` construct that appears in some file named by an `Include` (see Chapter 4 [Summary of the Specification Language], page 18).

The `::` marker distinguishes a commutative node (see Section 3.3 [Commutative Actions], page 16). This node must have two children, and those children must be distinct nonterminals. A commutative node rule may have arbitrary decorations.

A fragment rule describes a fragment consisting of two or more adjacent nodes:

```
Fragment: Terminal '(' Child [',' Child] ')' .
Child: Nonterm / Fragment .
```

Nodes participating in fragments may not be decorated. There is no limit on the size of a fragment.

5 Predefined Entities

TP generates a C module consisting of an interface file `tp_gen.h` and an implementation file `tp_gen.c`. The interface file exports definitions for the following identifiers:

`TPNode` the pointer type for internal representations of tree nodes

`TPNULL` a pointer of type `TPNode` representing no tree

`TPNull()` a macro without parameters that yields `TPNULL`, to be used where a function notation is needed (as in `WITH` clauses of LIDO's `CONSTITUENTS` construct)

Index