INDEXICAL ATTRIBUTE GRAMMARS

by

Senhua Tao

B.Sc., Beijing Polytechnical University, 1983
M.Sc., University of Victoria, 1987

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. R. N. Horspool, Departmental Member (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. Q. Wang, Outside Member (Department of Electrical Engineering)

Dr. J. Glasgow, External Examiner (Queen's University)

©SENHUA TAO, 1994

University of Victoria

All rights reserved. Thesis may not be reproduced in whole or in part, by
mimeograph or other means, without the permission of the author.

Supervisor: Dr. William W. Wadge

Abstract

In this dissertation we define a new attribute grammar system - *Indexical Attribute Grammars* (IAG). In IAG we define attributes over an implicit *indexical context space*. The indexical context space is a multidimensional space which is the product of a tree dimension, a multitime dimension, and an identifier dimension. Attributes on the indexical context space are intensions, whose values vary over different contexts: nodes of a parse tree, multitime points, and symbols. Indexical attribute grammars with denotational semantics form a new class of attribute grammars.

Indexical attribute grammars allow non-local attribute dependencies by using node switching operators. The use of communication attributes can therefore be reduced substantially in indexical attribute grammars.

Indexical attribute grammars can define attributes based on iterative algorithms. The value of an attribute at a node on a given parse tree can be defined as a data stream (or a nested data stream for a nested iteration) over the multitime dimension. The value of an attribute at a time point can be viewed as the value of the attribute at a particular step of the iteration. The attributes defined by iterative algorithms are temporal attributes, varying over the multitime dimension. Circular attributes whose evaluation can be terminated can be defined as non-circular but temporal attributes using time switching operators.

In indexical attribute grammars, we can define an aggregate attribute at a node on a given parse tree as a collection of values, gathered from other nodes, which varies over the identifier dimension. The information about identifiers can be collected as elements at the corresponding identifier points in the aggregate attribute. An aggregated value in the identifier dimension is not monolithic, its individual elements can be referred to by other attribute definitions through context switching operators.

The attribute evaluation of indexical attribute grammars is based on the tagged demand-driven computation model. The definitions of attributes on a given parse tree

form a dataflow graph. The evaluation of the attributes on the tree is the evaluation of the corresponding dataflow graph. Following the demand-driven method, only the values that are demanded at certain contexts are evaluated.

Examiners:

Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. R. N. Horspool, Departmental Member (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. Q. Wang, Outside Member (Department of Electrical Engineering)

Dr. J. Glasgow, External Examiner (Queen's University)

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor Bill Wadge who, through his insights and careful guidance, has made this dissertation possible. He has provided me with timely encouragement and advice as well as an outstanding amount of freedom to pursue research issues I considered important while at the same time critically appraising my work. Most importantly, Bill has that invaluable asset of the good supervisor, an ever open door.

I would like to thank my other committee members, Dr. R. N. Horspool, Dr. G. C. Shoja, and Dr. Q. Wang, for their conscientious reading of the dissertation and suggestions.

Special thanks go to Dr. Nigel Horspool for suggesting the topic of this dissertation.

I would also like to thank all members of the Department of Computer Science at University of Victoria for the friendly and pleasant working environment, especially members of the functional programming group in the department with whom I have had very stimulating discussions.

I was supported financially at University of Victoria by a University of Victoria Fellowship and a BC Advanced Systems Graduate Scholarship.

Contents

Abstract ii

Acknowledgements v

Contents vi

List of Figures ix

List of Tables xi

1 INTRODUCTION 1

1.1 Attribute Grammars 1

1.2 Existing Problems 4

1.3 Indexical Logic 7

1.4 Overview of Indexical Attribute Grammars 9

1.5 Overview of the Dissertation 11

2 BACKGROUND 13

2.1 Attribute Grammars 13

2.2 Related Work 18

2.2.1 Global Attributes 18
2.2.2 Circular Attribute Grammars 21
2.3 Indexical Programming 23

3 An Attribute Definition Language **27**
3.1 Syntax 27
3.2 The Context Space 27
3.2.1 The tree dimension 27
3.2.2 The Multitime Dimension 29
3.2.3 The Identifier Dimension 30
3.2.4 The Context Space 30
3.3 Functional Semantics Without Indexical Operators 30
3.4 Semantics of Indexical Operators 39
3.4.1 Node Switching Operators 40
3.4.2 Time Switching Operators 42
3.4.3 Identifier Switching Operators 44
3.5 Non-primitive Indexical Operators 45
3.6 Denotational Semantics of IFADL 49

4 Programming in IFADL **51**
4.1 Pointwise Operations 51
4.2 Expressing Algorithms Operating on Trees 53
4.3 Expressing Iterative Algorithms Operating on Trees 57
4.4 Evaluation of IFADL Expressions 60

5 THE DEFINITION OF IAG 62

5.1 The Definition 62

5.2 Syntax of Indexical Attribute Grammars 69

5.3 Denotational Semantics of Indexical Attribute Grammars 71

6 EXPRESSIVE POWER OF IAG 74

6.1 Noncircular Attribute Grammars 74

6.2 Circular Attribute Grammars 81

6.2.1 Classification of Circular Attribute Grammars 81

6.2.2 Circular Attributes via Temporal Attributes 83

6.2.3 Nested Circularities via Multiple Time Dimensions 93

6.3 Defining Aggregate Attributes 102

6.3.1 Aggregating Attributes from an Attribute Value Tree 104

6.3.2 Aggregating Attribute Values from an Attribute Value Tree with Nested Structure 107

6.3.3 Temporal Aggregate Attribute Values 110

7 AN IMPLEMENTATION STRATEGY 116

7.1 The Demand-Driven Dataflow Computation Model 116

7.2 Structure of IAGCG 119

7.3 The Eductive Attribute Evaluator EAE 121

8 CONCLUSIONS 124

8.1 Summary 124

8.2 Contributions and Advantages 125

8.3 Future Work 128

Bibliography 130

1 Syntax of IFADL 135

2 An Indexical Attribute Grammar for Type Checking 137

List of Figures

1.1 A parse tree with attributes 3

2.1 Local dependency graphs for Example 2.1 17

2.2 The derivation tree for string "10.01" 18

2.3 A dataflow network 25

3.1 An indexed tree 29

4.1 The values of A and B 52

4.2 The value of $A + B$ 52

4.3 The union-tree version of $A + B$ 53

4.4 The binary search tree for input list [5 2 7 3 6 9 8 1 4] 55

4.5 A circular dependency graph 58

4.6 A tree-structured circular dependency graph 58

4.7 A non-tree shape example 60

5.1 A parse tree with attribute values 64

5.2 Dependencies without conflicts 66

5.3 Dependencies with conflicts 67

6.1 An attribute value tree 78
6.2 An attribute value tree with nested structure 82
6.3 The attribute value tree at time 0 89
6.4 The attribute value tree at time 1 90
6.5 The attribute value tree at time 2 90
6.6 The attribute value tree at time 3 91
6.7 The attribute value tree at time 0 94
6.8 The attribute value tree at time 1 95
6.9 The attribute value tree at time 2 96
6.10 Branching new time dimensions for nested iterations 99
6.11 Reducing a time dimension when the inner iteration is finished 100
6.12 Increasing a time point for outer nested iteration 101
6.13 Reducing time dimensions when all the nested iterations are finished 103
6.14 An attribute value tree with aggregate values 106
6.15 Nested structure of a program 107
6.16 The nested structure of a program 111
6.17 The aggregation at time 0 113
6.18 The aggregation at time 1 113
6.19 The aggregation at time 2 114
6.20 The aggregation at time 3 115

7.1 Demand driven computation on a dataflow network 117
7.2 A dataflow network for the attribute instances in a parse tree 118
7.3 The structure of a compiler generated by IAGCG 120
7.4 A dataflow network 122

List of Tables

4.1 Values at iterations 59

Chapter 1

INTRODUCTION

In this dissertation we define a new attribute grammar system - *Indexical Attribute Grammars* (IAG). In IAG we define attributes over an implicit *indexical context space*. The indexical context space is a multidimensional space which is the product of a tree dimension, a multitime dimension, and an identifier dimension. Attributes on the indexical context space are intensions, whose values vary over different contexts: nodes of a parse tree, multitime points, and symbols. We show that the indexical approach enriches and improves conventional AG systems.

1.1 Attribute Grammars

Before attribute grammars were introduced, there existed a simple, elegant and formal method for describing the syntax of programming languages - namely *context-free grammars*. But we had no method for describing the semantics of programming languages declaratively. All the methods were operational. For example, we could write procedures to check the semantic correctness of programs. People found out that for some simple languages, such as arithmetic expressions, the meaning (for example, the compiled code) of a given expression can be synthesized in a straightforward way by recursively building up the meanings of its subexpressions. The meaning of an expression can therefore be evaluated bottom-up while constructing the corresponding parse tree. In that sense, we

can incorporate the semantic information of a language into its syntactic definition.

Unfortunately, most programming languages are context-dependent. For example, the *type* information for variables in Pascal is context-sensitive.

```
begin
  var x: integer;
  x = y;
end;
```

The above Pascal program phrase is syntactically correct, but semantically wrong, since the type of y is undefined. It is also clear that the semantic checking for the statement `x = y;` cannot be done by analyzing the type information in its own subtrees.

In 1967, Knuth introduced a new formalism - *attribute grammars* [Knu68][Knu71] for defining programming languages. An attribute grammar can define not only the syntax of a programming language, but also the context-sensitive semantics, at the same time, in a declarative way.

An attribute grammar is a context-free grammar with attribute definition rules. In attribute grammars, we associate a set of attribute symbols with each non-terminal grammar symbol. For a node labeled by a non-terminal symbol in a given parse tree, there is a corresponding set of attribute instances. The values of the attribute instances at the node represent the semantics of the node. For example, consider the following production which defines the syntactical structure of an assignment statement in a programming language:

```
statement -> identifier "=" expression
```

We associate an attribute symbol `type` with the grammar symbols `identifier` and `expression`. We also associate an attribute symbol `match` with the grammar symbol `statement`. The value of a `type` instance at a node labeled by an `identifier` indicates the type of the identifier. The value of a `match` instance at a node labeled by a `statement` describes whether the statement is semantically correct.

Attribute definitions specify how to compute the values of certain attribute instances as a function of other attribute instances. Attribute definitions are associated with the production rules of the underlying context-free grammar. For example, we can associate

a definition of match with the production which specifies the structure of an assignment sentence:

```
statement -> identifier "=" expression
  statement.match = identifier.type eq expression.type;
```

Here we use *symbol.attribute* to represent a grammar symbol with an attached attribute symbol, which we call an *attribute occurrence*. According to the definition, in a given parse tree (Figure 1.1), the node labeled by the non-terminal symbol statement will have a value of match. The value of match depends on the values of type at its first and third child nodes in its subtree. The dependencies are indicated by the arcs. If the two values are equal, then the value of match will be true. That means that the assignment statement is semantically correct. Otherwise, the value of match is false and the statement is semantically incorrect, though it is correct syntactically.

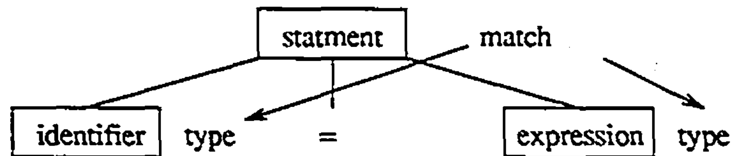

Figure 1.1: A parse tree with attributes

There are two important kinds of attributes in attribute grammars: *synthesized* attributes and *inherited* attributes. Inherited attributes are introduced to describe the context-sensitive semantics of a language. In a given parse tree, a synthesized attribute instance at a node depends on the attribute values in its subtree. An inherited attribute instance depends on the attribute values on its parent node. Synthesized attributes are evaluated bottom-up and inherited attributes are evaluated top-down. Knowing if an attribute is synthesized or inherited helps us to determine how to evaluate the values of the attribute over a given parse tree, for example, in a top-down or bottom-up manner.

Since attribute grammars allow us to specify both the syntax and semantics of a language, attribute grammars are an ideal formal specification tool for defining programming languages. Attribute grammars have been applied to many areas, such as compilation [Far84], control and data flow analysis [BJ78][CR88], code generation [GF82], text editing

[TRS1][Rep84][HK86], database systems [Pla86][Muc85], and VLSI design [JS86]. They have proved to be a useful formalism for specifying context sensitive semantics of programming languages and other systems.

1.2 Existing Problems

There are some problems with existing attribute grammar systems. One problem is the need for so-called *communication* attributes. In an attribute grammar we associate attribute definitions with production rules. There is a restriction on these attribute definitions: an attribute occurrence in a production rule can only be defined in terms of the attribute occurrences in the same production rule. In other words, we only allow attributes to be defined *locally* within a production. For example, in the following attribute grammar we can associate an attribute `type_table` with the grammar symbol `decl_block`.

```
program -> decl_block stmt_list
decl_block -> ...
  decl_block.type_table = ...
stmt_list -> statement stmt_list
statement -> id "=" expression
  statement.match = id.type eq expression.type;
```

In a given parse tree, there is an attribute instance of `type_table` at a node labeled by `decl_block`. The value of this attribute instance contains all the type declarations for all the variables declared in the declaration block. We can also think of the value of `type_table` as a symbol table in a conventional compiler. Conversely, to evaluate the value of `type` at a node labeled by `id` in a given parse tree, we have to know the type of the identifier declared in the declaration part. To do so, at a node labeled by `id` in the `stmt_list` structure, we need to access the value of `type_table` associated with the node labeled by `decl_block`. Since attribute grammars are context-free, there is no way to define an attribute, in the fourth production, which depends on an attribute occurrence in the second production rule directly. In order to define an attribute occurrence which depends on the attribute occurrences in other productions, we may have to introduce communication attributes in some productions to bridge the gap. These communication

attributes serve no purpose other than passing on desired values. For example, we can attach the attribute symbol `table` to non-terminal symbols `stmt_list`, `statement`, and `expression`. Then the attribute definitions in the corresponding production rules can be defined as followings:

```
program -> decl_block stmt_list
  stmt_list.table = decl_block.type_table;
decl_block -> ...
  decl_block.type_table = ...
stmt_list -> statement stmt_list
  statement.table = stmt_list0.table;
  stmt_list1.table = stmt_list0.table;
statement -> id "=" expression
  statement.match = id.type eq expression.type;
  id.type = get_type(id.lex, statement.table);
  expression.table = statement.table;
```

Here `stmt_list0` indicates the grammar symbol `stmt_list` occurred on the left hand side of the production rule and `stmt_list1` indicates the grammar symbol `stmt_list` occurred on the right hand side of the production rule. `get_type` is a function which returns the type from a type table for a given identifier. Evaluating these extra attribute values may involve a large amount of computing time. Moreover, it may also require a large amount of storage space for duplicate values, especially when the attributes have complex values.

Another problem related to the locality of attribute definitions is that of the so-called *aggregate* attributes. To analyze the semantics of a program, we usually need a symbol table to store the information about identifiers appearing in a program, as shown above. We can think of a symbol table as an attribute with aggregate values. The information about each identifier is an element of the aggregate value. Again, because of the locality property of attribute definitions, attributes cannot aggregate their elements in a parse tree directly. To aggregate attribute values on a parse tree, we may have to define communication attributes which accumulate the aggregated values as they are passed from node to node.

An additional problem related to aggregate attributes is that of *incremental evalu-*

ation. The value of an aggregate attribute consists of many elements, such as entries of a symbol table. Conventionally, an aggregate attribute is considered as a monolithic structure. Changing any of its individual elements results in the entire structure being updated. Thus, even those attributes that depend only on unchanged elements have to be re-evaluated. This makes incremental evaluation inefficient.

Another problem with attribute grammars is that of *circular attribute definitions.* Since there may exist both synthesized and inherited attributes in attribute grammars, it is possible that the value of an attribute could be indirectly defined in terms of itself. Such attributes are called *circular attributes.* Conventional attribute grammars do not allow circular attribute definitions. A circularly defined attribute at evaluation time will yield an undefined value. However, since recursive and iterative methods are natural for solving many problems, it seems that allowing circularly defined attribute grammars is unavoidable. Examples are the denotational semantics of a *while* statement, and *live variables* in data flow analysis. They can be defined naturally by recursive algorithms. Although circular attribute definitions can be transformed into non-circular ones, the transformed attribute grammars may involve many extra attributes and definition rules. Conventional attribute grammars lack the power to specify attribute definitions based on iterative or recursive algorithms.

Since the attribute values in a given parse tree represent the meaning of the corresponding language sentence, the semantic checking of a language sentence becomes the evaluation of the attribute values on the corresponding parse tree. To evaluate the two kinds of attribute values, an evaluation order among the attribute instances in a given parse tree must be decided on before evaluation starts. If an operation needs a value which has not been created, the evaluation process may fail because the operation operates on an undefined value. Sometimes, to evaluate all the attribute values in a given parse tree, we need a multipass evaluator to guarantee that all the attribute values are evaluated in correct order. Multipass evaluation is time inefficient. To avoid multipass evaluation, people impose restrictions on attribute definitions so that the attribute values in a given parse tree can be evaluated within one pass of tree walking. However, the restrictions

may decrease the expressive power of attribute grammars or involve extra attributes and corresponding definitions.

To solve these problems, new AG systems and approaches have been proposed. Examples are *copy rule chains* [Hoo86], *remote attributes* [RMT86], *global attributes* [RT86], *successive approximation evaluation* [Far86], *gate attributes* [WJ88], *incremental evaluation* [CR88][Alb89][Jon90], *action event-driven evaluation* [Kai89], and so on. Also, some new kinds of attribute grammars have been introduced, such as *coupled attribute grammars* [GG84][Gie88][RPJ94] and *higher order attribute grammars* [VSK89][TC90]. However most of these systems and approaches solve the problems at the attribute evaluation level. They are implementation-dependent. In this dissertation, we try to solve the problems at a higher level. We introduce indexical semantics into attribute grammars. Using indexical semantics, the above problems can be solved at the attribute definition level. We claim that indexical attribute grammars as defined in this dissertation form a new and more powerful class of attribute grammars.

1.3 Indexical Logic

Indexical logic is a subset of intensional logic. Indexical logic is concerned with expressions whose meanings depend on an implicit context, sometimes called *a possible world* [WA85] [van88]. An expression over a context space is an *intension*. An *intension* is a function which maps the expression from the context space to its value domain. The value of an expression at a context is also called an *extension*.

This type of logic was originally developed to help understand natural languages [Tho74][DWP80]. The meaning of a natural language sentence may depend on many conditions. For example, the meaning of *Today's sunset is 5 minutes earlier than yesterday's sunset* depends on when and where we utter the sentence. Here one condition is *time* and one condition is *place*. If a natural language contains only this kind of simple sentence whose meaning depends only on time and place, we can construct a context space for the language. The context space is two dimensional. One dimension is the *time* and

the other one is the *place*. The coordinates of the time dimension could be measured in days, and the coordinates of place dimension could be specified as cities. A context (d, c) in the context space consists of a time point d and a place point c.

A language expression over the context space is an intension. For example, the meaning of the expression *Today's sunset is 5 minutes earlier than yesterday's sunset* can be expressed as a formula $t = yesterday(t) - 5$. In the formula, t represents the time of the sunset, which is an intension. *yesterday* is an indexical operator that switches context from a context, namely today, to another context, namely yesterday. The value or the *extension* of the sentence then depends on a given context. The result of the application *yesterday(t)* on a given context (d, c) is the sunset time of one day before d at the same city c. If the current sunset equals to yesterday's sunset minus 5 minutes, then the sentence is true at (d, c), otherwise the sentence is false.

Conversely, the above expression cannot be true at all the contexts over the context space. Usually the sunset time will be delayed from winter to summer and be increased from summer to winter. It also depends on whether we are at the north pole or in the tropics. We say that the value of an expression *varies* over the context space. The value of the expression may be different in different contexts.

Through the study of attribute grammars, we see that in a given parse tree, an attribute may have different values at different nodes. The values may be defined by a single definition rule, or by different definitions in different production rules. It is difficult to distinguish these attribute instances in a particular parse tree at the definition level. We tend to think of those attribute instances as different, though they may share the same attribute name. The problem can be solved naturally with indexical semantics. For example, we can define the nodes of an arbitrary tree as our context space. That means that each node in a parse tree is determined by a unique context. Attributes are therefore intensions over the context space. Using intensional operators, we can clearly describe attribute dependencies over the context space at the definition level.

1.4 Overview of Indexical Attribute Grammars

In an IAG, attributes are intensions over a context space which is a product of a *tree dimension*, a *multitime dimension*, and an *identifier dimension*. A tree point, a time point, and an identifier point constitute a context, or a possible world. To manipulate the values of intensions, we use indexical operators to switch context from one to another.

For a given parse tree, we consider it as a subtree of the tree dimension. Attributes of the parse tree are defined in an IAG as intensions whose values vary over the tree dimension. The values of an attribute at the context space are defined by attribute definitions. Using node switching operators, we can define nonlocal attribute dependencies directly, especially those for upward dependencies. The use of communication attributes can be reduced substantially in an IAG.

As an intension, the value of an IAG attribute at a node of a given parse tree may also vary over the multitime dimension. Thus the value of an attribute at a node is a (possibly nested) data stream, whose elements are indexed by time points. When an attribute is varying in time, we call it a *temporal attribute*. The value of a temporal attribute at a time point can be defined in terms of its values at other time points by using time switching operators. This kind of "self-dependency" does not create undefined values, as long as they refer to defined values at different time points. As we mentioned in the last section, conventional attribute grammars do not allow circular attribute definitions, even though some attributes can be defined more clearly and more logically by recursive or iterative algorithms.

For the attributes which can be described by recursive or iterative algorithms, in an IAG, we can define them as temporal attributes, or functions of time. We explicitly describe the required initial values and the termination condition for a temporal attribute. At evaluation time, the evaluation order of temporal attributes is decided by the time switching operators. When the same initial values are given to a circular attribute, the evaluation always yields the same result.

The identifier dimension allows us to define aggregate attributes, such as symbol tables. In other words, the value of an aggregate attribute can vary at different identifier points.

For example, we can define an attribute *type_table* to record the types of identifiers in a block structure. For a given parse tree, at a node which denotes a block structure, the value of attribute *type_table* at each identifier point is the type information for the identifier in the block. Using identifier switching operators, attributes can aggregate values from a parse tree, so that the intermediate attributes which record partially aggregated values are eliminated. Each element in an aggregate value at a node has its own identifier index. Using identifier switching operators, an element of an aggregate value can be referred to individually. When an element is modified, it will effect only the attributes depending on the individual element, not the entire aggregate value.

The following is an example of an IAG grammar which defines the same type information as the one in section 2.

```
program -> decl_block stmt_list
  type_table = child(type_table, 0);
decl_block -> ...
  type_table = ...
stmt_list -> statement stmt_list
statement -> id "=" expression
  match = child(type, 0) eq child(type, 1)
  type = ati(lex, upasa(type_table));
```

Here `child` and `upasa` are node switching operators, `ati` is an identifier switching operator, and `lex` is an attribute symbol whose value at a node labeled by an identifier is the identifier's lexical value. The operator `upasa` allows us to refer to the first valid required attribute value on the path from the current node to the root node. The operator `ati` returns the element from an aggregate value at a given identifier point. In IAG, the attribute instances attached to the nodes labeled by `stmt_list`, `statement`, and `expression` are eliminated.

In this dissertation, we also define an indexical functional language named IFADL. IFADL is the attribute definition language for an IAG. The indexical operators of an IAG are implemented as the functions in an IFADL. Attributes in an IAG can be viewed as variables in IFADL, and attribute definitions are expressions in IFADL. The evaluator of an IAG is an interpreter for IFADL. Therefore, by using IFADL as the attribute defining

language, we make IAG grammars executable.

The evaluation of intensions can be based on a *demand-driven* computation method. Using this method, the evaluation order among the values of intensions is automatically decided at evaluation time. Consequentially, there is no need to divide attributes into synthesized or inherited attributes in IAGs. The value of an intension in a given context is evaluated only when the value is demanded. Therefore, only the required values are evaluated.

Since our proposed evaluator of an IAG system uses a demand-driven evaluation method, there is no need for restrictions on an IAG. The evaluator of an IAG system does not need special facilities to solve conventional evaluation problems, such as sharing attribute values, attaching dependent lists to elements of aggregated values, detecting circularities, or supporting fixed-point finding algorithms, because they are either handled automatically by the demand-driven scheme, or a part of the implementation of IFADL.

1.5 Overview of the Dissertation

In Chapter 2, we first introduce some background on conventional attribute grammars. We also discuss some problems with conventional attribute grammars such as communication attributes, circular attributes, and aggregate attributes. And we outline some related work for solving the problems.

In Chapter 3, we describe the attribute definition language IFADL. IFADL is developed from the functional programming language ISWIM. In this chapter we define an extension of ISWIM's semantics with indexical semantics based on a multi-dimensional context space. We also define primitive indexical operators in IFADL which switch contexts in the context space.

In Chapter 4, we describe programming in IFADL through examples. We show how to define values on nodes of a tree using the tree dimension, data streams using the time dimensions, and table-like aggregate values using the identifier dimension.

In Chapter 5, we define indexical attribute grammars. We use IFADL expressions to

specify attribute definition rules. In this chapter, we define parse trees as subtrees of the general tree dimension and attributes as IFADL variables whose values at nodes of the trees may also vary in the time and identifier dimensions. We define the distribution of attribute definitions over the nodes of a parse tree as an intension, and give the denotational semantics of attributes on the tree.

In Chapter 6, we show the expressive power of indexical attribute grammars. We show how communication attributes can be removed by using node switching operators. We show how to use time switching operators to define attributes as temporal objects, and how to specify circular attributes explicitly as non-circular attributes with stream values. Finally, we also show how to define attributes by using identifier switching operators to manipulate aggregate attribute values on a parser tree.

In Chapter 7, we describe an implementation strategy for evaluating attribute values in indexical attribute grammars. We also give the strategy for building compiler generators based on indexical attribute grammars.

Finally, in Chapter 8, we summarize the contributions of the dissertation and discuss future work.

Chapter 2

BACKGROUND

2.1 Attribute Grammars

Attribute grammars are a formalism for specifying the syntax and the context-sensitive semantics of programming languages, as well as for implementing editors and compiler-writing systems. Attribute grammars form an extension of the context-free grammar framework in the sense that information is associated with programming language constructs by associating attributes with the grammar symbols representing these constructs.

The basic idea of attribute grammars is that we associate attribute values with each node in the parse tree which represents the syntax of a given language string. These attribute values specify the semantics of the node. The attribute values in a parse tree are evaluated according to the corresponding definitions associated with the context-free grammar of the language. In this way, the semantics of the language can be defined together with the syntax of the language. In this section, our introduction to attribute grammars is based on [Knu68][Knu71][DJL88].

An attribute grammar is a triple $AG = (G, A, D)$. $G = (T, N, S, P)$ is the underlying context-free grammar with T the set of terminals, N the set of nonterminals, $S \in N$ the start symbol, and P the set of productions. A production $p \in P$ in a context-free grammar has the form

$$p: X \rightarrow X_0\, X_1\, \ldots\, X_{n_p-1}$$

where $n_p \geq 0$, $X \in N$, and $X_k \in T \cup N$ for $0 \leq k \leq n_p - 1$.

The second component A is a set of attribute symbols. A grammar symbol in a production may occur more than once. In order to distinguish the attribute values associated with different grammar symbol occurrences in an attribute definition, we call an attribute symbol together with a grammar symbol occurrence an *attribute occurrence* of the production rule.

We let O_p denote the set of attribute occurrences of the production p. An attribute occurrence o in a production p has the form

$$X.a$$

where X is a nonterminal grammar symbol in p and a is an attribute symbol.

The last component $D = < D_p >_{p \in P}$ is an indexed family of sets of attribute definition rules. Each production p has a set of attribute definition rules D_p. The attribute definition rules define some of the attribute occurrences in the production. In a given parse tree, an attribute definition in the production defines the value of the attribute at each node that corresponds to the grammar symbol occurrence in the production. For a given production $p \in P$ (with k grammar symbols), D_p is a set of attribute definition rules associated with p. An attribute definition rule has the form

$$o_j = e(o_0, \ldots, o_i, \ldots, o_m)$$

where $o_i \in O_p$ are attribute occurrences associated with the production p, and

$$e(o_0, \ldots, o_i, \ldots, o_m)$$

is an expression with $o_0, \ldots, o_i, \ldots, o_m$ as free variables. For example, suppose production p has the form

$$p : X \rightarrow Y\ Z$$

and $A = \{a, b\}$. The attribute definition rule

$$X.a = Y.b + Z.b$$

defines the value of the attribute occurrence $X.a$ to be the sum of the values of attribute occurrence $Y.b$ and attribute occurrence $Z.b$.

An attribute definition rule defines the value of a particular attribute occurrence as a function of other attribute occurrences in the same production. An attribute occurrence can only be defined in terms of attribute occurrences in the same production.

The data dependencies among the attribute occurrences in a production p form a *local dependency graph*

$$G_p = \{(o_i,\ o_j)|\ o_j \text{ occurs in a definition of } o_i \text{ in } D_p\}$$

The local dependency graph of production p has the attribute occurrences in D_p as vertices and their data dependencies as arcs. We say that in production p, an attribute occurrence o_i depends on attribute occurrence o_j if o_j appears in the definition of o_i. In this case, there will be an arc from o_i to o_j in the local dependency graph G_p.

A context-free grammar generates a unique tree structure (called *a parse tree*) for each of its sentences (if the grammar is not ambiguous). A node n of the tree is labeled by the grammar symbol X which occurs on the left hand side of the rule associated with n. An attribute symbol a together with a node n is called an *attribute instance* which is a pair (a, n). The semantics of the sentence is the value of a distinguished attribute instance at the root node of the parse tree.

A *derivation tree* is a parse tree together with the dependency graph of the attribute instances on the tree. The dependency graph is obtained by connecting together the local dependency graphs for each node. The derivation tree is used to determine the evaluation order of attribute instances on the tree.

For example, the following is an attribute grammar that defines a language whose sentences are binary numbers [Knu68][Knu71].

Example 2.1 (Converting binary strings to decimal values)

A = {value, length, scale}
p1: N → L "." L
 N.value = L_1.value + L_2.value

L_1.scale = 0
L_2.scale = − L_2.length
p2: N → L
N.value = L.value
L.length = 0
p3: L → L B
L_0.value = L_1.value + B.value
L_0.length = L_1.length + 1
L_1.scale = L_0.scale + 1
B.scale = L_0.scale + 1
p4: L → B
L.value = B.value
L.length = 1
B.scale = L.scale
p5: B → 1
B.value = $2^{B.scale}$
p6: B → 0
B.value = 0

In the above example, the attribute *value* is associated with grammar symbols N, L, and B. The attribute *length* is associated with grammar symbol L. The attribute *scale* is associated with grammar symbols L and B. The local dependency graphs for the above example are shown in Figure 2.1.

In the local dependency graphs, the vertices are grouped with their grammar symbols. For a production $p : X \to X_0X_1 \ldots X_{n_p-1}$, the attribute, associated with the grammar symbol X in p is at the top of G_p and the attributes associated with grammar symbols X_i in p, $0 \leq i \leq n_p - 1$ are placed at the bottom of G_p. The derivation tree for the binary number "10.01" generated from the grammar is shown in Figure 2.2.

At a node of a particular parse tree of the above attribute grammar, the attribute instance of *value* is a rational decimal value computed from attribute instances in the subtree rooted by the node. The value of *length* at a node is an integer that is the length of the substring represented by the subtree rooted by the node. The value of *scale* at a node is an integer that is the scale of the node. The attribute *value* associated with the start symbol N at the root node denotes the semantics of the sentence.

The semantics of a sentence is obtained by evaluating attribute instances on the

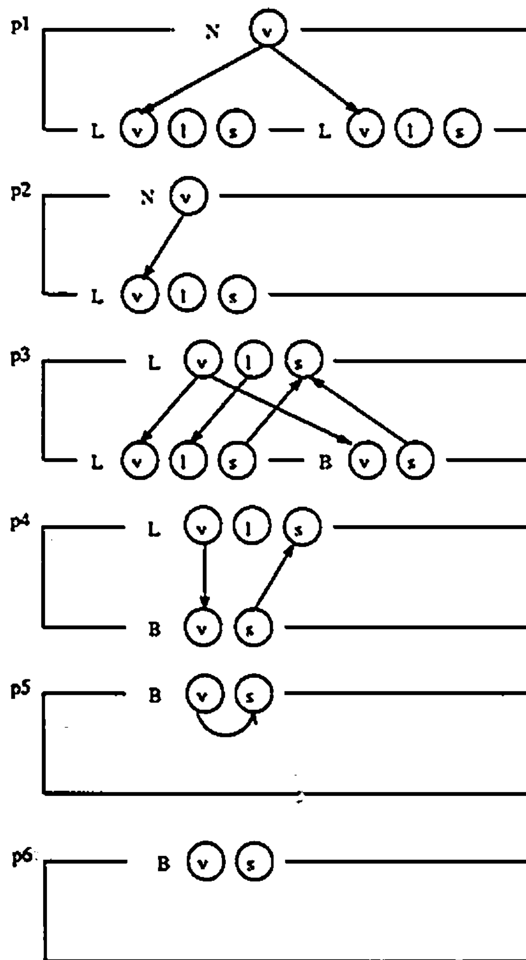

Figure 2.1: Local dependency graphs for Example 2.1

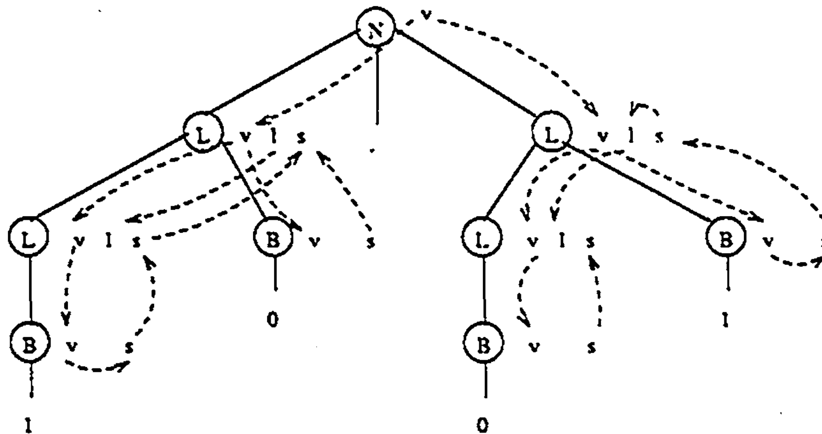

Figure 2.2: The derivation tree for string "10.01"

derivation tree. There are basically two kinds of attribute evaluation methods: tree-walk [KS76][Boc76][Kat84][Kos85][Joh87] and lazy evaluation [Tao87][Fro92]. In tree-walk evaluation, an *evaluation plan* has to be determined before the evaluation starts. The plan tells the evaluator how to scan a derivation tree in one or more passes and which attribute instances on the tree are evaluated at each pass. In lazy evaluation, the evaluator initially evaluates the attribute instances that constitute the semantics of the derivation tree. If a required attribute instance does not have a value yet, the evaluator will evaluate other attribute instances that are needed to compute the attribute instance. The evaluator repeats the process until all the required values are satisfied. In lazy evaluation, only the required attribute instances are evaluated.

2.2 Related Work

2.2.1 Global Attributes

According to the definition of attribute grammars, an attribute occurrence in a production rule can only be defined by the attribute occurrences that are in the same production rule. Because of the locality restriction of attribute definitions, if an attribute instance at a node in a parse tree refers to an attribute instance at another node which is not a parent, child, or sibling of the former, some communication attributes have to be defined at those nodes

that form a path to pass the required value between the two nodes. These communication attributes are auxiliary; they do not contribute any meaning to the attribute grammar by themselves.

To avoid keeping these duplicate attribute values, some techniques such as *copy rules* [Hoo86] are introduced into attribute grammars. An attribute instance defined by a copy rule will not be evaluated. In other words, an attribute instance defined by a copy rule will only pass a reference to a storage location that contains the real value. For example, using the copy rules we can rewrite Example 2.1 as follows (where a copy rule is indicated by $=_c$).

A = {value, length, scale}
N → L . L
  $N.value = L_1.value + L_2.value$
  $L_1.scale = 0$
  $L_2.scale = - L_2.length$
N → L
  $N.value =_c L.value$
  $L.length = 0$
L → L B
  $L_0.value = L_1.value + B.value$
  $L_0.length = L_1.length + 1$
  $L_1.scale = L_0.scale + 1$
  $B.scale = L_0.scale + 1$
L → B
  $L.value =_c B.value$
  $L.length = 1$
  $B.scale =_c L.scale$
B → 1
  $B.value = 2^{B.scale}$
B → 0
  $B.value = 0$

Some efforts have also been made to extend the attribute grammar formalism by allowing nonlocal dependencies. In [JF84], attribute grammars are extended to allow nonlocal attribute definitions. Global attributes such as *upward remote references* are also allowed in some attribute grammar formalisms [RT86][RMT86]. In their notation, an

upward remote reference has the form

$$\mathbf{up}(X_1.a_1, \ldots, X_k.a_k) \quad \text{(where } X_i.a_i \text{ denotes an attribute occurrence)}$$

For each $X_i.a_i$, it refers to the first attribute instance in the sequence of attribute instances, $(a_i, n_1), (a_i, n_2), \ldots, (a_i, n_k)$, on the path from the point of reference (a node) to the root of the parse tree where $n_j (1 \le j \le k)$ is labeled by X_i. Using this concept, the communication attributes can be removed from attribute grammars. The following is an example in [RT86] with upward remote references.

Example 2.2 (Using upward remote references)

```
Program → Block
      Block.env = φ
Block → DeclList StatList
DeclList1 → DeclList2 Decl
DeclList → Decl
Decl → Id Block
      Block.env = ProcDecl(up(Block.env), spelling(Id));
      up(Block.env) = ProcDecl(up(Block.env), spelling(Id));
Decl → Id1 Id2
      up(Block.env) = VarDecl(up(Block.env), spelling(Id1), spelling(Id2));
StatList1 → StatList2 Stat
StatList → Stat
Stat → Id
      CheckUse(up(Block.env), spelling(Id));
Stat → Block
```

The above is a type checking example. In a program of the language defined by the above attribute grammar, the value of an attribute *env* is associated with each node labeled by the grammar symbol `Block` in the corresponding parse tree. To check the type of an identifier, the upward remote reference function **up** will trace the type information which is the value of *env* associated with the first ancestor node labeled by the grammar symbol `Block` on the path to the root node.

This example also tries to solve the so called *aggregate attributes* problem. The aggregate attribute in this example is *env*, a symbol table which contains the type information about identifiers. Instead of collecting the elements of *env* from each node labeled by

Block, the upward remote reference function up also allows the elements of *env* directly to be "written" into the attribute value of *env* at their first ancestor nodes labeled by *Block*. In fact, the upward remote reference can be considered as syntactic sugar for a sequence of copy rules.

2.2.2 Circular Attribute Grammars

The dependencies of attribute instances in a derivation tree not only raise the evaluation order problem, but also raise the circularity problem, that is, an attribute in an attribute grammar may depend on itself directly or indirectly through attribute definitions. If an attribute *instance* recursively depends on itself in a derivation tree generated by an attribute grammar, then the attribute is called a *circular attribute*. When a circularly defined attribute instance is evaluated, the computation cannot terminate. In the definition of attribute grammars given by [Knu68][Knu71], these circularities are viewed as errors, since they yield undefined values. But there are many problems whose solutions are naturally based on recursive algorithms. To express recursive algorithms in traditional attribute grammars, one approach is to transform certain recursive algorithms into non-recursive ones by introducing more attributes and definition rules. In this case, the resulting attribute grammars usually become difficult to understand and the evaluation may become inefficient.

Although circularities in traditional attribute grammars are treated as errors, certain circular attribute grammars may have valid meanings. A result given by [Far86] shows that if an attribute grammar has circular, but well-defined attribute definitions, a *Fixed-Point-Finding* evaluator can evaluate the circular attribute instances using a successive approximation approach. A well-defined attribute grammar requires that for the values of circular attribute instances, their definitions must be monotonic and satisfy an ascending chain condition. In the evaluation, initially all the attribute instances are assigned the value $\perp$. A circularly defined attribute instance may be evaluated several times until the value stops changing. The evaluation terminates when all the attribute instances have stable values.

Example 2.3 given by [Far86] shows an attribute grammar fragment for computing *live* variable information in the dataflow analysis of programs. The solution of the example is, for each statement, the set of identifiers that are alive on entry to that statement. Attribute instances of *live* constitute this solution. The circularity in the attribute grammar is caused by the definition of attribute *live* associated with the production for **while**-statements, where *stmt.live* depends indirectly on *stmts.live* which indirectly depends on itself.

Example 2.3

```
stmt → id "=" exp
      stmt.live = (stmt.out − id) ⋃ exp.in
stmt → "while" exp "do" stmts
      stmt.live = stmt.out ⋃ (stmts.live ⋃ exp.in)
      stmts.out = stmt.out ⋃ (stmts.live ⋃ exp.in)
stmts → stmt ";" stmts
      stmts0.live = stmt.live
      stmt.out = stmts1.live
      stmts1.out = stmts0.out
stmts → .
      stmts.live = stmts.out
```

There are also some natural functions that are not defined as above but whose solutions are computable [WJ88]. For example, to simulate the denotational semantics of a programming language that involves **while** statements, we can define attributes *init* and *final* for the statements which specify the initial and final states. Since the attribute *final* of a **while** statement depends on the value of *init* of the **while** statement, the attribute grammar of the language involves circularities. Although this kind of circularity does have least fixed-points, the semantic function of the attribute *final* does not satisfy the monotonic and ascending chain conditions. The termination of evaluating this circular definition depends on a default value of the conditional expression of the **while** statement.

One solution for this kind of circular attributes is to add a *gate* attribute to a cycle [WJ88]. An attribute instance in a cycle has a sequence of values. The evaluation for attribute instances in a cycle is iterative. The gate attribute is the key of the cycle. The gate attribute initially picks up a value from outside of the cycle to start the evaluation,

and then picks up the values from inside of the cycle iteratively. Thus, the gate attribute instance has a sequence of values indexed by the iterations. In this sense, the attribute value can be considered as a data stream. A problem with this solution is that it does not handle nested loops.

Several papers also point out that circular attributes can be evaluated by using incremental algorithms or tree transformations [Alb89][SEFR89][TW89][Jon90]. In [Kai89], circular attributes are defined by *propagated equations* as *histories* of programs, though the goal of this work is to extend attribute grammars to specify both static and dynamic semantics of programming languages using *action equations*. The corresponding attribute evaluator adopts an *event driven* method. For a cycle in a derivation tree, the evaluator propagates demands for circular attribute values repeatedly until the termination condition of the cycle is satisfied.

2.3 Indexical Programming

By indexical programming, we mean programming in a language that is at the same time a formal system based on indexical semantics [FW86]. In an indexical program, the value of an expression is an intension – a map from a universe of possible words, also called *a context space*, to a domain of values. Indexical languages provide context switching operators, also called indexical operators. The operators allow values from different contexts to be combined.

Lucid [WA85] was the first indexical functional language; it is a semantic enrichment of the functional language ISWIM [Lan66]. A Lucid program is an expression, and output of the program is the value of the expression. An expression can be a where-clause, consisting of a subject expression and a set of equations (or definitions). The equations in the where-clause define a local environment for the value of its subject. The value of a Lucid expression is an infinite stream of data items instead of a single one. The context space of Lucid consists of time points represented by natural numbers; each data item in a stream is indexed by a time point.

In Lucid, the original ISWIM operators are extended in a pointwise way: they perform the same operations on their operands at each time point. For example, consider the following expression

$$a + b.$$

Let a be the infinite stream $< 1, 3, 5, 7, \ldots >$ and b be the infinite stream $< 2, 4, 6, 8, \ldots >$. The operator "+" is extended pointwise on time points, so the result of the expression is the infinite stream $< 3, 7, 11, 15, \ldots >$ which is the infinite stream of the sums of the corresponding values of a and b at each point in time.

Lucid has three primitive indexical operators for context switching: **first**, **next**, and **fby**.

The unary operator **first** takes the first element from its operand stream and produces a constant stream. For example, let **x** be stream $< 1, 2, 3, \ldots >$, then `first x` is $< 1, 1, 1, \ldots >$.

The unary operator **next** removes the first element from its operand stream. For example, let **x** be stream $< 1, 2, 3, \ldots >$, then `next x` is $< 2, 3, 4, \ldots >$.

The binary operator **fby** (for "followed by") takes the first element of its first operand and appends the stream of its second operand to it. For example, let **x** be the stream $< 1, 3, 5, \ldots >$, and **y** be the stream $< 2, 4, 6, \ldots >$, then `x fby y` is $< 1, 2, 4, 6, \ldots >$. Using **fby** infinite data streams can be defined recursively. For example,

```
x
where
    x = 1 fby x + 1
end.
```

This program defines **x** (the output) to be the stream $< 1, 2, 3, \ldots >$. The first argument of **fby** is the initial value of the stream from which successive future values are to be generated. Note that since Lucid, as a functional language, is referentially transparent, the variable **x** denotes the same stream everywhere in the program. Thus, in the definition `x = 1 fby x + 1`, the **x** on the right hand side of the **fby** is the same as the one being defined which begins with 1. Since **x** is defined to be 1 at time point 0 and 1 is defined to

be 1 at time point 0, x's value at time point 1, which is 2, can be produced according to the definition from x + 1, and so on.

A Lucid program as a set of equations is really a textual form of a directed graph, which is called a dataflow graph or network. The nodes in the graph are the operations in the text and arcs are variables or expressions. For example, the following program

```
fib
where
    fib = 1 fby 1 fby fib + next fib
end.
```

is the textual form of the dataflow network in Figure 2.3.

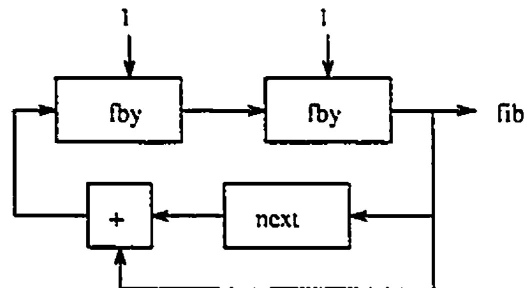

Figure 2.3: A dataflow network

Thus, every Lucid program has a unique representation as a network of operators. [AJ86] has termed such dataflow networks corresponding to Lucid programs *operator nets*.

The evaluation of Lucid programs is based on a computation model called *eduction* or *tagged demand driven dataflow* [FW86][AFH85]. Unlike data driven dataflow [Den80], eduction can be thought of as a form of dataflow with two-way traffic in communication lines. Data flows in one direction from producers to consumers in the usual way. In the other direction, demands are sent from consumers upstream to producers. Both demands and data are tagged with the stream indices or time points.

The computation of a Lucid program is demand driven: the value of the program's output, or the value of the defining expression, is evaluated at the time points that the user demands. The computation at any particular time point leads to the demands of values of various program variables. These variables may be defined in terms of context

switching operators that may demand variable values at time points different from the time point associated with the original demand.

In the next chapter, we define a new indexical functional language, which is also based on ISWIM, as an attribute definition language for attribute grammars.

Chapter 3

An Attribute Definition Language

3.1 Syntax

The *indexical functional attribute definition language* IFADL is a semantic enrichment of the functional language ISWIM [Lan66]. The syntax of IFADL and ISWIM is basically the same, except that IFADL has an additional set of primitive indexical operators. In other words, an IFADL program without indexical operators is syntactically an ISWIM program. The syntax of IFADL is given in Appendix A.

3.2 The Context Space

IFADL enriches ISWIM with indexical semantics. IFADL programs are defined on an implicit context space $\mathcal{C}$. The context space $\mathcal{C}$ is the product of a tree dimension, a multitime dimension, and an identifier dimension.

3.2.1 The tree dimension

The tree dimension $\mathcal{T}$ consists of all the nodes of an infinitely branching tree.

Definition 3.1 (tree dimension) *The tree dimension $\mathcal{T}$ is the set of all lists of natural numbers, i.e.*

$$\mathcal{T} = \bigcup_{i \in \omega} \omega^i$$

where ω is the set of natural numbers and ω^i is the Cartesian product of i copies of ω.

For the tree dimension $\mathcal{T}$, the ordering of the nodes in $\mathcal{T}$ is defined as follows.

Definition 3.2 (The ordering of node indices) *Let $i, j \in \mathcal{T}$. $i \sqsubset j$ if $i \neq j$ and*

1. $i = [\,]$ *or*

2. $last(i) \; < \; last(j)$ *or*

3. $last(i) \; = \; last(j) \; \wedge \; nolast(i) \sqsubset nolast(j)$
   where $nolast(list) = reverse(tail(reverse(list)))$.

Where [ ] is the empty list and *last*, *nolast*, *reverse*, and *tail* are functions. *last* takes a list l as its argument and returns the last element of l. *tail* takes a list l as its argument and returns l with the first element of l is removed. *reverse* takes a list l as its argument and returns a new list j. Here j has the same elements as l except that the elements are in reverse order. The above definition defines the depth-first ordering among nodes of $\mathcal{T}$.

Given a tree π that is a conventional ordered tree, we relate nodes of π to elements of $\mathcal{T}$ as follows.

1. The root node of π corresponds to $[\,] \in \mathcal{T}$.

2. Let n be a non-root node of π, m be the parent node of n in π, and n be the ith child node of m, n corresponds to $cons(i, s)$ where m corresponds to $s \in \mathcal{T}$.

3. T_π is the set of corresponding elements of π defined in 1 and 2.

where $cons(i, s)$ is a function with two arguments. The argument i is a natural number. The function returns a list which is the list s that results by inserting i at the beginning of s.

Definition 3.3 (Index tree) *A subset $S \subseteq T$ is an index tree if and only if*

1. $[\,] \in S \ \wedge\ \forall i \in S\ i \neq [\,]\ \rightarrow\ tail(i) \in S$.

2. $\forall i \in T\ \forall k \in \omega\ cons(k,i) \in S \rightarrow \forall c \leq k\ cons(c,i) \in S$

Remark 3.1 *T_π is an index tree if and only if $\exists S \subseteq T\ \wedge\ T_\pi = S$ and S is an index tree.*

The above definition states that: an index tree T_π must have at least one node – the root node and the indices of the child nodes must be consecutive, i.e.

$$\forall 0 \leq i \leq k-1,\ \text{cons}(i,n) \in T_\pi\ \wedge\ \forall i \geq k\ \text{cons}(i,n) \notin T_\pi$$

For example, Figure 3.1 shows the index tree corresponding to the parse tree for the binary string "10.01", whose grammar is defined in Example 2.1.

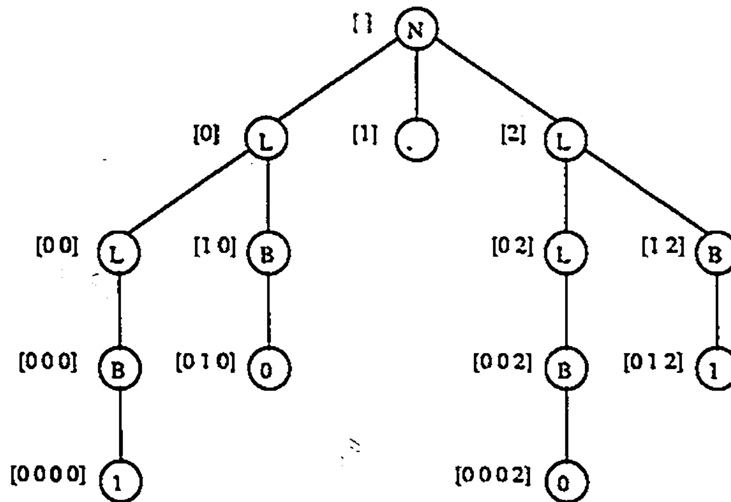

Figure 3.1: An indexed tree

3.2.2 The Multitime Dimension

Definition 3.4 (Multitime dimension) *The multitime dimension is the set*

$$\omega^\omega,$$

the Cartesian product of ω copies of the set ω of natural numbers.

Each time dimension consists of a sequence of time points which are represented by natural numbers. The multitime dimension is the set of all infinite sequences of natural numbers.

3.2.3 The Identifier Dimension

The identifier dimension $\mathcal{I}$ is the set of all identifiers. By an identifier we mean a string of characters that denotes a variable name in a program. In the identifier dimension, an identifier is called *an identifier point.*

Definition 3.5 (identifier dimension) *The identifier dimension $\mathcal{I}$ is defined by*

$$\mathcal{I} = \bigcup_{i \in \omega} \Sigma^i$$

where Σ is a predetermined set of characters.

3.2.4 The Context Space

The tree dimension $\mathcal{T}$, the multitime dimension ω^ω, and the identifier dimension $\mathcal{I}$ determine the context space $\mathcal{C}$.

Definition 3.6 (context space) *The context space $\mathcal{C}$ is defined as*

$$\mathcal{C} \; = \; \mathcal{T} \times \omega^\omega \times \mathcal{I}.$$

A context (n, t, i) in the context space $\mathcal{C}$ therefore consists of a node $n \in \mathcal{T}$, an infinite sequence of time points $t \; = < t_0, t_1, \ldots > \in \omega^\omega$ and an identifier point $i \in \mathcal{I}$.

3.3 Functional Semantics Without Indexical Operators

IFADL extends the functional semantics of ISWIM with indexical semantics based on the context space $\mathcal{C}$. The value of an IFADL expression is not a single value, it combines the values of the expression at all the contexts in $\mathcal{C}$. More precisely, an IFADL expression is a function from node $n \in \mathcal{T}$, time point $t \in \omega^\omega$, and identifier $i \in \mathcal{I}$ to the basic data

domain. Sometimes it may not be appropriate for an expression to have a value on all the contexts. IFADL provides a data value *eod* for those contexts at which no "real" data value is appropriate.

In IFADL, the context space is infinite. If a variable a is defined to have the value 5 at all the nodes of $\mathcal{T}$, to demand the value of a, an IFADL interpreter will produce an infinite output. However, practically, we may only need the values of a variable at certain nodes. For example, we may need the values of a at the nodes which are the first and second child of their parent nodes, and where the lengths of the indices are less then 3. In other words, we need the values of a at nodes: [ ], [0], [1], [00], [10], [01], [11]. To limit the evaluation to the above nodes, we define a to have the special data value *eod* at the rest of nodes in $\mathcal{T}$. When the interpreter evaluates a at a node n and detects that the value of a is *eod*, the interpreter will stop evaluation at all the nodes in the subtree rooted at n.

The special data value *eod* is very important for terminating the evaluation of an IFADL program over the infinite context space $\mathcal{C}$. However, this extension of the data domain forces us to modify the semantics of all operators of ISWIM over the original data domain to deal with combinations involving *eod* and "ordinary" data.

In this section, we define the semantics of extended ISWIM functions from the context space C to the extended data domain with the special value *eod*.

Let D be the original flat data domain of ISWIM and D_+ be D extended with the special data object *eod*. The semantics of an expression in IFADL is an intension defined as follows.

Definition 3.7 (Intension) *An intension is a function from $\mathcal{T} \times \omega^\omega \times \mathcal{I}$ to D_+.*

In the following descriptions, we use ID (for "Intensional Domain") to denote the set of all intensions, i.e. $\mathcal{T} \times \omega^\omega \times \mathcal{I} \to D_+$.

In the definition 3.1, $\mathcal{T}$ specifies the largest possible index tree. If an intension (or an IFADL expression) has non-*eod* values at all the nodes of $\mathcal{T}$, at a time point $t \in \omega^\omega$ and an identifier point $i \in \mathcal{I}$, we say that the intension has a *full tree shape* at (t, i). In practice, however, an intension in IFADL may only have non-*eod* values at a subset of $\mathcal{T}$.

We say that the set of nodes with non-*eod* values of an intension on $\mathcal{T}$ forms the *shape* of the intension at (t, i).

Definition 3.8 (Shape of an intension) *Given an intension $x \in ID$, a time point $t \in \omega^\omega$, and an identifier $i \in \mathcal{I}$, the shape $\hat{T}_x$ of x at (t, i), $\hat{T}_{x_{(t,i)}} \subseteq \mathcal{T}$, is defined by*

$$\hat{T}_{x_{(t,i)}} = \{n \in \mathcal{T} \mid x(n, t, i) \neq eod\}$$

where $x_{(t,i)} = \lambda n \in \mathcal{T}.x(n, t, i)$.

Definition 3.9 (Tree shape) *Given an intension $x \in ID$, a time point $t \in \omega^\omega$, and an identifier $i \in \mathcal{I}$, x has tree shape at (t, i) if and only if $\hat{T}_{x_{(t,i)}}$ is an index tree as defined by Definition 3.3.*

The non-*eod* values of an intension may not always form a tree shape. For example, if the value of an intension at the root node of a tree is defined as *eod*, but there are values which are not *eod* elsewhere, then the intension will not have tree shape.

IFADL extends the semantics of ISWIM functions in $D^k \to D$ $(k \geq 1)$ to functions in $ID^k \to ID$ in a pointwise way. In an intension, a pointwise ISWIM function applies to the value of the intension at every node, every time point, and every identifier in the context space without switching context.

In the following, we extend the semantics of ISWIM operators in IFADL in two steps. We first extend the definitions of all ISWIM functions that are functionally sequential [Yag84] to the extended data domain D_+. Then we define the meanings of the extended ISWIM operators over D_+ in terms of the indexical semantics of IFADL.

Extending strict functions from D to D_+ is simple. By a strict function we mean that it evaluates all its arguments. In other words, as long as one of the arguments of the function has a value $\perp$, the function has the value $\perp$. *eod* is defined in a similar way as $\perp$ in a strict function. That means that if one of the arguments of the function has a value *eod*, the function has the value *eod*, unless there is an argument with a value $\perp$. For non-strict functions, however, the extension will not be straightforward. For example, if the condition operand of the *if-then-else* operator has value *true*, the result value only

depends on the value of the *then* operand no matter what value the *else* operand has, even if it is $\perp$.

Before we extend ISWIM functions, first let us look at some examples of strict and non-strict functions.

The first example is $x + y$. $+$ is a strict function defined by:

$$+(x,y) = \begin{cases} x+y & (x \neq \perp) \wedge (y \neq \perp) \\ \perp & \text{otherwise} \end{cases}$$

The second example is $or(x,y)$, which is a non-strict function. $or(x,y)$ evaluates its arguments sequentially. *or* is defined by:

$$\text{or}(x,y) = \begin{cases} \textit{true} & \text{x = true} \\ \textit{true} & \text{(x = false)} \wedge \text{(y = true)} \\ \textit{false} & \text{(x = false)} \wedge \text{(y = false)} \\ \perp & \text{otherwise} \end{cases}$$

From the definition we see that, if the value of x (the first argument) is *true*, the result of the function is *true*. The result does not depend on the value of y, even if y is $\perp$. The function depends on the value of y only if x is *false*. On the other hand, if the value of x is $\perp$, the result is also $\perp$ independently of the value of y. Functions of this kind are called *functionally sequential functions*.

The third example is $paror(x,y)$, which is also a non-strict function, but its arguments are evaluated in parallel. *paror* is defined by:

$$\text{paror}(x,y) = \begin{cases} \textit{true} & \text{(x = true)} \vee \text{(y = true)} \\ \textit{false} & \text{(x = false)} \wedge \text{(y = false)} \\ \perp & \text{otherwise} \end{cases}$$

The two arguments of $paror(x,y)$ are evaluated in parallel. As long as one of the arguments has a *true* value, the result of the function is *true* even if the other argument is $\perp$. $paror(x,y)$ is not functionally sequential because a *true* value of either argument can determine the result of the function.

The following definitions extend ISWIM functions with $k \geq 0$ arguments from the extended domains D_+ to the extended domain D_+.

In order to define partial function applications (which have some but not all of the arguments instantiated) so that the no-strict case can be handled, in the following we first generalize the definition of what is called a k-ary function. In the definition, we consider that k is not necessarily a natural number. Instead, it denotes any set of mathematical objects, such as natural numbers. Each argument of a k-ary function is indexed by an object in the set denoted by k. Thus in a partial function application: given a set of indices and corresponding values, we can instantiate an arbitrary subset of arguments.

Definition 3.10 (K-ary function) *For any set k, a K-ary function is a map from D^k to D where $D^k = \{r \mid r : k \rightarrow D\}$*

Assume $D = \{t, f, \perp\}$ and $k = \{1, 2\}$, then

$$D^k = \{ \; <(1,t)(2,t)>, <(1,t)(2,f)>, <(1,t)(2,\perp)>, \\ <(1,f)(2,t)>, <(1,f)(2,f)>, <(1,f)(2,\perp)>, \\ <(1,\perp)(2,t)>, <(1,\perp)(2,f)>, <(1,\perp)(2,\perp)>\}$$

For a k-ary function, a subset of its arguments can be formally defined as follows.

Definition 3.11 (Partial elements of D^k) *Given $x \in D^k$ and $m \subseteq k$, the restriction $x|m$ of x to m is defined by*

$$x|m = \{(i,y) \in x \mid i \in m\}$$

Given a k-ary function f and a k-indexed set of arguments, the result of the function application $f(x)$ may depend on a subset of the arguments and be independent of others. In the following, we define a family of subsets of k, each of which determines a subset of x sufficient to determine the value of $f(x)$.

Definition 3.12 (Functional Dependency Set) *Given a k-ary function $f : D^k \rightarrow D$ and $x \in D^k$, the functional dependency set of f at x is a family of sets of indices defined by*

$$FDS(f,x) = \{I \subseteq k \mid \forall x' \in D^k \; ((x|I) = (x'|I)) \rightarrow f(x) = f(x')\}$$

The above definition says that a subset of k is a dependency set – a subset of the arguments that determine the result of the function application – if all the other arguments which are not in the dependency set are irrelevant to the result.

The following are some of the functional dependency sets for *or* and *paror*:

$$
\begin{array}{l}
FDS(or, <(1,t)(2,t)>) = \{\{1\}\{1,2\}\} \\
FDS(or, <(1,f)(2,t)>) = \{\{1,2\}\} \\
FDS(or, <(1,\bot)(2,f)>) = \{\{1\}\{1,2\}\} \\
FDS(paror, <(1,t)(2,t)>) = \{\{1\}\{2\}\{1,2\}\} \\
FDS(paror, <(1,f)(2,t)>) = \{\{2\}\{1,2\}\} \\
FDS(paror, <(1,\bot)(2,f)>) = \{\{1,2\}\}
\end{array}
$$

For all dependency sets of a function application, we are particularly interested in the intersection of the dependency sets, because, if it is not empty, each argument in the intersection is definitely depended on by the function application.

Definition 3.13 (Minimum functional dependency set of f at x) *Given a k-ary function $f : D^k \to D$ and $x \in D^k$, the minimum functional dependency set of f at x is defined by*

$$\mu FDS(f,x) = \bigcap_{I \in FDS(f,x)} I$$

Notice that $\mu FDS(f,x)$ is not necessarily in $FDS(f,x)$.

The following are some of the minimum functional dependency sets for *or* and *paror*:

$$
\begin{array}{l}
\mu FDS(or, <(1,t)(2,t)>) = \{1\} \\
\mu FDS(or, <(1,f)(2,t)>) = \{1,2\} \\
\mu FDS(or, <(1,\bot)(2,f)>) = \{1\} \\
\mu FDS(paror, <(1,t)(2,t)>) = \{\} \\
\mu FDS(paror, <(1,f)(2,t)>) = \{2\} \\
\mu FDS(paror, <(1,\bot)(2,f)>) = \{1,2\}
\end{array}
$$

The minimum dependency sets defined above are argument-dependent, that is, their constituents depend on a particular set of arguments. In the following, we define the argument-independent minimum dependency set of a function as the intersection of the argument-dependent minimum dependency sets for all possible sets of arguments.

Definition 3.14 (Minimum functional dependency set of f) *Given a k-ary func-*

tion $f : D^k \to D$, the minimum functional dependency set of f is defined by

$$mFDS(f) = \bigcap_{x \in D^k \wedge f(x) \neq \bot} \mu FDS(f, x)$$

The following are the minimum functional dependency sets of *or* and *paror*:

$$mFDS(or) = \{1\}$$
$$mFDS(paror) = \{\}$$

If *mFDS(f)* is not empty, each of its elements indicates an index of the argument list of the function such that the computation of the function does not converge if an argument at that index is not available. In other words, if the argument at that index is $\bot$, the function must have value $\bot$.

The following is the definition of what is called a partial function application, that is, a function applied to a subset of its argument list. The result of a partial function application is a function with the number of arguments less than the original one.

Definition 3.15 (Partial function application) *Given a k-ary function f, $m \subseteq k$, and $z \in D^m$, the partial application $f[z]$ of f to z is defined by*

$$f[z] : D^{k-m} \to D$$

$$f[z] = \lambda x \in D^{k-m}.f(z \cup x)$$

Lemma 3.1

$$\forall f \in D^k \to D \; \forall m \subset k \; \forall x \in D^k$$

$$f(x) = f[x|m](x|(k-m))$$

Proof: Directly from the definition. □

In the following, we define what is a functionally sequential function in terms of the minimum functional dependency set and partial function applications. The definition is equivalent to the definition given in [Yag84]. The functional sequentiality property of a function guarantees that the function is computable.

Definition 3.16 (Functionally sequential function) *Given a k-ary function f, f is functionally sequential if and only if*

- *f is constant, or*
- $mFDS(f) \neq \phi$ *and* $\forall x \in D^k\ f(x) \neq \bot \rightarrow f[x|mFDS(f)]$ *is functionally sequential.*

In the following, we complete the first step of the pointwise extension of ISWIM functions by extending the functions over the extended data domain D_+ with the special value *eod*.

Definition 3.17 (Extension of ISWIM functions) *Let* $f : D^k \rightarrow D$ *be a k-ary functionally sequential function, the extended function* $f_+ : (D_+)^k \rightarrow D_+$ *is defined by*

$$f_+(x) = \begin{cases} f(\{(i,\bot) | i \in k\}) & f \text{ is constant} \\ \bot & \exists i \in mFDS(f)\ x(i) = \bot \\ eod & \exists i \in mFDS(f)\ x(i) = eod \\ & \text{and } \forall j \in mFDS(f)\ x(j) \neq \bot \\ (f[x|mFDS(f)])_+(x|(k - mFDS(f))) & \text{otherwise} \end{cases}$$

Proposition 3.1 *Let* $f : D^k \rightarrow D$ *be a functionally sequential function. Then, the extended function* $f_+ : (D_+)^k \rightarrow D_+$ *of f is functionally sequential.*

Finally, we complete the second step of the pointwise extension of ISWIM functions by giving them indexical semantics.

Definition 3.18 (Interpretation of ISWIM operators in IFADL) *Let* I_{iswim} *be the interpretation of the language ISWIM. The semantics of a pointwise operator θ in IFADL is a function*

$$I_{ifadl}(\theta) : ID^k \rightarrow ID$$

such that

$$\forall w \in C \;\; I_{ifadl}(\theta)(e_1, \ldots, e_k)(w) \;=\; (I_{iswim}(\theta))_+(e_1(w), \ldots, e_k(w))$$

The above definition states that at a context in the context space, the result of a pointwise operation in IFADL is the result of the (extended) original ISWIM operation on the values of its operands at that context.

In IFADL, a constant in ISWIM is also extended pointwise to all contexts in the context space; it is an intension that has the same value at each context as the constant in ISWIM's data domain D. The definition of the semantics of a constant symbol in IFADL is given as follows.

Definition 3.19 *Let c be a constant symbol in both ISWIM and IFADL*

$$I_{iswim}(c) \in D.$$

The semantics of c in IFADL is the intension

$$I_{ifadl}(c) : \mathcal{T} \times \omega^{\omega} \times \mathcal{I} \rightarrow D$$

such that

$$\forall n \in \mathcal{T}\ \forall t \in \omega^{\omega}\ \forall i \in \mathcal{I}\ I_{ifadl}(c)(n, t, i) = I_{iswim}(c).$$

By the above definitions of general intensions and constants, there are two classes of intensions in IFADL. One is intensions that vary on the context space, and the other is constants that do not vary on the context space. By an intension having dimensionality we mean that there is a non-*eod* value of the intension at one context which is different from a non-*eod* value at another context in a dimension.

In the following, we define *dimensionality* and *constancy* of intensions to distinguish intensions in the two classes.

Definition 3.20 (Dimensionality/Constancy) *For $x \in ID$,*

1. *x has dimensionality in the tree dimension $\mathcal{T}$ if and only if*

$$\exists n, n' \in \mathcal{T}\ \exists t \in \omega^{\omega}\ \exists i \in \mathcal{I}\ x_{(n,t,i)} \neq x_{(n',t,i)}\ \wedge\ x_{(n,t,i)} \neq eod\ \wedge\ x_{(n',t,i)} \neq eod$$

otherwise x has constancy in $\mathcal{T}$.

2. *x has dimensionality in the multitime dimension ω^ω if and only if*

$$\exists n \in \mathcal{T}\ \exists t, t' \in \omega^\omega\ \exists i \in \mathcal{I}\ x_{(n,t,i)} \neq x_{(n,t',i)}\ \wedge\ x_{(n,t,i)} \neq cod\ \wedge\ x_{(n,t',i)} \neq cod$$

otherwise x has constancy in ω^ω.

3. *x has dimensionality in a time dimension $k \in \omega$ if and only if*

$$\exists n \in \mathcal{T}\ \exists t, t' \in \omega^\omega\ \exists i \in \mathcal{I}\ (\forall j \in \omega\ (j \neq k) \rightarrow (t_j = t'_j)) \wedge$$
$$x_{(n,t,i)} \neq x_{(n,t',i)}\ \wedge\ x_{(n,t,i)} \neq cod\ \wedge\ x_{(n,t',i)} \neq cod$$

otherwise x has constancy in the time dimension k of the list of time dimensions ω^ω.

4. *x has dimensionality in the identifier dimension $\mathcal{I}$ if and only if*

$$\exists n \in \mathcal{T}\ \exists t \in \omega^\omega\ \exists i, i' \in \mathcal{I}\ x_{(n,t,i)} \neq x_{(n,t,i')}\ \wedge\ x_{(n,t,i)} \neq cod\ \wedge\ x_{(n,t,i')} \neq cod$$

otherwise x has constancy in $\mathcal{T}$.

5. *x has dimensionality in $\mathcal{C}$ if and only if*

   - *x has dimensionality in the tree dimension $\mathcal{T}$, or*
   - *x has dimensionality in the time dimensions ω^ω, or*
   - *x has dimensionality in the identifier dimension $\mathcal{I}$.*

   otherwise x has constancy in $\mathcal{C}$.

3.4 Semantics of Indexical Operators

IFADL provides a set of primitive indexical operators for switching contexts in the context space $\mathcal{C}$. These indexical operators can be further divided into four classes: node switching operators, time switching operators, and identifier switching operators. The operators in each class manipulate the values of intensions in one dimension.

3.4.1 Node Switching Operators

There exist one indexical variable *index* and four primitive node switching operators in IFADL: *root*, *parent*, *nextsib*, and *child*. In the following definitions, given an intension $x \in$ ID, a node $n \in \mathcal{T}$, a time point $t \in \omega^{\omega}$, and an identifier point $i \in \mathcal{I}$, we use the notation $x_{(n,t,i)}$ to denote the value of x at context (n,t,i).

At context (n,t,i), the value of **index** is the index of node $n \in \mathcal{T}$, which is a list. The semantics of *index* is defined as follows.

Definition 3.21 *The indexical variable* **index**

$$index : ID$$

is defined by:

$$index = \lambda(n,t,i).\ n$$

The unary operator **root** switches context from any node $n \in \mathcal{T}$ to the root node of $\mathcal{T}$. Let x be an intension. The meaning of *root* x at all contexts is the value of x at the root node at the same time points and identifier point. The semantics of *root* is defined as follows.

Definition 3.22 *The operator* root *denotes a function*

$$root : ID \rightarrow ID$$

defined by:

$$root = \lambda x \lambda(n,t,i).\ x_{([\,],t,i)}$$

The unary operator **parent** switches context from a node to its parent node. Let x be an intension. The meaning of *parent* x at (n,t,i) is the value of x at n's parent node at the same time points and identifier point. The semantics of *parent* is defined as follows.

Definition 3.23 *The operator* **parent** *denotes a function*

$$parent : ID \rightarrow ID$$

defined by

$$parent = \lambda x \lambda(n,t,i).x_{(upnode(n),t,i)}$$

where

$$upnode(n) = \begin{cases} [\,] & if\ n = [\,] \\ tail(n) & otherwise \end{cases}$$

The unary operator **nextsib** switches context from a node to its next sibling node. Let x be an intension, the meaning of *nextsib* x at (n,t,i) is the value of x at n's next sibling node at the same time points and identifier point. The semantics of *nextsib* is defined as follows.

Definition 3.24 *The operator* **nextsib** *denotes a function*

$$nextsib: ID \rightarrow ID$$

defined by

$$nextsib = \lambda x \lambda(n,t,i).x_{(m,t,i)}$$

where $m = cons(head(n)+1, tail(n))$

The binary operator **child** switches context from a node to one of its child nodes. Let x and k be intensions where k has integer type, the meaning of child(x,k) at context (n,t,i) is the value of x at the kth child of n at the same time points and identifier point. The semantics of *child* is defined as follows.

Definition 3.25 *The operator* **child** *denotes a function*

$$child: ID \times ID \rightarrow ID$$

defined by:

$$child = \lambda x \lambda k \lambda(n,t,i).\ x_{(cons(k_{(n,t,i)},n),t,i)}$$

3.4.2 Time Switching Operators

The time switching operators of IFADL are extensions of the operators of Lucid [WA85]. IFADL has five primitive time switching operators: *first*, *next*, *fby*, *branch*, and *contemp*. Time switching operators switch a context $(n, < t_0, t_1, \ldots >, i)$ to another context $(n, < t'_0, t_1, \ldots >, i)$. The functions of the operators are the same as in Lucid except that they work only in the first time dimension of the context space $\mathcal{C}$.

The unary operator **first** switches context from any time point $< t_0, t_1, \ldots >$ to the time point $< 0, t_1, \ldots >$ in the first time dimension, at a node $n \in \mathcal{T}$, at an identifier point $i \in \mathcal{I}$. Given an intension $x \in ID$, the result of *first* x at context $(n, < t_0, t_1, \ldots >, i)$ returns the value of x at time points $< 0, t_1, \ldots >$ at the same node and identifier point. The formal semantics of *first* is defined as follows.

Definition 3.26 *The operator* **first** *denotes a function*

$$first : ID \rightarrow ID$$

defined by:

$$first = \lambda x \lambda(n, < t_0, t_1, \ldots >, i).\ x_{(n, <0, t_1, \ldots>, i)}$$

The unary operator **next** switches context from a time point $< t_0, t_1, \ldots >$ to the time point $< t_0 + 1, t_1, \ldots >$ at a node n at an identifier point i. Given an intension $x \in ID$, the result of *next* x at context $(n, < t_0, t_1, \ldots >, i)$ returns the value of x at at time points $< t_0 + 1, t_1, \ldots >$ at the same node and identifier point. The formal semantics of *next* is defined as follows.

Definition 3.27 *The operator* **next** *denotes a function*

$$next : ID \rightarrow ID$$

defined by:

$$next = \lambda x \lambda(n, < t_0, t_1, \ldots >, i).\ x_{(n, <t_0+1, t_1, \ldots>, i)}$$

At a node n and an identifier point i, the binary operator **fby** has two operands, say x and y, which are sequences of data values indexed by time points.

$$x_{(n,<0,t_1,...>,i)}, x_{(n,<1,t_1,...>,i)}, \dots$$

$$y_{(n,<0,t_1,...>,i)}, y_{(n,<1,t_1,...>,i)}, \dots$$

The result of x *fby* y at a node n, at an identifier i returns a sequence of values:

$x_{(n,<0,t_1,...>,i)}, y_{(n,<0,t_1,...>,i)}, y_{(n,<1,t_1,...>,i)}, y_{(n,<2,t_1,...>,i)}, \dots$ at n and i. The formal semantics of *fby* is defined as follows.

Definition 3.28 *The operator* **fby** *denotes a function*

$$fby: ID \times ID \rightarrow ID$$

defined by:

$$fby = \lambda x \lambda y \lambda(n, < t_0, t_1, \dots >, i). \begin{cases} x_{(n,<0,t_1,...>,i)} & t_0 = 0 \\ y_{(n,<t_0-1,t_1,...>,i)} & t_0 > 0 \end{cases}$$

In order to switch context in the multitime dimension, we define two more time switching operators in IFADL, *branch*, and *contemp*.

The unary operator **branch** is the combination of Lucid operators *active* and *current* (*branch* $x =$ *current active* x). The formal semantics of *branch* is defined as follows.

Definition 3.29 *The operator* **branch** *denotes a function*

$$branch: ID \rightarrow ID$$

defined by:

$$branch = \lambda x \lambda(n, < t_0, t_1, t_2, \dots >, i).\ x_{(n,<t_1,t_2,...>,i)}$$

Informally, considering x to be a data stream in time dimension 0

$$x = < x_0, x_1, x_2, \dots >_0$$

branch x returns a new data stream x' in time dimension 1

$$x' = << x_0, x_0, x_0, \ldots >_0, < x_1, x_1, x_1, \ldots >_0, < x_2, x_2, x_2, \ldots >_0, \ldots >_1$$

where the value of x' at each time point i in time dimension 1 is a constant data stream of x_i in time dimension 0.

The formal semantics of the unary operator **contemp** is defined as follows.

Definition 3.30 *The operator* **contemp** *denotes a function*

$$contemp: ID \rightarrow ID$$

defined by:

$$contemp = \lambda x \lambda(n, < t_0, t_1, \ldots >, i).\ x_{(n, < t_0, t_0, t_1, \ldots >, i)}$$

The operator *contemp* can also be understood in terms of double streams. Informally speaking, it collapses a double sequence into a single stream by diagonalization. Considering x to be a double stream in time dimension 1 (outer) and 0 (inner)

$$x = << a_0, a_1, a_2, \ldots >_0, < b_0, b_1, b_2, \ldots >_0, < c_0, c_1, c_2, \ldots >_0, \ldots >_1$$

The operation *contemp* x reduces x into a single data stream in time dimension 0

$$< a_0, b_1, c_2, \ldots >_0 .$$

3.4.3 Identifier Switching Operators

To manipulate aggregate values in the identifier dimension, IFADL defines one indexical variable *idindex* and one primitive indexical operator *ati*.

At a context (n, t, i), the value of **idindex** is the identifier index i, which is a character string. The formal semantics of *idindex* is defined as follows.

Definition 3.31 *The indexical variable* **idindex**

$$idindex: ID$$

is defined by:

$$idindex = \lambda(n, t, i).\ i$$

The binary identifier switching operator **ati** (for *at identifier*) switches context from an identifier point to another in the identifier dimension. At a context (n, t, i), the operator x *ati* y switches to the context whose identifier point is the same as the value of y at the current context and returns the value of x at that context. The formal semantics of *ati* is defined as follows.

Definition 3.32 *The operator* **ati** *denotes a function*

$$ati: ID \times ID \rightarrow ID$$

defined by:

$$ati = \lambda x.\lambda y.\lambda(n, t, i).\ x_{(n,t,i')}$$

where $i' = y_{(n,t,i)}$.

The function of the operator *ati* can be thought as selecting an element from a symbol table.

3.5 Non-primitive Indexical Operators

In the following, using the above defined primitive indexical operators, we also define several non-primitive indexical operators in IFADL, which we will use in later chapters.

We first define four non-primitive node switching operators *atn*, *isroot*, *ischild*, and *sibling*.

Given two intensions x and n where n has list type and is constant in $\mathcal{T}$, the binary node switching operator x **atn** n switches context from any node to the node whose index is equal to n. The operator returns the value of x at n. *atn* does not switch context in the time dimension or the identifier dimension. The following is the definition of *atn* in terms of the primitive node switching operators.

```
x atn n = root move(x, n)
  where
      move(x, n) = if n eq nil then x
                         else child(move(x, nolast(n)), last(n)) fi;
  end;
```

Note that the constancy requirement for n is to guarantee that the level-by-level context switching on the tree always refers to the same node n.

To complete, we also give the formal semantics of *atn*.

$$atn: ID \times TN \rightarrow ID$$

where $TN = \mathcal{T} \times \omega^{\omega} \times \mathcal{I} \rightarrow \mathcal{T}$.

$$\text{atn} = \lambda x \lambda y \lambda(n, t, i).\ x_{(n', t, i)}$$

where $n' = y_{(n,t,i)}$.

When n is constant in the tree dimension, the expression x *atn* n has the same value at every node at some time and identifier points; it is equal to x's value at node n at the times and identifier. In other words, if n is constant in $\mathcal{T}$, so is x *atn* n.

Using the primitive node switching operators, we can also define the following non-primitive indexical operators.

$$\begin{aligned} isroot &= index\ eq\ [\ ]; \\ ischild(k) &= head(index)\ eq\ k; \\ sibling(x, k) &= parent\ child(x, k); \end{aligned}$$

The binary operator **sibling** switches context from a node to one of its sibling node and returns the value of x at that node. Given two nodes $n, m \in \mathcal{T}$ where m is n's parent node, an intension x, and an integer k, sibling(x, k) returns the value of x at m's k^{th} child node.

In the following, we define a non-primitive time switching operator **asa**. Given two intensions x and y, at time points $t = \langle t_0, t_1, \ldots \rangle$, the binary time switching operator x *asa* y switches context to the time point $t' = \langle t'_0, t_1, \ldots \rangle$ at which y has the first *true* value since the initial time (time 0) in the time dimension 0. The operator returns the value of x at t', or $\perp$ if there is no such t'. *asa* does not switch context in the tree and identifier dimensions. The following is the definition of *asa* in terms of the primitive time switching operators.

```
x asa y = if first y
          then first x
          else ((next x) asa (next y));
```

To complete, we also give the formal semantics of *asa*.

$$\text{asa}: \mathbb{D} \times \mathbb{B} \rightarrow \mathbb{D}$$

where $\mathbb{B} = \mathcal{T} \times \omega^{\omega} \times \mathcal{I} \rightarrow \{true, false, eod\}$.

$$\text{asa} = \lambda x \lambda y \lambda (n, < t_0, t_1, \ldots >, i).$$
$$\begin{cases} x_{(n,<t_0',t_1,\ldots>,i)} & \exists t'' \in \omega \; y_{(n,<t_0'',t_1,\ldots>,i)} \\ \perp & \text{otherwise} \end{cases}$$

where

$$t_0' = \min\{t_0'' \in \omega \mid y_{(n,<t_0'',t_1,\ldots>,i)}\}$$

A given boolean expression with value over $\mathcal{T}$ may have a tree shape at a given time and identifier point. Sometimes we need to know whether the expression's value is true at each point on the tree shape. In the following, we define a special indexical operator **globally**. The operator returns *true* if the value of its argument x is *true* at each point on the tree shape. Otherwise the operator returns *false*. It is defined in terms of the primitive node switching operators.

```
globally(x) = (x eq true) and
    child(((x eq eod) or (globally(x) eq true)) and
    ((nextsib(x) eq eod) or (globally(nextsib(x)) eq true)),0);
```

Using operator **globally**, we can define a node-time switching operator **tasa** (for *tree as soon as*) which switches context in the tree dimension and time dimension. Given intensions x and y, where we consider y as a sequence of boolean-valued trees, the binary operator x *tasa* y returns the value of x at each node at the time at which y first time has *true* value at *every* node on its own tree shape. Externally *tasa* only switches context in the time dimension, but internally it also switches context in the tree dimension to aggregate local conditions at nodes.

The operator *tasa* can be defined as

```
x tasa p = x asa globally p
```

Notice that *tasa* actually defines a "local" global termination condition for each subtree of an index tree.

Finally, we define a node-identifier switching operator agg (for *aggregate*). Given an intension x, an identifier intension id that is constant in the identifier dimension, and an intension y that gives a tree shape, at context (n, t, i) where n is in the index tree $T_y(t, i)$, the function $agg(x, id, y)$ scans the subtree rooted by n in the depth-first order. It returns the value of x at context(n', t, i) where n' is the first node encountered in the search such that at context(n', t, i) the identifier point i is equal to the value of id at the same context.

The agg is a function with type

$$agg:\ ID\ \times\ LEX\ \times ID\ \rightarrow\ ID$$

where $LEX = \mathcal{T} \times \omega^{\omega} \times \mathcal{T} \rightarrow \Sigma$. agg is defined by the primitive indexical operators as follows.

```
agg(x, id, tree) = element
    where
    element = if iseod(tree) then eod
              elseif idindex eq id then x
              elseif not iseod(k) then k
              else child(element, head(index)+1)
              fi
              where
                k = child(element,0);
              end;
    end;
```

The formal indexical semantics of agg is

$$agg\ =\ \lambda x\ \lambda id\ \lambda y.\ \lambda(n, t, i).\ \begin{cases} x_{(n',t,i)} & \exists n'' \in \hat{T}_y(n, t, i)\ id_{(n'',t,i)} = i \\ \perp & otherwise \end{cases}$$

where

$$n' = min\{n'' \in \hat{T}_y(n, t, i) \mid id_{(n'',t,i)} = i\}$$

where min is based on the ordering of nodes in $\mathcal{T}$ defined in Definition 3.2.

At a node on the index tree of T_{tree}, the value of an intension defined by *agg(x, id, tree)* can be thought as a symbol table, where *id* denotes a table entry, x denotes the information for each entry, and *tree* denotes the domain of the aggregation.

3.6 Denotational Semantics of IFADL

To define denotational semantics of IFADL, we first define the environment of IFADL.

The environment *env* in IFADL is defined as

$$\text{env} : \text{Env} \; = \; V - \bigcup_{k \in \omega} (\text{ID}^k - \text{ID})$$

where V is the set of variable symbols in the program.

The semantic function $[\![\;]\!]$ from expressions to their values has the type

$$\text{Exp} - \text{Env} - \text{ID}$$

In the following denotational semantics, θ_i is an i-ary pointwise operator. For simplicity, we assume that all user-defined functions are unary.

$$[\![< \text{constant} >]\!] \; \text{env} \; = \; \lambda(n,t,i).\ I_{\text{ifadl}}(< \text{constant} >)$$

$$[\![< \text{identifier} >]\!] \; \text{env} \; = \; \lambda(n,t,i).\ \text{env}(< \text{identifier} >)(n,t,i)$$

$$[\![\theta_1 < \text{exp} >]\!] \; \text{env} \; = \; \lambda(n,t,i).\ I_{\text{ifadl}}(\theta_1)([\![< \text{exp} >]\!] \; \text{env}) \; (n,t,i)$$

$$[\![< \text{exp}_1 > \theta_2 < \text{exp}_2 >]\!] \; \text{env} \; =$$

$$\lambda(n,t,i).\ I_{\text{ifadl}}(\theta_2)(([\![< \text{exp}_1 >]\!] \; \text{env}), ([\![< \text{exp}_2 >]\!] \; \text{env})) \; (n,t,i)$$

$$[\![\text{if} \; < \text{exp}_0 > \; \text{then} \; < \text{exp}_1 > \; \text{else} \; < \text{exp}_2 > \; \text{fi}]\!] \; \text{env} \; =$$

$$\begin{aligned} \lambda(n,t,i).\ \text{if} \; & [\![< \text{exp}_0 >]\!] \; \text{env} \; (n,t,i) \\ & \text{then} \; [\![< \text{exp}_1 >]\!] \; \text{env} \; (n,t,i) \\ & \text{else} \; [\![< \text{exp}_2 >]\!] \; \text{env} \; (n,t,i) \end{aligned}$$

$$\begin{aligned} [\![\text{case} \; < \text{exp}_0 > \; \text{of} \; & < \text{constant}_{1_1} > \ldots < \text{constant}_{1_j} >:< \text{exp}_1 >; \ldots; \\ & < \text{constant}_{k_1} > \ldots < \text{constant}_{k_j} >:< \text{exp}_k >; \ldots; \\ & \text{default} :< \text{exp} >; \; \text{end}]\!] \; \text{env} \; = \end{aligned}$$

$$\begin{aligned} \lambda(n,t,i).\text{if} \; & [\![< \text{exp}_0 >]\!] \; \text{env}(n,t,i) = I_{\text{ifdal}}(< \text{constant}_{j_k} >) \\ & \text{then} \; [\![< \text{exp}_j >]\!] \; \text{env} \; (n,t,i) \end{aligned}$$

$$\text{else } [\![< \text{exp} >]\!] \text{ env } (n, t, i)$$

$$[\![\text{index}]\!] \text{ env } = \lambda(n, t, i).\ n$$

$$[\![\text{root } < \text{exp} >]\!] \text{ env } = \lambda(n, t, i).\ [\![< \text{exp} >]\!] \text{ env } ([\,], t, i)$$

$$[\![\text{parent } < \text{exp} >]\!] \text{ env } = \lambda(n, t, i).\ [\![< \text{exp} >]\!] \text{ env } (\text{upnode}(n), t, i)$$

$$\text{where upnode}(n) = \begin{cases} [\,] & \text{if } n = [\,] \\ \text{tail}(n) & \text{otherwise} \end{cases}$$

$$[\![\text{nextsib } < \text{exp} >]\!] \text{ env } = \lambda(n, t, i).\ [\![< \text{exp} >]\!] \text{ env } (\text{nextnode}(n), t, i)$$

$$\text{where nextnode}(n) = \text{cons}(\text{head}(n)+1, \text{tail}(n))$$

$$[\![\text{child}(< \text{exp}_1 > < \text{exp}_2 >)]\!] \text{ env } =$$

$$\lambda(n, t, i).\ [\![< exp_1 >]\!] \text{ env } (\text{cons}(([\![< \text{exp}_2 >]\!] \text{ env } (n, t, i)), n), t, i)$$

$$[\![\text{first } < exp >]\!] \text{ env } = \lambda(n, < t_0, t_1, \ldots >, i).\ [\![< \text{exp} >]\!] \text{ env } (n, < 0, t_1, \ldots >, i)$$

$$[\![\text{next } < \text{exp} >]\!] \text{ env } = \lambda(n, < t_0, t_1, \ldots >, i).\ [\![< \text{exp} >]\!] \text{ env } (n, < t_0 + 1, t_1, \ldots >, i)$$

$$[\![< \text{exp}_1 > \text{ fby } < \text{exp}_2 >]\!] \text{ env } =$$

$$\lambda(n, < t_0, t_1, \ldots >, i) \begin{cases} [\![< \text{exp}_1 >]\!] \text{ env } (n, < 0, t_1, \ldots >, i) & t = 0 \\ [\![< \text{exp}_2 >]\!] \text{ env } (n, < t_0 - 1, t_1, \ldots >, i) & t > 0 \end{cases}$$

$$[\![branch < exp >]\!]\ env = \lambda(n, < t_0, t_1, \ldots >, i).[\![< exp >]\!]\ env\ (n, < t_1, t_2, \ldots >, i)$$

$$[\![contemp < exp >]\!]\ env = \lambda(n, < t_0, t_1, \ldots >, i).$$

$$[\![< exp >]\!]\ env\ (n, < t_0, t_0, t_1, \ldots >, i)$$

$$[\![\text{idindex}]\!] \text{ env } = \lambda(n, t, i).\ i$$

$$[\![< \text{exp}_1 > \text{ ati } < \text{exp}_2 >]\!] \text{ env } =$$

$$\lambda(n, t, i).[\![< \text{exp}_1 >]\!] \text{ env } (n, t, ([\![< \text{exp}_2 >]\!] \text{ env } (n, t, i)))$$

where $p =< exp >$ env.

$$[\![< \text{identifier} > < \text{exp} >]\!] \text{ env } =$$

$$\lambda(n, t, i).\ (([\![< \text{identifier} >]\!] \text{ env})([\![< \text{exp} >]\!] \text{ env}))\ (n, t, i)$$

$$[\![< \text{exp} > \text{where} \ldots; f_i =< \text{exp}_i >; \ldots; f_j(x) =< \text{exp}_j >; \ldots; \text{end}]\!] \text{ env} =$$

$$\lambda(n, t, i).\ [\![< \text{exp} >]\!] \text{ env}'\ (n, t, i)$$

$$\text{where env}' = [\ldots\ [\![< \text{exp}_i >]\!] \text{ env}'/f_i\ \ldots\ \lambda x.[\![< \text{exp}_j >]\!] \text{ env}'/f_j\ \ldots] \text{ env}.$$

In the above semantic definitions, the notation $[v/x]$env denotes an environment in *Env* just like *env* except that for the variable symbol x it returns the intension v.

In the next chapter, we describe how to use IFADL for programming.

Chapter 4

Programming in IFADL

In this Chapter, we treat IFADL as an independent indexical functional language and show how to program in IFADL, through examples. Since the identifier dimension and its associated context switching operators are specially designed for expressing aggregate attributes, we will leave the discussion about how to program in the identifier dimension to later chapters.

4.1 Pointwise Operations

An IFADL program is an IFADL expression. An IFADL expression without indexical operators performs the same operations at all points in the context space $\mathcal{C}$. The dimensionality of a program without indexical operators depends only on the dimensionality of its input data. The following are two examples of IFADL expressions which specify algorithms which operate on trees.

We are given two input variables, A and B, which are constant in the multitime dimension ω^{ω} and identifier dimension $\mathcal{I}$. Let the values of A and B have tree shapes $\hat{T}_A$ and $\hat{T}_B$ shown in Figure 4.1, that it follows from the semantics given in chapter 3.

The value of the IFADL expression

$$A + B$$

has the shape of the intersection of tree $\hat{T}_A$ and tree $\hat{T}_B$. That is, the expression $A + B$

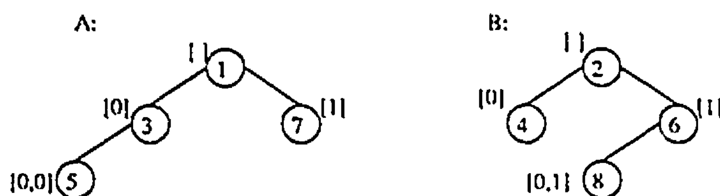

Figure 4.1: The values of A and B

has a tree shape which is a subtree of $\hat{T}_A$ and $\hat{T}_B$. For example, given two variables A and B which have the input values shown in Figure 4.1, the output of the expression $A + B$ is an indextree which is the intersection of $\hat{T}_A$ and $\hat{T}_B$.

Since A and B both have non-*eod* values at the node [ ], when the value of $A + B$ is demanded, the IFADL interpreter performs an addition operation and returns the value 3 at [ ]. Similarly, the expression $A + B$ will have values 7 and 13 at the nodes [0] and [1], respectively. At the nodes [0,0] and [0,1], since one of the operands has an *eod* value, according to the semantics of +, the expression $A + B$ will have *eod* values at both of the nodes. Since A and B have *eod* values at the rest of the nodes in T, the results of $A + B$ will be *eod* at these nodes if their values are demanded. The result of the expression $A + B$ for the given inputs of A and B is shown in Figure 4.2.

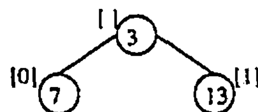

Figure 4.2: The value of $A + B$

For the same inputs of A and B, the following expression has the tree shape of the union of tree $\hat{T}_A$ and tree $\hat{T}_B$, with the sum of the values of A and B at each common node.

```
if (not iseod(A)) and (not iseod(B)) then A + B
elseif (not iseod(A)) then A
else B fi.
```

The operation + is only performed at the nodes on which both A and B have non-*eod* values. For the nodes on which there is only one non-*eod* value, the expression at the

nodes will simply be a copy of the non-*eod* value. The result of the above expression is shown in Figure 4.3.

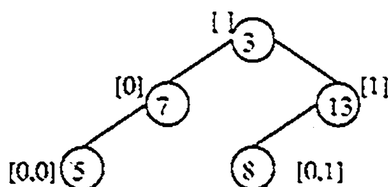

Figure 4.3: The union-tree version of $A + B$

4.2 Expressing Algorithms Operating on Trees

Using indexical operators, algorithms which operate on trees can be expressed in IFADL. In the following, we give three program examples which use node switching operators.

The first program builds a binary search tree from a given list of integers. The program is an expression. The expression is defined at all the nodes in $\mathcal{T}$. At the root node, the program reads the input list and keeps the first element of the list at the root node. The program also divides the list into two sublists *less* and *greater*. The sublist *less* contains all the elements in the list which are smaller than or equal to the first element. The sublist *greater* contains all the elements in the list which are greater than the first element. At the remaining nodes, if a node is its parent's first child, it will get the *less* list from its parent node if *less* is not an empty list. If a node is its parent's second child, it will get the *greater* list from its parent node if *greater* is not an empty list. The nodes which are the first or second child of their parent will perform the same operation as they performed at the root node. If a node is neither first child nor second child, it will get an *eod* as its input. After the evaluation, the expression has a tree shape, which is a binary search tree.

In the program, we define two variables *bstree* and *partial-list*. The value of *partial-list* at the root node is the input list. The value of *partial-list* at a left or right child node is a sublist of its parent's partial list that contains those elements smaller or greater than the head of the partial list at its parent node. The value of *bstree* at each node is the head of

the partial list at that node.

Example 4.1 (Binary search tree)

```
bstree
where
    bstree = if partial-list eq nil then eod
             else head(partial-list) fi;

    partial-list = if isroot then input-list
                   elseif parent partial-list eq nil then nil
                   elseif islchild then less(parent bstree,
                                            tail(parent partial-list))
                   elseif isrchild then greater(parent bstree,
                                            tail(parent partial-list))
                   else nil fi;

    less(x, list) = if list eq nil then nil
                    elseif x >= head(list)
                        then cons(head(list), less(x, tail(list)))
                        else less(x, tail(list)) fi;

    greater(x, list) = if list eq nil then nil
                    elseif x < head(list) then cons(head(list),
                                                 greater(x, tail(list)))
                    else greater(x, tail(list)) fi;

    islchild = ischild(0)
    isrchild = ischild(1)
end.
```

Figure 4.4 shows the binary search tree with the value of *partial-list* at each node. when *input-list* = [5 2 7 3 6 9 8 1 4]. The values of *bstree* are shown inside each node and *pl* means *partial-list*.

By adding a definition, we can easily convert the above program into a quicksort program. The following is the modified code.

Figure 4.4: The binary search tree for input list [5 2 7 3 6 9 8 1 4]

Example 4.2 (Quicksort)

```
root sorted-list
where
    bstree = if partial-list eq nil then eod
                 else head(partial-list) fi;

    partial-list = if isroot then input-list
                   elseif parent partial-list eq nil then nil
                   elseif islchild then less(parent bstree,
                                             tail(parent partial-list))
                   elseif isrchild then greater(parent bstree,
                                             tail(parent partial-list))
                   else nil fi;

    sorted-list = if iseod(bstree) then nil
                    else append(lchild(sorted-list),
                                cons(bstree, rchild(sorted-list))) fi;

    less(x, list) = if list eq nil then nil
                    elseif x >= head(list)
                        then cons(head(list), less(x, tail(list)))
                        else less(x, tail(list)) fi;

    greater(x, list) = if list eq nil then nil
                    elseif x < head(list) then cons(head(list),
                                                   greater(x, tail(list)))
```

```
                    else greater(x, tail(list)) fi;

    islchild = ischild(0)
    isrchild = ischild(1)
    lchild(x) = child(x, 0);
    rchild(x) = child(x, 1);
end.
```

The above programs built binary search trees from inputs. We can also define an IFADL program on a given tree. The following example evaluates the decimal value of the binary number "10.01". The program is defined on the index tree (Figure 3.1) which corresponds to the parse tree of the binary string "10.01". The variables in the program can be understood as the attributes defined in Example 2.1 in Chapter 2. The decimal value of "10.01" is the value of the variable *value* at the root node.

Example 4.3 (Binary string)

```
root value
where
    value = case index of
            []: child(value, 0) + child(value, 2);
            [0],[2]: child(value, 0) + child(value, 1);
            [0 0],[0 2]: child(value, 0);
            [1 2],[0 0 0]: 2 ** scale;
            [0 2],[0 0 2]: 0;
            default: eod;
            end;
    scale = case index of
            [0]: 0;
            [1]: -1;
            [0 0]: (parent scale) + 1;
            [1 2],[0 0 0]: parent scale;
            default: eod;
            end;
    length = case index of
             [2]: child(length, 0) + 1;
             [0 2]: 1;
             default: eod;
             end;
end.
```

The above program is not easy to understand since there are many indices in the case expressions. In the next two chapters, we will see how the problem can be solved by using IFADL to define attributes in indexical attribute grammars.

4.3 Expressing Iterative Algorithms Operating on Trees

We can express iterative algorithms on trees in IFADL using the multitime dimension to define sequences of trees. If an IFADL variable is sensitive to time, or it is a temporal variable, then the variable is not a constant over the multitime dimension. In this case, the variable at different time points may have different values. The values of the variable may form a tree shape at a particular time point. We can consider an intension in IFADL as a sequence of trees indexed by time points.

Given an iterative algorithm, we let each iteration step correspond to a time point in the first time dimension; the initial step corresponds to time 0. Note that here we only consider the first time dimension and assume that in all other time dimensions the intension is constant. At the initial iteration step, we define an initial tree. At each of the following steps, a new tree is defined, according to the algorithm, in terms of the trees defined at previous steps (time points). The iteration ends at a time point at which the current tree satisfies the termination condition described in the algorithm. The final value of the iteration is the tree at the ending time point.

For example, Figure 4.5 shows a graph and a recursive algorithm for calculating the values on the nodes of the graph [WJ88]. The evaluation starts by giving an initial value to each node, and terminates when all the values of g, a, b, and c become stable.

The above problem can be easily translated into an IFADL program. First, we define the graph as a tree with four nodes (Figure 4.6). The node g is the root node indexed by [-] of the tree. g has three children a, b, and c indexed by [1], [2], and [3], respectively. The following IFADL program specifies the recursive data dependency among the nodes of the tree. The termination condition is also specified explicitly in the program.

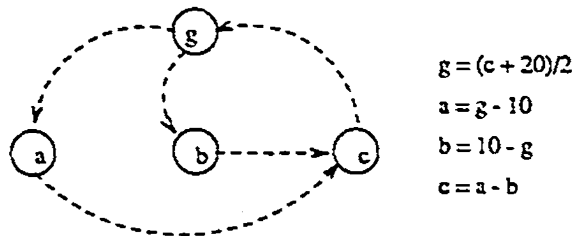

Figure 4.5: A circular dependency graph

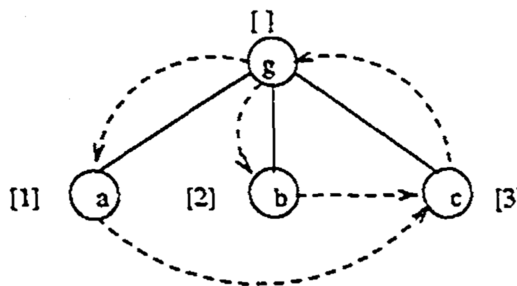

Figure 4.6: A tree-structured circular dependency graph

Example 4.4

```
root (a asa ending)
where
    a = case index of
        [ ]:  init fby (child(a, 2) + 20)/2;
        [0]: init fby ((parent a) - 10);
        [1]: init fby (10 - parent a);
        [2]: init fby (sibling(a, 0) - sibling(a, 1));
        default: eod;
    end;

    ending = p atn [] and p atn [0] and p atn [1] and p atn [2]
             where p = a eq next a;

end.
```

In the above program, the variable a defined at nodes [ ], [0], [1], and [2] is constant in the identifier dimension. At these nodes, a is circularly defined if the time dimension is ignored and there are no initial values for a at the nodes. By using the above iterative

algorithm, based on given initial values, the fixed-point solution for a can be found. The termination condition of the iteration in the program is explicitly expressed by the definition of the variable *ending*. The semantics of the program is the value of a at the time point at which a has the stable value at each node in the tree. Table 4.1 shows the value of a at each node n at time points 0 and 1 when the input variable *init* has value 20 at the root node.

	time	
	0	1
$a_{[\,]}$	20	20
$a_{[0]}$	10	10
$a_{[1]}$	-10	-10
$a_{[2]}$	20	20

Table 4.1: Values at iterations

In the above program, we define the termination of the iteration in a rather ad hoc way. In general, there are two kinds of termination conditions in iterative algorithms. One kind is the so called *local* termination conditions. Given a set of elements, such as nodes on a tree, the termination of an iteration at a node only depends on the current and previous values at the local node and is independent of values of other nodes. In expressing a local termination condition, we can use the operator x *asa* p where the value of p at a node is a sequence of boolean values that is defined locally by values of other variables at the same node. In other words, p may have different value sequences at different nodes. For example, in the above program, a local termination condition can be simply defined as (a *eq next* a).

In many cases, however, a local termination condition is not sufficient. The termination of an iteration may depend on the global stability of all the elements concerned, which we call a *global* termination condition. In our case, it is the stability of the affected values at all the nodes in a tree. A global termination condition is usually an aggregate value formed from values at various nodes. For example, in the above program, the global termination condition is expressed by the definition of the variable *ending*.

In the following, we rewrite the program in Example 4.4 using the temporal operator *tasa*. It defines that a has the final value at the root, when at *every* node on the tree the two consecutive values of a in the time dimension are the same.

```
root (a tasa (a eq next a))
where
    a = case index of
        []:  init fby ((child(a, 2) + 20)/2);
        [0]: init fby ((parent a) - 10);
        [1]: init fby (10 - parent a);
        [2]: init fby (sibling(a, 0) - sibling(a, 1));
        default: eod;
    end;
end.
```

4.4 Evaluation of IFADL Expressions

The semantics of IFADL expressions are defined over the infinite context space C. However, it is impossible to evaluate an expression at all contexts in C. Actually, what we are really concerned with are those contexts at which the expression has a non-*eod* value. In other words, the evaluation for an expression will only be performed on the shape of the expression at the relevant time and identifier points.

As we explained in 3.2, an expression may not have a tree shape. For example, the shape of variable A in Figure 4.7 is not a tree shape, since the value of a at node $[1, 1]$ has an *eod* value.

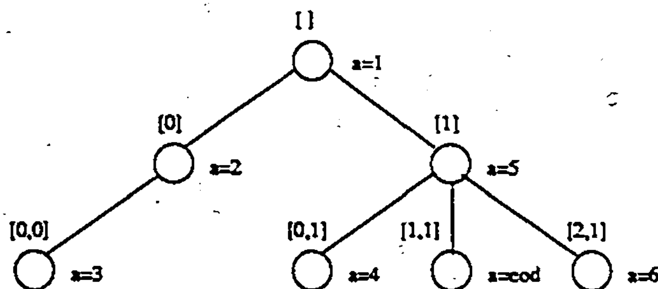

Figure 4.7: A non-tree shape example

One possibility is to *assume* the expression has a tree shape which is a subset of its

full shape. For example, if we exclude the value of a at node [2.1], the rest of the values of a form a tree shape. It is obvious that we lost some information about variable a, but it makes the evaluation for a possible. Since the context space $\mathcal{C}$ is infinite, the evaluation must have some kind of evaluation order on $\mathcal{C}$. No matter whether the depth-first or the breadth-first evaluation order is used, after we find an *eod* value when we search child nodes or sibling nodes, the evaluation has to be stopped in that direction, since we cannot be sure that there are any more non-*eod* values at the contexts in that direction. Therefore, only the values at the contexts which form the tree shape of the expression will be evaluated.

In the next chapter, we will formally define indexical attribute grammars in which we use IFADL to define attributes.

Chapter 5

THE DEFINITION OF IAG

In this chapter, we show how IFADL expressions can be used to specify attribute definition rules in attribute grammars. The semantics of attributes defined by IFADL expressions can be interpreted by the indexical semantics of IFADL. We call such attribute grammars *indexical attribute grammars*.

5.1 The Definition

To specify attributes in attribute grammars as IFADL variables, we need to give attribute occurrences a new form. In the following we first redefine attribute occurrences.

Definition 5.1 (Attribute Occurrence) *Given an attribute grammar ((T,N,S,P), A, D), and a production $p \in P$ of the form $p : X \rightarrow t_0 X_0 t_1 X_1 t_2 \ldots t_{n_p-1} X_{n_p-1} t_{n_p}$, where X_i $(0 \leq i < n_p)$ are non-terminal symbols and t_i $(0 \leq i \leq n_p)$ are terminal symbols. An occurrence o of an attribute symbol $a \in A$ together with a grammar symbol X in p is*

$$\begin{cases} a & \text{when } a \text{ is with } X \\ a\$i & \text{when } a \text{ is with } X_i \end{cases}$$

where $0 \leq i < n_p$.

Definition 5.2 (Indexical Attribute Grammar) *An indexical attribute grammar is a 5-tuple IAG = (G, A, E, GD, LD):*

- *G is a context-free grammar $G = (T, N, S, P)$.*

- *A is a set of attribute symbols.*

- *E is an IFADL expression in which only attribute symbols in A can appear as free variables. The value of E for a given parse tree denotes the semantics of the parse tree.*

- *GD is a set of attribute definition rules (the "global" definitions). An attribute definition rule in GD is an IFADL definition $a = exp$ where $a \in A$ is an attribute symbol and exp is an IFADL expression in which only attribute symbols in A can appear as free variables. Each attribute symbol in A can have at most one definition in* GD.

- *$LD = \{LD_p\}_{p \in P}$ is an indexed family of sets of attribute definition rules, where each LD_p is a set of attribute definitions for a subset of the attribute occurrences (defined in 5.1) associated with production $p \in P$. An attribute definition rule in LD_p is an IFADL definition* o = exp *where o is an attribute occurrence associated with p and* exp *is an IFADL expression in which only attribute symbols in A can appear as free variables. Each of the attribute occurrences associated with p can have at most one definition in LD_p.*

The following is an example of an IAG. Here the meaning of the example is not significant, we use it to show how an attribute grammar is defined in IAG.

```
root child(b, 0)
%%
a = 5;
%%
X -> Y Z
     a = child(a, 0) + child(a, 1);
     b$0 = parent a;
```

Unlike conventional attribute grammars, the attribute value which denotes the semantics of a language sentence does not have to be restricted to the root node of the parse

tree which represents the sentence. Using indexical operators, we can refer to any values in a parse tree as the semantics of the language sentence represented by the parse tree.

In the above example, according to the defining expression `root child(b, 0)`, the semantics of a language sentence defined by the above grammar is the value of b at node [0] in the corresponding parse tree. The IAG has one global definition for attribute a which is `a = 5`. The global definition defines the value of a to be 5 at all the nodes in a parse tree generated by the IAG unless it is overridden by the local definitions for a. The first local definition `a = child(a, 0) + child(a, 1)` says that the value of attribute a at a node labeled by X is equal to the sum of its values at its first child and second child nodes. In the second local definition `b$0 = parent a`, the b$0 at the left hand side of the definition denotes an attribute occurrence. The definition defines the value of b at the nodes labeled by Y to be the value of a at the current node's parent node. Let us consider a parse tree with only three nodes labeled by X, Y and Z. Figure 5.1 shows the values of a on the parse tree.

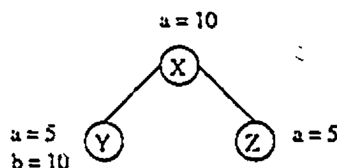

Figure 5.1: A parse tree with attribute values

In this dissertation and without any loss of generality, we associate attributes only with non-terminal grammar symbols. Therefore, for a given parse tree, we are only interested in those nodes that are labeled by production rules. For simplicity, in the following descriptions, we remove all the leaf nodes that are labeled by terminal grammar symbols in parse trees, and refer to parse trees in the simplified version.

A conventional parse tree π is a labeled tree structure. The root node of π is labeled by a production rule p whose left hand side is the start symbol S of G. The non-leaf nodes in π are labeled by production rules and the leaf nodes are labeled by terminal symbols. If a non-leaf node of π is labeled by a production $p : X \rightarrow t_0 X_0 t_1 X_1 t_2 \ldots t_{n_p-1} X_{n_p-1} t_{n_p}$,

the node has n_p child nodes. If its ith child node is also a non-leaf node, the ith child node is labeled by a production p' in which X_i is the left hand side grammar symbol of p'. If its ith child node is a leaf node, the ith child node is labeled by the grammar symbol X_i.

As we described in 3.2, a parse tree π corresponds to an index tree $T_\pi \in \mathcal{T}$.

Definition 5.3 (Production-tree) *The production tree $\mathcal{P}_\pi$ of a parse tree π generated by an IAG is an intension*

$$\mathcal{P}_\pi : \mathcal{T} \to P_+$$

where $P_+ = P + \{cod\}$ is the extended domain of production rules $P \in G$ in the IAG with the special value cod.

$\mathcal{P}_\pi$ is defined by

$$\mathcal{P}_\pi(n) = \begin{cases} p & p \text{ is the production attached to the node of } \pi \text{ corresponding to } n \\ cod & n \text{ does not correspond to a node of } \pi \end{cases}$$

Definition 5.4 (Attribute occurrence definition tree $\mathcal{O}$) *Given a parse tree π generated by an IAG, $\mathcal{O}_\pi$ is the intension*

$$\mathcal{O}_\pi : \mathcal{T} \to LD_+$$

is defined by

$$\mathcal{O}_\pi(n) = \begin{cases} LD_{\mathcal{P}_\pi(n)} & \mathcal{P}_\pi(n) \neq cod \\ cod & otherwise \end{cases}$$

The attribute occurrence definition tree $\mathcal{O}_\pi$ for a given parse tree π is an intension, that is, the value of $\mathcal{O}_\pi$ at a node $n \in \mathcal{T}$ is the set of attribute occurrence definitions associated with the production $p \in P$ in the attribute grammar, where p is the value of $\mathcal{P}_\pi$ at the same node n. In other words, we can also think that the attribute definition rules in IAG are "distributed" to each node in a parse tree. If a node in a parse tree is labeled by production p, the node has attached to it a set of attribute occurrence definition rules which are the attribute definition rules associated with p in the attribute grammar.

In a production rule p, we can give definitions of attributes associated both with the left and with the right hand side of grammar symbols in p. Therefore, for a given parse

tree, the definitions of attributes at a particular node may be associated with different production rules. For example, for a parse tree generated by the following fragment of an attribute grammar, the definitions for attributes a and b at a node labeled by the grammar symbol B come from production rules $p1$ and $p2$ respectively.

```
    (IAG version)                          (Conventional version)
p1: A -> B C                               p1: A -> B C
          a$0 = sibling(a, 1);                       B.a = C.a;
p2: B -> D                                 p2: B -> D
          b = child(b, 0);                           B.b = D.b;
```

Figure 5.2 shows the values of attributes a and b on a parse tree with four nodes labeled by A, B, C, and D. The dashed lines indicate the data dependencies among the attribute values on the parse tree.

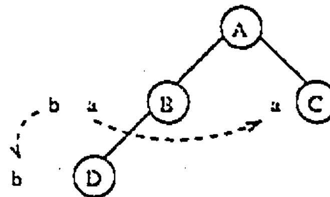

Figure 5.2: Dependencies without conflicts

It is possible that an attribute at a node in a parse tree has more than one definition. For example, the following attribute grammar fragment gives two definitions to a at a node labeled by a grammar symbol B in a parse tree (Figure 5.3). In this case, we say that the definition for a is inconsistent.

```
    (IAG version)                   (Conventional version)
p1: A -> B C                          p1: A -> B C
          a$0 = parent a;                       B.a = A.a;
p2: B -> D                            p2: B -> D
          a = child(a, 0);                      B.a = D.a;
```

Definition 5.5 (Consistency of $\mathcal{O}$) *Given a parse tree π generated by an IAG and production tree $\mathcal{P}_\pi$, $\mathcal{O}_\pi$ is consistent if and only if*

$$\forall n \in T_\pi \neg(\exists a \in A\ (\exists(a, exp) \in \mathcal{O}_\pi(n)\ \wedge\ \exists((a,i), exp) \in \mathcal{O}_\pi(tail(n))\ \wedge\ head(n) = i))$$

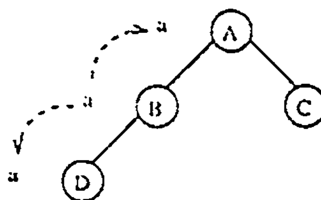

Figure 5.3: Dependencies with conflicts

In indexical attribute grammars, the definitions of attributes are not limited by locality as they are in conventional attribute grammars. On a parse tree, attribute definitions can directly access attribute values at nodes on other parts of the tree through IFADL *node switching* operators. For example, the following attribute definition

```
a = root b
```

at a node n of a parse tree, defines the attribute a at node n to have the value of the attribute b at the root node of the tree.

In conventional attribute grammars, there are usually many identical attribute definition rules associated with different productions. If these duplicate definition rules could be replaced by a single definition, attribute grammars would look much simpler. Since attributes in indexical attribute grammars are intensions over the tree dimension $\mathcal{T}$, it should be possible to define attributes by global definitions.

The global definition of an attribute a is an expression valid at all the nodes of all the possible parse trees generated from the IAG. If an attribute has a global definition, it means that at any node of a parse tree, the attribute has the same definition, although its value at different nodes may be different. For example, a global attribute definition

```
a = child(b, 0) + child(b, 1)
```

says that for a given parse tree π, the value of a, at any node $n \in T_\pi$, is the sum of the values of b at n's first and second child nodes. The attribute a may have dimensionality since its value may differ at different nodes on T_π, if b's value has dimensionality on T_π.

An attribute defined by a global definition may also be constant. For example, a global attribute definition

```
a = root env
```

defines that the attribute a has the same value at all the nodes of T_π as the value of the attribute env at the root node.

To evaluate an attribute at a node, we first need to find the definition for the attribute at that node. The local definition of an attribute at a node at a given parse tree π may be associated with either the same node or the parent node in the corresponding attribute occurrence definition tree $\mathcal{O}_\pi$. Also, in some attribute grammars, an attribute may have the same definition at all but few nodes of a parse tree. We may still define such attributes using global definitions, and meanwhile keep the local definitions for the exceptional nodes. When an attribute has both global and local definitions at a node, the local definition of the attribute overrides its global definition. The following definition formalizes this convention.

Definition 5.6 (Attribute definition tree $\mathcal{D}$) *Given a parse tree π generated by an IAG whose $\mathcal{O}_\pi$ is consistent, $\mathcal{D}_\pi$ is an intension*

$$\mathcal{D}_\pi : T \to (A \to Exp_+).$$

where Exp *is the set of all IFADL expressions with attribute symbols in A as free variables.*

$\mathcal{D}_\pi$ is defined by

$$\mathcal{D}_\pi(n) = \lambda a.\ exp'$$

where

$$\begin{aligned} exp' \;=\; & \mathit{if}\ ((a, exp) \in \mathcal{O}_\pi(n))\ \mathit{or}\ (n \neq [\,]\ \mathit{and}\ ((a\$head(n), exp) \in \mathcal{O}_\pi(tail(n))) \\ & \mathit{then}\ exp \\ & \mathit{else\ if}\ ((a = exp) \in GD)\ \mathit{then}\ exp \\ & \mathit{else}\ eod \end{aligned}$$

Given a parse tree π, the corresponding intension $\mathcal{D}_\pi$ specifies the definitions of attributes on π. The value of $\mathcal{D}_\pi$ at a node n on π is the set of the definitions of attributes at n. Each of the definitions defines the value of an attribute symbol at n.

For a parse tree, the defining expression E in IAG denotes the semantics of the corresponding language string. By indexical semantics, the value of E is not a single value, it may have different value at different nodes on the tree. In this case, E's value has dimensionality. We can also consider that E has a single value, however, if its value has constancy, that is, it has the same value at every node.

5.2 Syntax of Indexical Attribute Grammars

An indexical attribute grammar has three parts. The first part is the expression E which defines the semantics of a language string generated from the grammar. The second part is a set of global attribute definitions GD. The third part is the context grammar G with local attribute definitions LD. The following is simplified syntax of indexical attribute grammars.

iag	—	*result global_def local_def*
result	—	*expression*
global_def	—	*definition_list*
local_def	—	*production_def local_def*
		| *production_def*
production_def	—	*production definition_list*
production	—	*nonterminal* " — " *symbols*
symbols	—	*symbol symbols*
symbol	—	*terminal*
		| *nonterminal*
definition_list	—	*definition definition_list*
definition	—	*attr_occur* " = " *expression*
attr_occur	—	*attribute_symbol*
		| *attribute_symbol* "$" *num*

In the above syntax, *expression* and *num* are defined in the syntax of IFADL. The following example shows what an indexical attribute grammar looks like.

Example 5.1

```
% the defining expression E
(root zero) div (root one)

% the context-free grammar and local definition
```

```
A -> B A
     zero = child(zero, 0) + child(zero, 1);
     one =  child(one, 0) + child(one, 1);
     | B
     zero = child(zero, 0);
     one =  child(one, 0);

B -> '1'
     zero = 0;
     one =  1;
     | '0'
     zero = 1;
     one =  0;
```

In the above context-free grammar, there are two nonterminals $N = \{A, B\}$ and two terminals $T = \{`1', `0'\}$. The sentences in the defined language are sequences of characters '1' and '0'. There are two attribute symbols *zero* and *one*. In the example, all the definitions of attributes are associated with the left hand side grammar symbols of the productions. For a string of the language, the value of the attribute *zero* at a node n in the corresponding parse tree specifies the total number of leaf nodes which are labeled by character '0' in the subtree rooted at n. The value of the attribute *one* at a node n specifies the total number of leaf nodes which are labeled by character '1' in the subtree rooted at n.

The expression $E =$ *(root zero) div (root one)* denotes the semantics of the language sentences. The semantics of a given sentence is the value of the attribute *zero* at the root node divided by the value of the attribute *one* also at the root node of the corresponding parse tree. It is the ratio between the numbers of '0' and '1' in the sentence. For example, if the sentence is "0011", then E has value

```
E = (root zero) div (root one)
  = 2 div 2
  = 1
```

5.3 Denotational Semantics of Indexical Attribute Grammars

An IAG defines the syntax and semantics of a language L(IAG). The syntax of a sentence (or a string) s of L(IAG) is defined by the context-free grammar component G of IAG; its structure is represented by a parse tree π generated from G. The semantics of the parse tree π, hence of the sentence s, is the value of the defining expression E of IAG. The value of E depends on attribute values attached to the index tree T_π corresponding to π.

Attribute definitions of a finite parse tree π generated from an IAG directly correspond to an IFADL program. By a finite tree we mean that the tree has a finite number of nodes. Given a finite parse tree π, the attribute definition tree $\mathcal{D}_\pi$, and the defining expression E, the corresponding IFADL program PT can be constructed as follows.

- E in the IAG is the subject of PT.
- In the *where* clause of PT, we define each attribute symbol $a \in A$ of the IAG by a *case* expression on the indexical variable *index*. In the expression, a is defined by $\mathcal{D}_\pi(n)(a)$ for each index $n \in T_\pi$.

In general, we can conclude the following. For all languages defined by indexical attribute grammars, the semantics of their finite sentences can be defined by the denotational semantics of their corresponding IFADL programs. For example, Example 5.2 is the corresponding IFADL program of the parse tree of string "0011" generated by the indexical attribute grammar defined in Example 5.1.

Example 5.2

```
(root zero) div (root one)
where
    zero =  case index of
            []: child(zero, 0) + child(zero, 1);
            [0]: 1;
            [1]: child(zero, 0) + child(zero, 1);
            [0 0]: eod;
```

```
            [0 1]: 1;
            [1 1]: child(zero, 0) + child(zero, 1);
            [0 0 1]: eod;
            [0 1 1]: 0;
            [1 1 1]: child(zero, 0);
            [0 0 1 1]: eod;
            [0 1 1 1]: 0;
            [0 0 1 1 1]: eod;
            default: eod;

    one =  case index of
           []: child(one, 0) + child(one, 1);
           [0]: 0;
           [1]: child(one, 0) + child(one, 1);
           [0 0]: eod;
           [0 1]: 0;
           [1 1]: child(one, 0) + child(one, 1);
           [0 0 1]: eod;
           [0 1 1]: 1;
           [1 1 1]: child(one, 0);
           [0 0 1 1]: eod;
           [0 1 1 1]: 1;
           [0 0 1 1 1]: eod;
   default: eod;
end.
```

However, the above transformation approach is ad-hoc. In general, we can define the denotational semantics of parse trees generated by indexical attribute grammars directly as follows.

Definition 5.7 (Attribute value tree $\mathcal{V}$) *Given a parse tree π generated by an IAG, $\mathcal{V}_\pi$ is an intension*

$$\mathcal{V}_\pi : \mathcal{T} \times \omega^\omega \times \mathcal{I} \rightarrow (A \rightarrow D_+)$$

$\mathcal{V}_\pi$ is the least solution of the following equation

$$\mathcal{V}_\pi = \lambda(n,t,i)\lambda a.[\![\mathcal{D}_\pi(n)(a)]\!]\ \mathcal{V}_\pi(n,t,i)$$

The above equation with the fixed point solution defines the denotational semantics of an IAG. For any language string defined by an IAG, there is a corresponding parse

tree π. The values of attributes on π represent the semantics of the string. For a given node in π and an attribute symbol, the attribute definition tree $\mathcal{D}_p(n)(a)$ returns an attribute definition (or an expression of IFADL) for the attribute a at the node n. $\mathcal{V}_\pi$ is the environment for evaluating the denotational semantics of the expressions returned by $\mathcal{D}_\pi$. For a given context (n, t, i) and an attribute symbol a, the value of $\mathcal{V}_\pi$ at (n, t, i) is updated by evaluating a at (n, t, i) according to its definition from $\mathcal{D}_\pi(n)(a)$. In different contexts and for different attribute symbols, the expressions from $\mathcal{D}_\pi$ may be different. The evaluation for the denotational semantics of these expressions cumulatively updates $\mathcal{V}_\pi$ until it reaches a fixed point. That means that there are no more values of attributes on $\mathcal{V}_\pi$ that can be updated by further evaluation.

$\mathcal{V}_\pi$ is the least valuation of the tree with the property that the value $\mathcal{V}_\pi$ assigns to each attribute is equal to the value it assigns the attribute's defining expression. In other words, it is the least valuation which makes all the defining equations true at the nodes to which they are attached.

For a given parse tree π, there may be more than one attribute value trees $\mathcal{V}'_\pi$ that satisfies the above equation. The least solution is the attribute value tree $\mathcal{V}_\pi$ such that for all solutions $\mathcal{V}'_\pi$,

$$\forall n \in T \forall t \in \omega^\omega \forall i \in \mathcal{I}\ \forall a \in A\ \ \mathcal{V}_\pi(n, t, i)(a) \sqsubseteq \mathcal{V}'_\pi(n, t, i)(a)$$

in the domain D_+.

Definition 5.8 (The semantics of language strings) *For a given IAG, the semantics of each of its generated language strings represented by parse tree π is defined by*

$$[\![< def_exp >]\!]\ \mathcal{V}_\pi$$

where def_exp is the defining expression of the IAG.

In the next chapter, we describe the expressive power of the above defined indexical attribute grammars.

Chapter 6

EXPRESSIVE POWER OF IAG

6.1 Noncircular Attribute Grammars

Indexical attribute grammars have at least the same expressive power as conventional attribute grammars, but they are more concise and have simple denotational semantics. Using indexical attribute grammars, we can define conventional noncircular attribute grammars by implicitly specifying dependencies among attribute instances using node switching operators. In the first example (Example 6.1) of this section, we redefine a conventional attribute grammar given by Knuth [Knu68].

Example 6.1 (Converting binary strings to decimal values)

```
% The defining expression E
root value;
% The context-free grammar and local attribute definitions
N -> L . L
        value = child(value, 0) + child(value, 1);
        scale$0 = 0;
        scale$1 = -length;
     | L
        value = child(value, 0);
        scale$0 = 0;

L -> L B
        value = child(value, 0) + child(value, 1);
```

```
        length = child(length, 0) + 1;
        scale$0 = parent scale + 1;
        scale$1 = parent scale;
      | B
        value = child(value, 0);
        length = 1;
        scale$0 = parent scale;

B -> 1
        value = 2 ** scale;
      | 0
        value = 0;
```

The defining expression E, *root value*, in the above indexical attribute grammar denotes the semantics (value) for a given binary number. Since the attribute definition rules do not involve any time or identifier switching operators, E is constant in the time and identifier dimensions. Therefore, we can just think of E as the single value of the attribute *value* at the root node.

One advantage of indexical attribute grammars is the ability to give nonlocal attribute definitions using node switching operators. Attributes in an attribute value tree $\mathcal{V}_\pi$ can be defined directly as combinations of attribute values at other parts of the tree without using copy chain rules to pass the values. Therefore upward communication attributes and some downward communication attributes can be removed in indexical attribute grammars.

In indexical attribute grammars, the value of an attribute may be constant in the tree dimension - we call these *constant attributes*. In other words, theoretically speaking, the value of a constant attribute is broadcast to all nodes at a value tree $\mathcal{V}_\pi$, so that other attributes can directly use the value of the constant attribute.

Example 6.2 is an indexical attribute grammar for type checking. In the example, use global definitions to reduce the number of attribute definition rules associated with the production rules.

For simplicity, here we omit some of the production rules and terminal symbols. The purpose of the attribute grammar is to report the existence of a type error but not the location or nature of the error.

Example 6.2 (Global references using node switching operators)

```
% The defining expression E
root (child(no-error, 1))

% The global attribute definitions
     type   = if (child(type, 0) eq child(type, 1))
                 then child(type, 0)
                 else Err [illegible]i;

% The context-free grammar and the local attribute definitions
Prog     -> DeclList StatList
            type-table = child(type-list, 0);

DeclList -> Decl DeclList
            type-list = cons(child(type, 0), child(type-list, 1));
            | .
            type-list = nil;

Decl     -> Id ":" Id ";"
            type  = makepair(child(lex, 0), child(lex, 1));

StatList -> Stat StatList
            no-error = child(no-error, 0) and child(no-error, 1);
            | Stat
            no-error = child(no-error, 0);

Stat     -> Id "=" Expr  ";"
            no-error = gettype(child(lex, 0), root type-table)
                       eq child(type, 1);

Expr     -> Expr + Expr
            | Expr - Expr
            | Id
            type = gettype(child(lex, 0), root type-table);
```

There are five attribute symbols in Example 6.2. Attribute *type-table* is defined at the root node only. From any node on a parse tree we can directly switch to the root node to refer to the value of *type-table* by the node switching operation. The value of *type-table* is a list of pairs which can be viewed as a symbol table. The attribute *type-list* contains partial type information at the nodes labeled by grammar symbols `DeclList` and `Decl`.

The attribute *no-error* has boolean value at nodes associated with grammar symbols `Expr`, `StatList` and `Stat`, which denotes whether there is a type error in the subtree rooted by the node. Attribute *type* at a node denotes the type of the node. *Err* is a special data value which denotes a type error. Since the definitions of attribute *type* in productions

```
Expr -> Expr + Expr
Expr -> Expr - Expr
```

are the same, we define the attribute using a global definition. In theory, the definition should define the attribute *type* only at those nodes which are labeled by grammar symbol `Expr`. However, since the semantics of a sentence in the defined language only depends on the value of *type* at the nodes labeled by `Expr`, the value of *type* at other nodes of the tree is irrelevant, or the definition of *type* at other nodes never affects the semantics of the language string. Note that an attribute defined by a global definition may not be a constant attribute, unless its defining expression is constant in the tree dimension. For example the attribute *type* has dimensionality in the tree dimension.

Given a program fragment defined by the above attribute grammar

```
   ...
a : int
b : char
   ...
a = b ;
   ...
```

the attribute dependencies for the above program fragment on the corresponding attribute value (derivation) tree is illustrated in Fig 6.1.

The evaluation is started by demanding the value of *no-error* at node [1]. This demand then produces new demands for other attributes at other nodes according to the data dependencies. When a demand for the value of `type` reaches a node labeled by `Expr` and the only child of the node is labeled by `id`, the value returned is the type of the identifier from *type-table* at the root node [ ], accessed directly by using the node switching operator *root*.

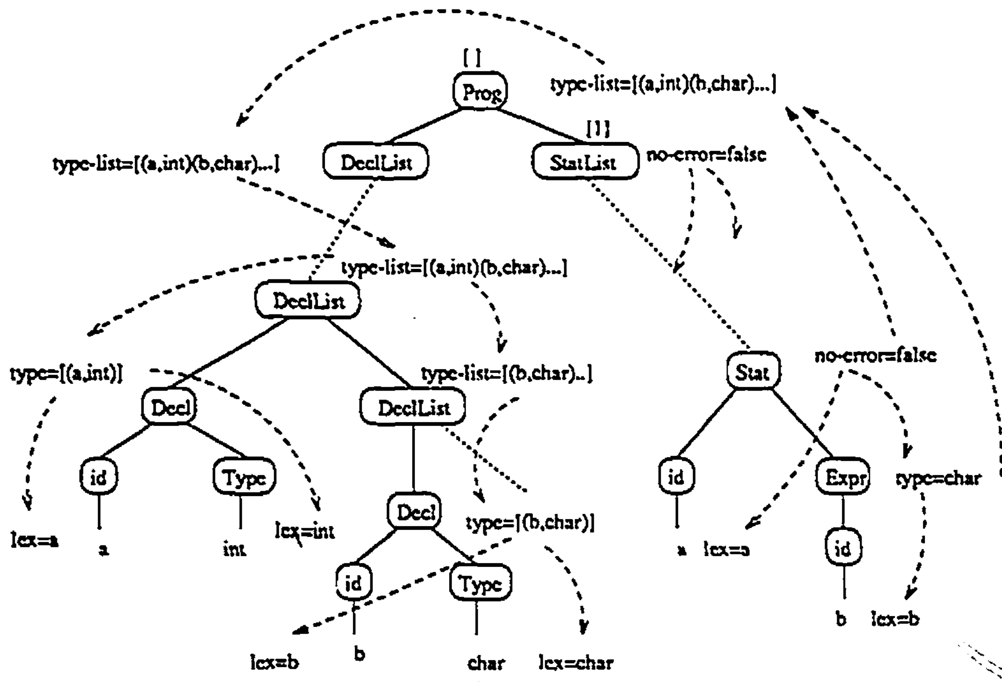

Figure 6.1: An attribute value tree

Thus, at evaluation time, the value of *type-table* does not have to be passed to each node in the subtree rooted at a node labeled by the grammar symbol `StatList`. The attributes in the tree which depend on the value of *type-table* can directly refer to the value of *type-table* through the node switching operator *root*, so not even the address of *type-table* has to be passed node by node. The space and time saving is significant for an attribute evaluator using this approach. Since the symbol table is often very large, it is impractical to implement attribute evaluation by copying the duplicated values to each node on a parse tree. Although we can just pass the address to each node, the process of passing addresses in a large parse tree is still inefficient in time and space.

In the above example, a program in the defined language does not have nested scopes. Therefore, the value of attribute *type-table* can be obtained from all the nodes in the corresponding attribute value tree. In general, however, a language may allow nested scopes. In this case, attribute instances in different subtrees which represent different scopes, may depend on the values of attributes at the root nodes of their own subtrees,

instead of globally depending on the attribute instances at a single node. Furthermore, in a nested block structure, if a variable is declared in both an inner and its outer block, the declaration of the variable in the inner block should have higher priority. If a variable is declared only in the outer block, the variable should have the same declaration in the inner block as in the outer block.

To allow attribute instances to depend on the root node of a subtree, in the following, we first define a unary node switching operator upasa (for "*up as soon as*"). The operator upasa applied to x, at a node n, searches n's ancestor nodes to find the first ancestor at which the attribute instance of x has an non-*eod* value, then returns the value of x at that node.

The operator upasa is defined by

```
upasa(x) = if isroot then eod
           elseif iseod(parent x) then upasa(parent x)
                                  else parent x
           fi;
```

Example 6.3 is a modified example which allows nested *Block* structures. In the example, the attribute *type-table* is associated with the grammar symbol Block, which contains the type information about the variables in the scope of the current block structure.

Example 6.3 (Global references on nested scopes)

```
% The defining expression E
root child(no-error,0)

% The global attribute definitions
     type = if (child(type, 0) eq child(type, 1))
             then child(type,0);
             else Err fi;
     no-error = true;

% The context-free grammar and local attribute definitions
Prog     -> "block" Block
            type-table = [ ];

Block    -> DeclList StatList
```

```
            no-error = child(no-error, 0) and child(no-error, 1);
            type-table = scope-type(child(type-list, 0),
                                     upasa(type-table));

DeclList -> Decl DeclList
            no-error = child(no-error, 0) and child(no-error, 1);
            type-list = cons(child(type, 0), child(type-list, 1));
            | Decl
            type-list = NIL;

Decl     -> "block" Block
            no-error = child(no-error, 1);
            | Id ":" Id
            type  = makepair(child(lex, 0), child(lex, 1));

StatList -> Stat StatList
            no-error = child(no-error, 0) and child(no-error, 1);
            | .

Stat ->     Id "=" Expr  ";"
            no-error = gettype(child(lex, 0), upasa(type-table)) eq
                       child(type, 1);

Expr ->     Expr + Expr
            | Expr - Expr
            | Id
            type = gettype(child(lex, 0), upasa(type-table));
```

In Example 6.3, the declarations of variables in different blocks are stored in the attribute *type-table* associated with the nodes labeled by a grammar symbol Prog or Block in a given parse tree. The function *scope-type* compares the type-table at the current block and the type-table at the first outer block and adds the proper type information of variables to the type-table at the current block.

For the following program fragment which has nested block structures, the attribute dependencies on the corresponding attribute value tree are illustrated in Fig 6.2.

```
 ......
block
  a : char
  ......
  block
```

```
  a : int
  b : int
......
  a = b ;
......
```

6.2 Circular Attribute Grammars

IAG allows attribute values to be data streams. Using time switching operators, circular attributes of conventional attribute grammars can be defined non-circularly in terms of data streams, as long as the iteration for computing these "*circularly*" defined attribute values terminates.

6.2.1 Classification of Circular Attribute Grammars

In an indexical attribute grammar, it is also possible that attribute definitions at different nodes of a parse tree contain circularities. The circularities are caused by using node switching operators. Generally speaking, circular attribute grammars can be classified into three categories. The first kind of circular attribute grammars are defined as erroneous due to a mistake by the user. In this case, the semantics of circularly defined attributes is $\perp$ (denoting non-terminating computation). We call this kind of circularity *deadlock*. Attribute definitions that cause deadlock are also errors in indexical attribute grammars. Deadlock occurs in an indexical attribute grammar when there exists a parse tree generated from the grammar, such that an attribute instance defined by an IFADL expression at a node on the tree indirectly depends on its own value at the same node. In this dissertation, we do not discuss deadlock problems in detail. In general, we can detect deadlock in two steps. The first step is to find a circular attribute definition using the method given in [Knu68] [Knu71]. The second step is to detect whether there is deadlock in the corresponding IFADL program. For the circularities found in the first step, we collect the corresponding attribute definitions. Using these attribute definitions we form an IFADL program. Then the approach given in [Wad81] can be used to detect some

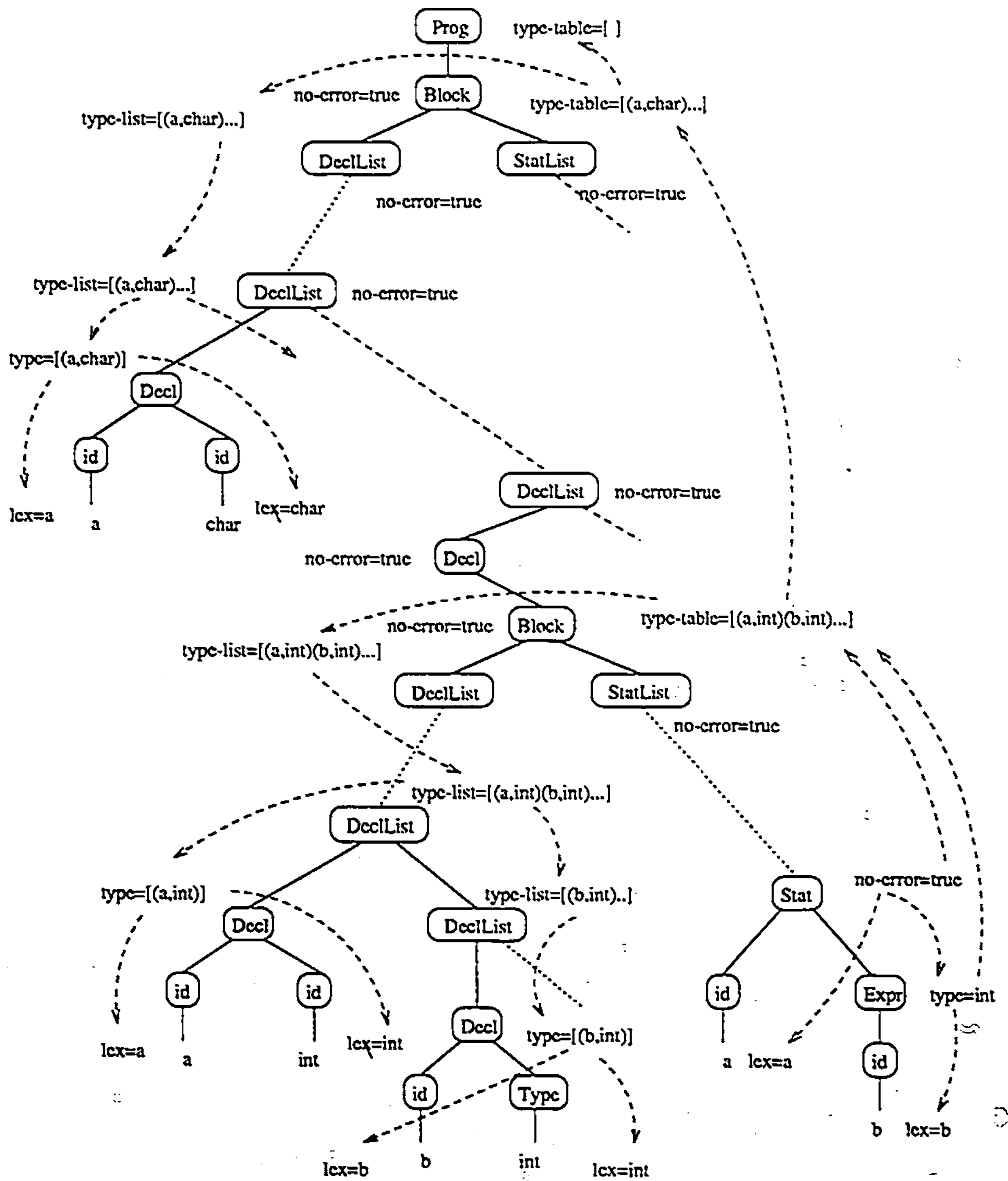

Figure 6.2: An attribute value tree with nested structure

instances of possible deadlocks.

The second kind of circular attribute grammars are the so called *well-defined* circular attribute grammars with *fixed-point solutions*, and the fixed-point solutions can be computed by a successive approximation approach. Well-defined circular attribute grammars relax the definition of traditional attribute grammars by allowing attribute definitions to depend recursively on each other but with some restrictions, such as that attribute values must be in chain partial order and monotonicity of semantics functions [Far86]. Applications of this kind of attribute grammars include instruction selection for code generation [GF82], control and data flow analysis [Far86], and VLSI design problems [JS86].

The third kind of circular attribute grammars is more general. Such attribute grammars do not deadlock, but their solutions cannot be computed by the successive approximation either. To build the evaluator, it needs explicit expressions from the user to specify where and how the evaluation of a cycle of circularly defined attributes starts and when the evaluation terminates [WJ88].

Indexical attribute grammars have the ability to handle non-deadlocked circular attribute grammars. In IAG, an attribute can be defined iteratively in the time dimensions by using time switching operators. In this case, an attribute value at a time point can depend on its and other attributes' values at previous time points; and the attribute has an initial value at time 0. Using time switching and other indexical operators, we can define circular attributes as non-circular but *temporal* attributes. We call this kind of attribute definitions *temporal* attribute definitions.

6.2.2 Circular Attributes via Temporal Attributes

Conventional attribute grammars have no expressive power to specify circular attribute definitions except those that can be solved by the successive approximation fixed-point finding approach. For example, the definition of an attribute x which is circularly defined would have the form:

$$A.x = \ldots B.y \ldots$$

where $B.y$ directly or indirectly depends on the value of $A.x$.

An attribute definition that would be circularly defined in conventional attribute grammars can be expressed as a non-circular but temporal attribute definition in IAG, by breaking the circular dependency into a temporal but non-circular dependency. To do so, we need the following two steps.

For an IAG, we say that there is a set of circular definitions of attribute occurrences, if there is a parse tree generated by the IAG such that on the tree the values of the attribute instances defined by the attribute occurrence definitions in the set depend on each other. Given an IAG with circular attribute occurrence definitions, for each set of circular definitions of attribute occurrences, we first find the *key definition* in the set. In an attribute occurrence definition $(o, exp) \in LD_p$, o is an attribute occurrence associated with production p having the form $b \in A$ or $(b, i) \in A \times \omega$, and exp is an IFADL expression with attribute symbols as free variables, of the form $e(a_1, a_2, \ldots a_k)$. In the following we also write the definition in the form

$$o = e(a_1, a_2, \ldots, a_k)$$

Let (o, exp) be in a set of circular definitions of attribute occurrences. (o, exp) is the key definition of the set if the following conditions are satisfied.

- For each node n of each parse tree generated by the IAG at which the attribute instance b_n of the attribute symbol b in o is defined by (o, exp), b_n must always be in a cycle in the dependency graph of attribute instances corresponding to the tree.
- By giving b_n an initial value, the attribute instances in the cycle can be evaluated iteratively until their values are consistent with their definitions.

For a set of circular definitions of attribute occurrences in an IAG, with the key definition (o, exp), we can rewrite the key definition as a temporal definition by redefining exp using the following IFADL expression, whose value at a node of a parse tree is a data stream

$$o = initial\ fby\ e(a_1, a_2, \ldots, a_k)$$

where *initial* is an IFADL expression whose value can be a constant or a combination of the values of attribute instances outside the cycle. At a node n on a parse tree at which the attribute instance b_n is defined by the above definition, where b is in o, the initial value of b_n at time point 0 in the first time dimension is the value of *initial* at node n at the same time point. Since the time switching operator *fby* switches a time point $t_0 > 0$ to the previous time point $t_0 - 1$, at time point $t_0 > 0$ the expression $e(a_1, a_2, \ldots, a_k)$ is evaluated at $t_0 - 1$, which depends on the values of attributes $a_1, a_2, \ldots, a_k$ at $t_0 - 1$. In other words, by the above definition, the value of b_n at each time point $t_0 > 0$ depends on the value of itself, directly or indirectly defined in the original circular definitions, at time point $t_0 - 1$. Since b_n at time 0 has the defined value, at all time points the value of b_n is computable. Thus the circular dependency is broken into the non-circular temporal dependency. Also, by the above definition, since the value of b_n is a data stream, the values of all other attribute instances on the parse tree that depend on b_n are also data streams.

The second step is to reduce the values of attribute instances in a cycle that are data streams defined by iterations to scalar values by certain termination conditions. Here we only need to reduce those stream attribute values that some other attribute values outside the cycle depend on. For each attribute occurrence definition (o, exp) that defines such an attribute instance in the cycle, where an attribute symbol b appears in o, we introduce a new attribute symbol b' and its occurrence definition (o', exp') that has the form

$$o' = b \; asa \; ending$$

or

$$o' = b \; tasa \; ending$$

where *ending* is a boolean expression in IFADL with attribute symbols as free variables. *ending*, whose value varies in the time dimension, defines the termination condition of the iteration. If the termination is determined locally at a node and only depends on the attribute values at the node, the operator *asa* is used. If the termination is determined globally and depends on the attribute values at nodes on a subtree or the whole tree, the

operator *tasa* is used. The operators *csa* and *tasa* reduce the temporal dimensionality of the value of attribute b to constancy. The references to the attribute occurrence o in all other attribute occurrence definitions outside the cycle should be substituted by the new attribute occurrence o'.

The following example shows how to define temporal attribute definitions. We define a programming language that allows a constant identifier to be used before it is declared. Obviously, a one-pass compiler for the language is not sufficient. For example, for a constant declaration fragment

```
CONST
    a = b + c;
    b = c + 1;
    c = 100;
```

the compiler has to traverse the parse tree three times in order to compute the values of constant identifiers a, b, and c. During the first pass, a can not be assigned any value since both the values of b and c are unknown. Similarly, b can not be assigned any value during the the first pass. Only c is successfully declared as a constant with value 100. After second pass, b is declared as a constant 101. At last, a is declared as a constant 201.

If an attribute evaluator adopts a successive approximation approach for evaluating circular attributes, the attribute grammar can be defined as follows.

Example 6.4 (Evaluation using successive approximation approach)

```
DefModule   -> ConstDecls
               env = child(env,0);

ConstDecls  -> ConstDecl ConstDecls
               env = adddecl(child(def,0), child(env,1));
               |
               env = [ ];

ConstDecl   -> Id "=" Expr ";"
               def = makepair(child(lex,0), child(val,1));

Expr        -> Expr "+" Expr
               val = child(val,0) + child(val,1);
```

```
| Expr "-" Expr
val = child(val,0) - child(val,1);
| Id
val = getval(child(lex,0), root env);
| Num
val = child(val,0);
```

The circular definition is caused by the definition at the first production rule. The value of *env* at the root node depends on the value of *env* at node [0] which indirectly depends on the value of *env* at the root node.

Using the time dimension and time switching operators, the process for computing the values of the constant identifiers can be defined by temporal attribute definitions. The attribute grammar in Example 6.5 defines a part of a constant declaration fragment of a programming language. In the language, constant declarations are treated as definitions, which can be defined in any order.

Example 6.5 (Evaluation using demand-driven approach)

```
root finalenv

DefModule   -> ConstDecls
               finalenv = child(env,0) asa
                          (child(env,0) eq next child(evt,0));
               env = [ ] fby child(env,0);

ConstDecls -> ConstDecl ConstDecls
               env = adddecl(child(def,0), child(env,1));
               | .
               env = [ ];

ConstDecl  -> Id "=" Expr ";"
               def = makepair(child(lex,0), child(val,1));

Expr       -> Expr "+" Expr
               val = child(val,0) + child(val,1);
               | Expr "-" Expr
               val = child(val,0) - child(val,1);
               | Id
               val = getval(child(lex,0), root env);
               | Num
```

```
val = child(val,0);
```

For a given constant declaration, its semantics is the value of attribute *finalenv* which is a list associated with the root node of the corresponding attribute value tree. The list consists of pairs that denote the values of constant identifiers. Besides *finalenv*, the attribute grammar has other two attributes *env* and *val*. The value of *env* is a list of pairs. A pair consists of a constant identifier and its value. The value of *val* is an integer. The function *adddecl* (for "*add declaration*") has two parameters. one is a pair and the other is a list. *adddecl* inserts the pair into the list. Before the insertion, *adddecl* checks the list to see if the identifier in the pair has already had a value in the list. If the identifier has no entry in the list, then the pair is inserted; otherwise, it updates its value with the value in the pair. The function *getval* gets the value of an identifier from the attribute *env* associated with the root node. If an identifier has no entry in the list, the *getval* will return an *eod* value.

The temporal attribute definition

```
env = [ ] fby child(env,0);
```

is associated with the first production rule. At time 0, the *env* at the root node is initialized to an empty list. The value of *env* at its first child is the aggregation value from the nodes in the subtree rooted at the first child node. Consequently, all the identifiers which request values from *env* will have *eod* value initially. After time 0, the value of *env* at the root is gradually updated until the value of *env* at a time point is equal to the value at the next time point.

For example, for the following constant definitions

```
a = b + c;
b = c + 1;
c = 100;
```

the attribute value trees at different time points are shown in Fig 6.3 – 6.6.

The evaluation starts from demanding the value of *finalenv* at the root node [ ] at time point 0 which requires the value of *env* at node [0] at time 0 and time 1. The demands

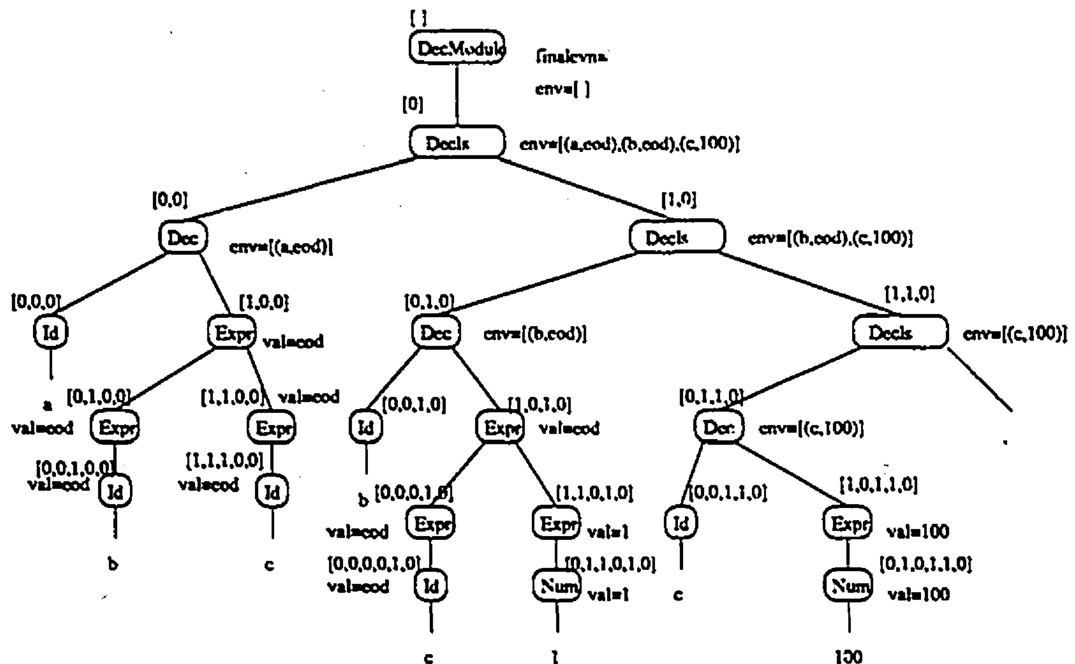

Figure 6.3: The attribute value tree at time 0

are then sent out to the appropriate nodes in the tree. At time 0, the value of *env* has the initial value [ ] at the root node. The value of *val* at the node [1,0,0] has an *eod* value since the constant identifiers *a* and *b* have no entry in the environment table – the value of *env* at the root node. The value of *val* at node [1,0,1,0] is also equal to *eod* since at time 0 the constant identifier *b* has an *eod* value. Therefore, *env* at nodes [0], [1,0] has partial values including *eod* values for the constant identifiers *a* and *b*.

At time 1, the value of *env* at the root is equal to $[(a, \text{eod}), (b, \text{eod}), (c, 100)]$ which is the value of *env* at node [0] at time 0. The value of *val* at node [1,0,0] still has an *eod* value since the constant identifiers *a* and *b* have no non-*eod* value in the environment table. The value of *val* at node [1,0,1,0] has the non-*eod* value 101 since at time 1 the constant identifier c has value 100 at the environment table. The attribute *env* at the nodes [0], [1,0] therefore has the updated values shown in Fig 6.4.

At time 2, all the attributes *env* and *val* have obtained the complete values. Hence at time 3, the value of *env* at the root node is the same as its value at time 2. Since the

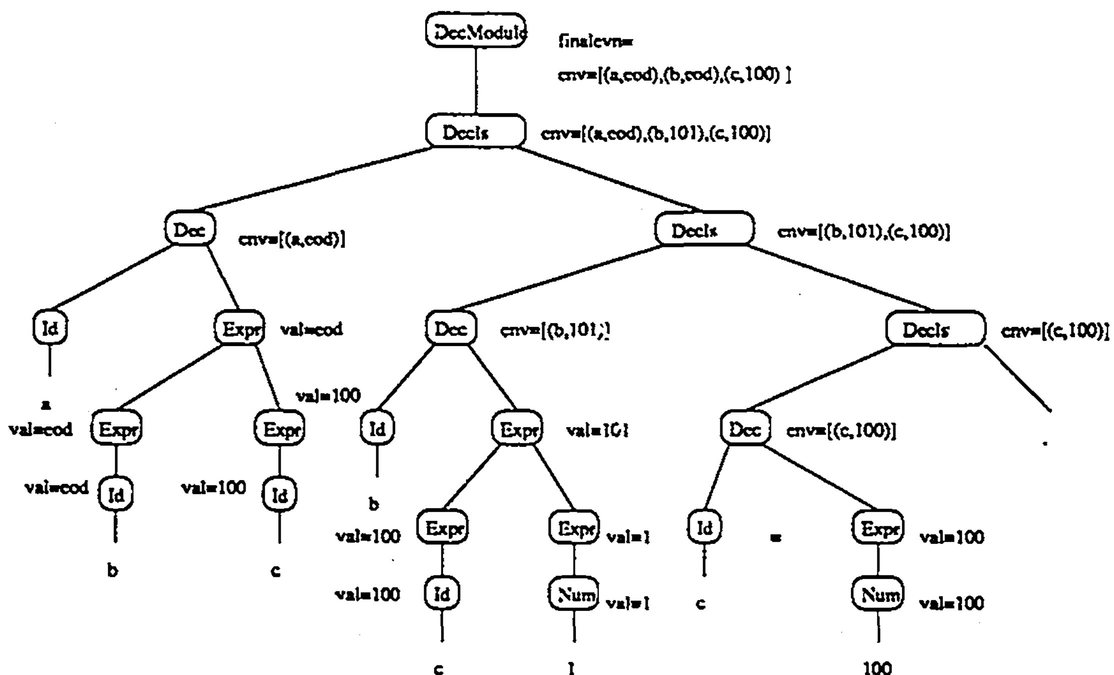

Figure 6.4: The attribute value tree at time 1

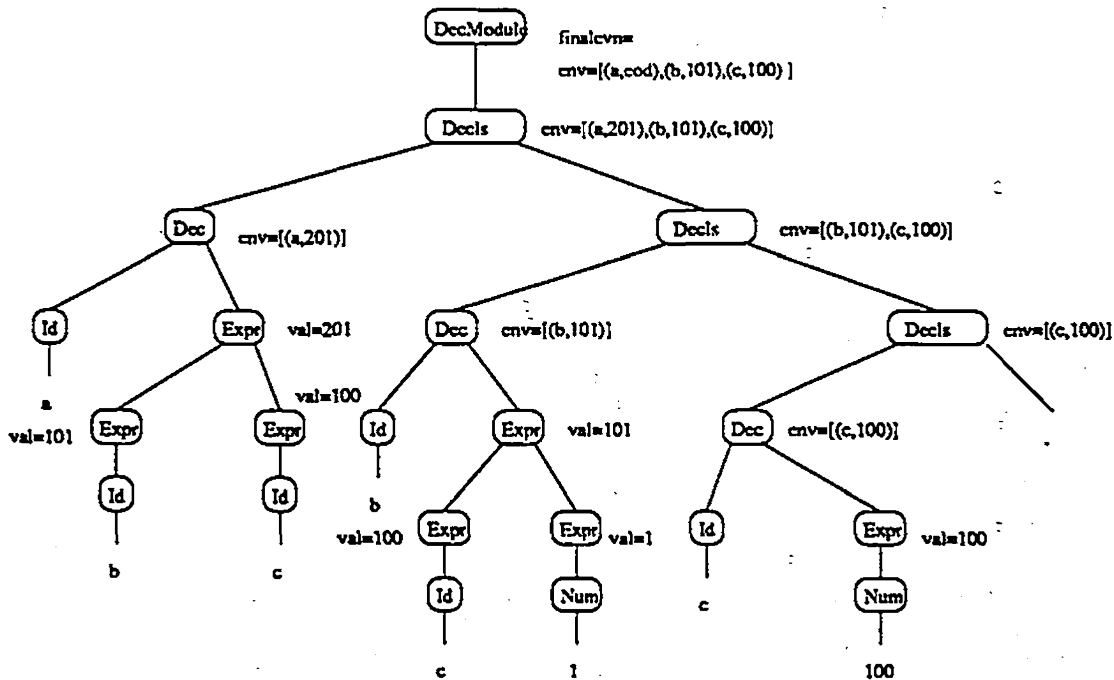

Figure 6.5: The attribute value tree at time 2

value of *env* at time 2 is equal to its value at time 3. finally, *finalenv* has a constant value which is the value of *env* at node [0] at time 2.

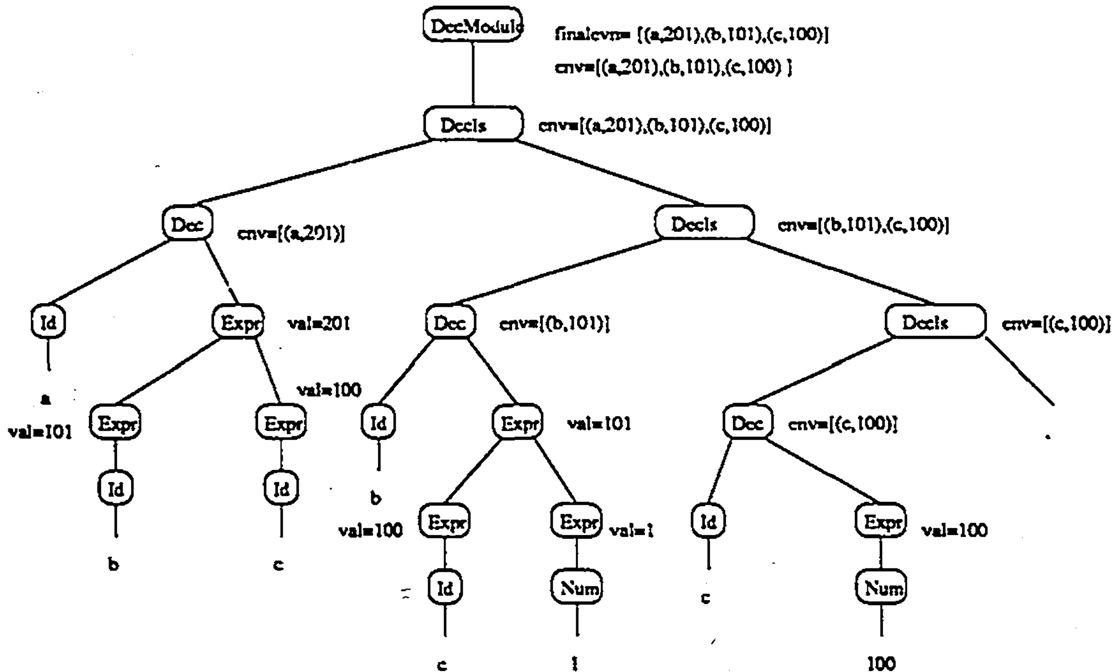

Figure 6.6: The attribute value tree at time 3

Example 6.6 shows another application of indexical temporal attribute grammars. Some functional program languages are not statically typed. The type checking for these languages is done at run time. In the following we show that, using temporal attribute definitions, part of this kind of type checking can also be performed at compile time. Example 6.5 is an attribute grammar fragment for the functional language Lucid without nested *where* clauses. An expression in Lucid may have one of the three basic types - - integer, char, and list. To perform type checking, we associate an attribute *type* with every node. The purpose of associating attribute *type* even with nodes labeled by grammar symbols `DefList` and `Def` is to make *type* have the same tree shape as the attribute value tree, since the operator *tasa* expects tree shape. The type checking terminates when the value of *type* at every node in a given parse tree does not change along the time dimension.

Example 6.6 (Defining type inference by circular attribute definitions)

```
root (type tasa (type eq next type))
%Global definitions
            type = if child(type,0) eq "int" and
                      child(type,1) eq "int"
                      then "int"
                      else Err
                   if;

%Productions and local definitions
Prog     -> Expr
            type-list = [ ];
            type = child(type,0);
            | Expr "where" DefList "end"
            type-list = [ ] fby child(type-list,1);
            type = child(type,0);
DefList  -> Def DefList
            type-list = insert-type-list(child(newtype,0),
                                         child(type-list,1));
            type = "Def";
            | .
            type-list = [ ];
            type = "Def";
Def      -> ID "=" Expr ";"
            type = "Def";
            newtype = makepair(child(lex, 0),child(type,1));
Expr     -> expr "+" expr
            | expr "-" expr
            | expr "fby" expr
            type = if child(type,1) eq eod
                      then child(type,0)
                   elseif child(type,0) eq child(type,1)
                       then child(type,0)
                       else Err
                   fi;
            | ID
            type = get(child(lex,0),root type-list);
            | NUM
            type = "int";
            | CHAR
            type = "char";
```

For example, given an expression with the form:

```
a
  where
    a = b fby c;
    b = 5;
    c = 'A' fby c;
  end;
```

The expression will yield a type error since b has the type 'int' and c has the type 'char'. Figures 6.7–6.9 show the attribute value trees at time 0, 1, and 2.

The evaluation process is similar to Example 6.5. At time 1, the values of attribute *type* at all the nodes on the tree become non-*eod* values. Therefore at time 2, the semantic expression of the tree has the type information [(a,Err),(b,int),(c,char)], which is the aggregated value of *type* at the root node at time 2.

6.2.3 Nested Circularities via Multiple Time Dimensions

The temporal attribute definitions described in the last section do not involve multiple time dimensions. In other words, the values of attributes at all time points other than the first one, t_0, have constancy. In this case, we can simply think that there is only one time dimension – t_0. However, many programming languages support nested structures. When a temporal attribute definition is associated with a nested grammar structure, switching contexts only in the first time dimension t_0 is not enough to specify the values of temporal attributes at different nested structures. To specify temporal attributes on nested structures, multi-time dimensions are required to represent different nesting levels.

The example below, which specifies the denotational semantics of a programming language with *while*-statements, involves nested temporal definitions. In the example, the two attributes *init* and *final* specify the states before and after a statement is executed, respectively. At nodes labeled by the non-terminal symbols `Stat` and `StatList`, the attributes *init* and *final* will have non-*eod* values which are lists of pairs consisting of an identifier name and the value of the identifier. The attribute *val* at a node labeled by `Exp` or `Int` represents the value that expression denotes. At nodes other than those labeled by

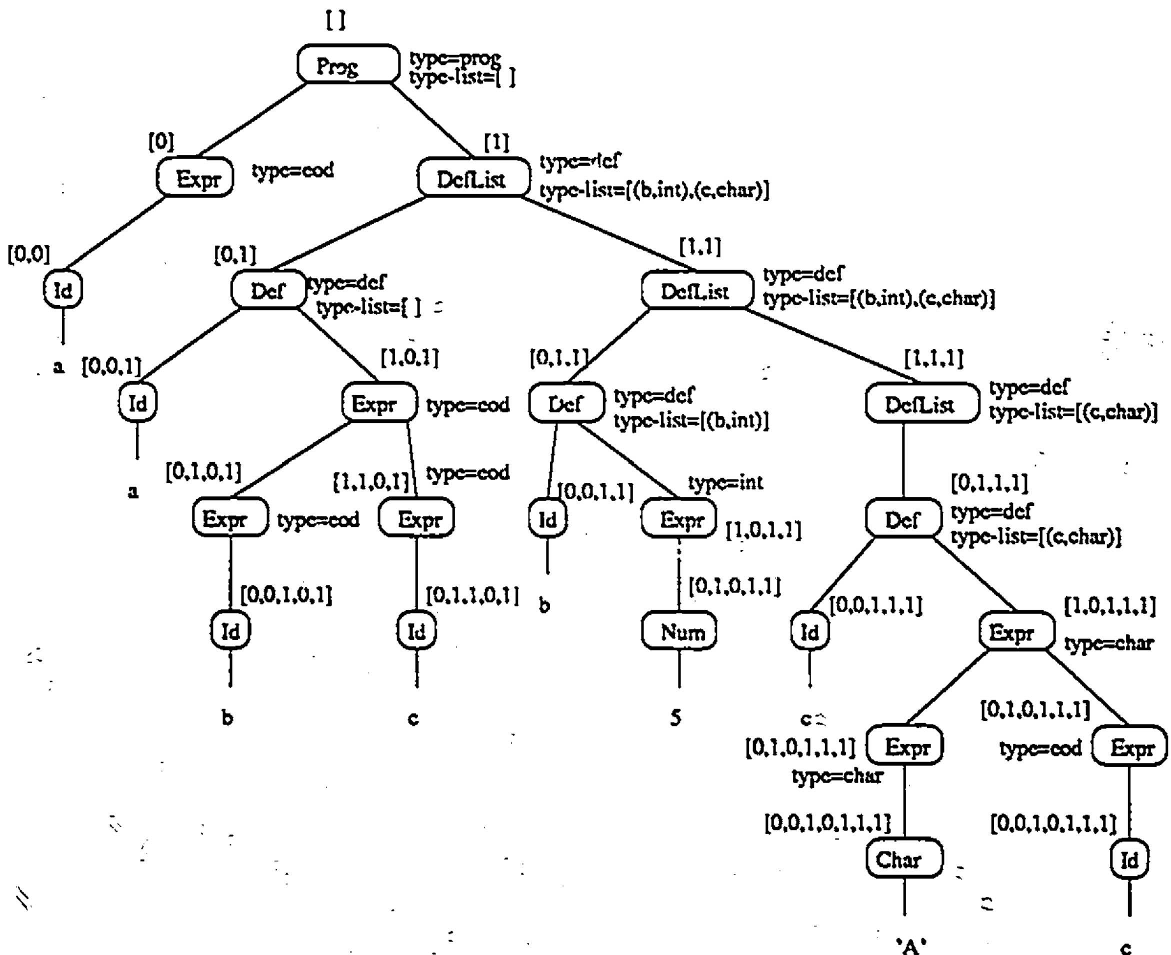

Figure 6.7: The attribute value tree at time 0

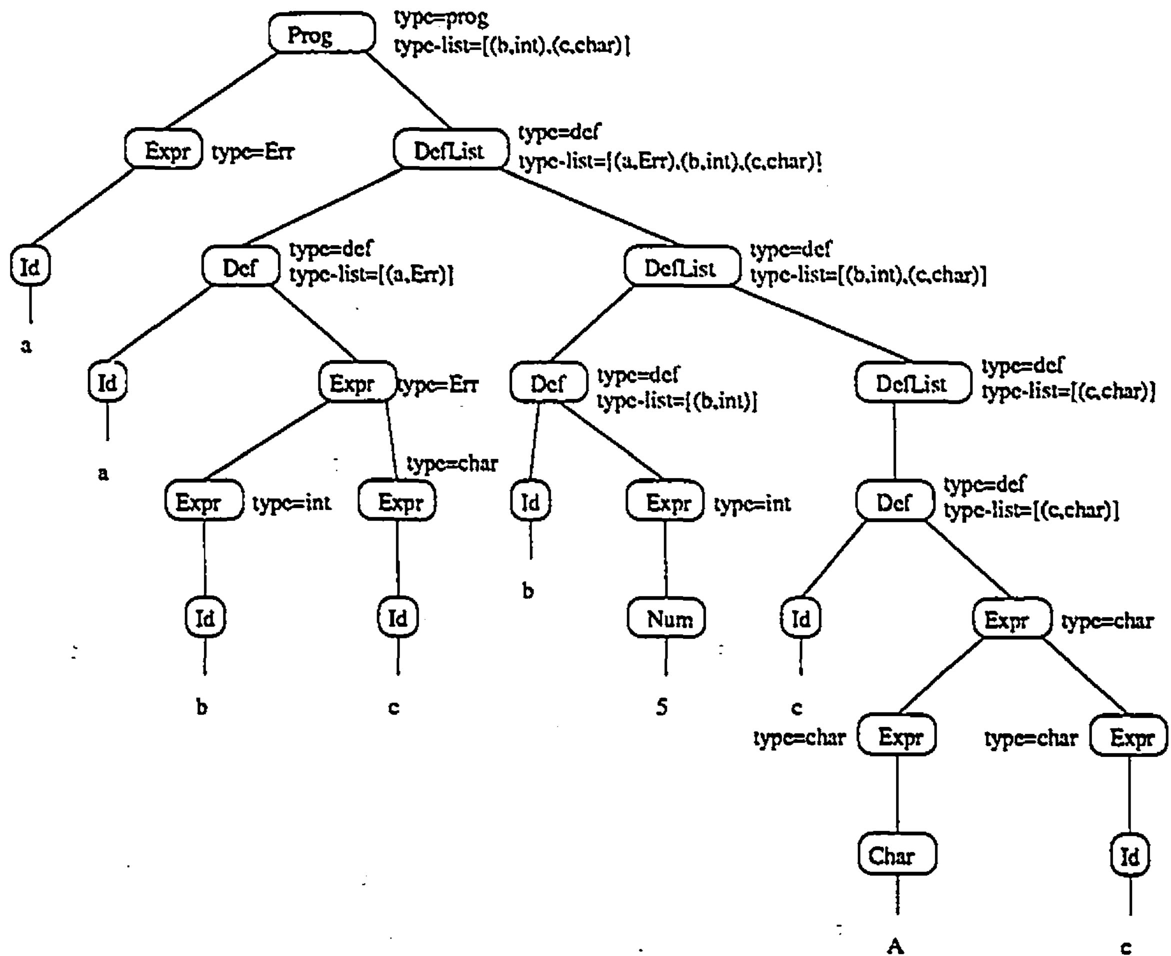

Figure 6.8: The attribute value tree at time 1

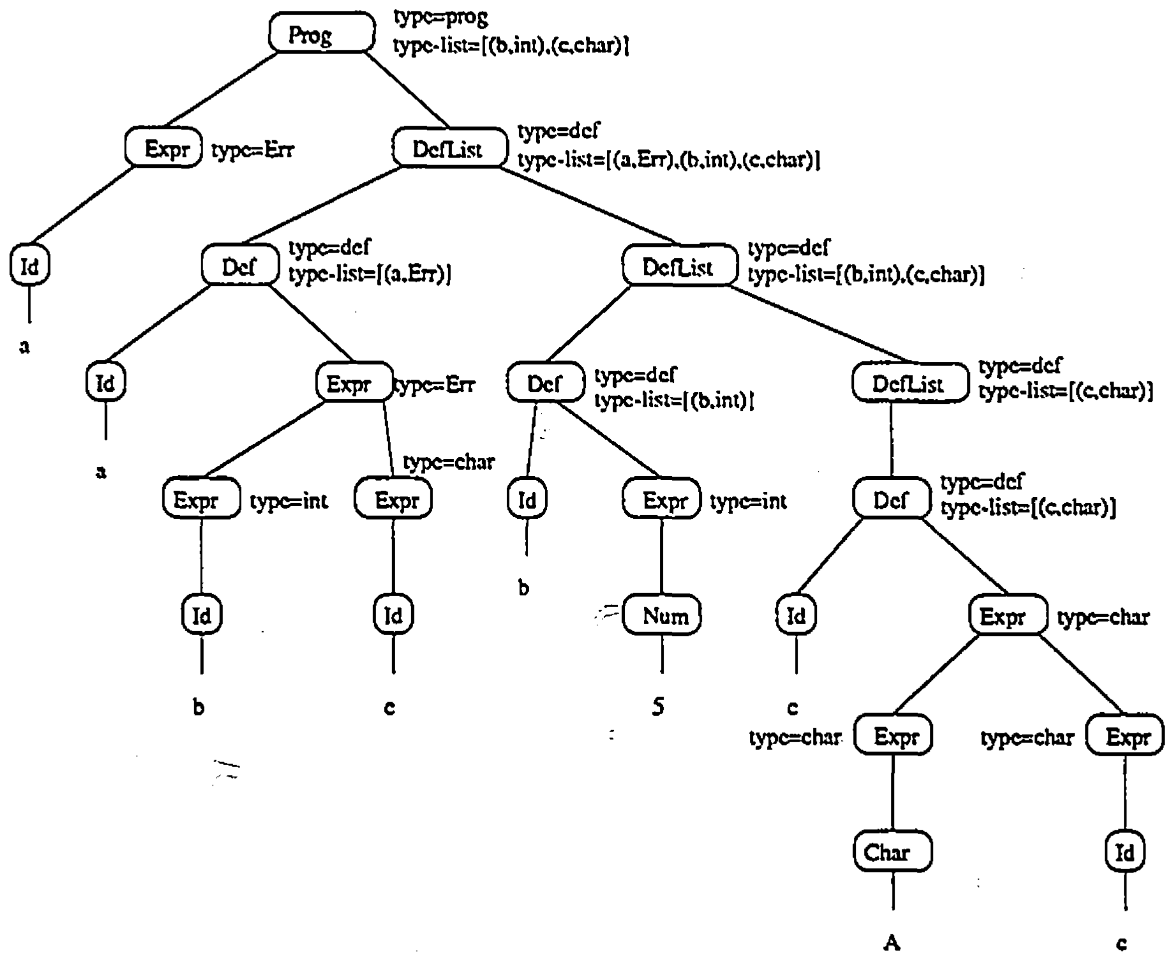

Figure 6.9: The attribute value tree at time 2

Exp or Int, *val* will have an *eod* value. The attribute *lex* represents the lexical values of identifiers at a node labeled by Id.

The function *makepair* takes two arguments, an identifier and the value of the identifier, and makes a pair from them. The function *addenv* adds a pair into a list. If an identifier already has a value pair in the list, *addenv* will remove the old pair and insert the new pair. The function *lookup* gets a value for an identifier from a list (which is the value of *init*).

Example 6.7 (Supporting nested circular definitions)

```
% The defining expression E;
root child(final, 0);

% The context-free grammar and local attribute definitions;
Program  -> StatList
              init$0 = [ ];

StatList -> Stat ";" StatList
              final = child(final, 1);
              init$0 = parent init;
              init$1 = sibling(final, 0);
            | .
              final = init;

Stat     -> "while" Exp "do" StatList
              final = contemp (child(final, 1) asa
                                 (not next child(val, 0)));
              init$0 = sibling(init,1);
              init$1 = branch ((parent init) fby final);
            | Identifier "=" Exp
              final = addenv(makepair(child(lex,0), child(val,1)), init);

Exp      -> Exp "+" Exp
              val = child(val, 0) + child(val, 1);
            | Expr "-" Exp
              val = child(val, 0) - child(val, 1);
            | Id
              val = lookup(child(lex, 0), upasa(init));
            | Int
              val = child(val, 0);
```

In the above attribute grammar, if we ignore the indexical operators *contemp* and *branch*, we can see that the circularity is involved by the attribute definition rule for (*init*,1) in the production for *while*-statement. The (*init*,1) first gets the initial value from its parent node at time 0 then it gets the value of *final* at the same node after time 0. The difference is that the operator *branch* also branches a new time dimension t'_0 for *init* and shifts the original time dimensions from $< t_0, t_1, \ldots >$ to $< t'_0, t_0, t_1, \ldots >$. The new time dimension represents a new nested iteration level. The termination of the evaluation depends on the value of *val* associated with the node labeled by the grammar symbol Exp in the same production. When the value of *val* becomes *false* at time point t of the new time dimension t'_0, the operator *contemp* reduces the time dimension t'_0 of *final* and obtains its value at time 0 of the original time dimension t_0.

Figures 6.10 -6.13 show the attribute value trees for the following program fragment in the language defined by the above IAG at different crucial time points.

```
a = 2;
while a > 0 do
  b = 2;
  while b >0 do
    b = b - 1;
  end;
  a = a - 1;
end;
```

At the first *while*-statement, the *branch* operator at node [1,0,1,0] branches the time dimension from $< t >$ to $< t', t >$ and at the second *while*-statement, the *branch* operator at node [1,0,1,1,0,1,0] branches the time dimension from $< t', t >$ to $< t'', t', t >$.

Note that the value of *final* at node n = [0,1,1,0,1,0] at time $(0,0)$ depends on the value of *final* at time $(t, 0, 0)$ at n's second child node [1,0,1,1,0,1,0], when the value of *val* becomes *false* at time $(t+1, 0, 0)$ at n's first child node [0,0,1,1,0,1,0]. Therefore the attribute values, which *final* at n depends on, at the nodes in the subtree rooted at n must be evaluated first.

At time $(2, 0, 0)$, the inner iteration is stopped when *val* at node [0,0,1,1,0,1,0] has a *false* value. After the first inner iteration is finished, the attribute evaluation for the

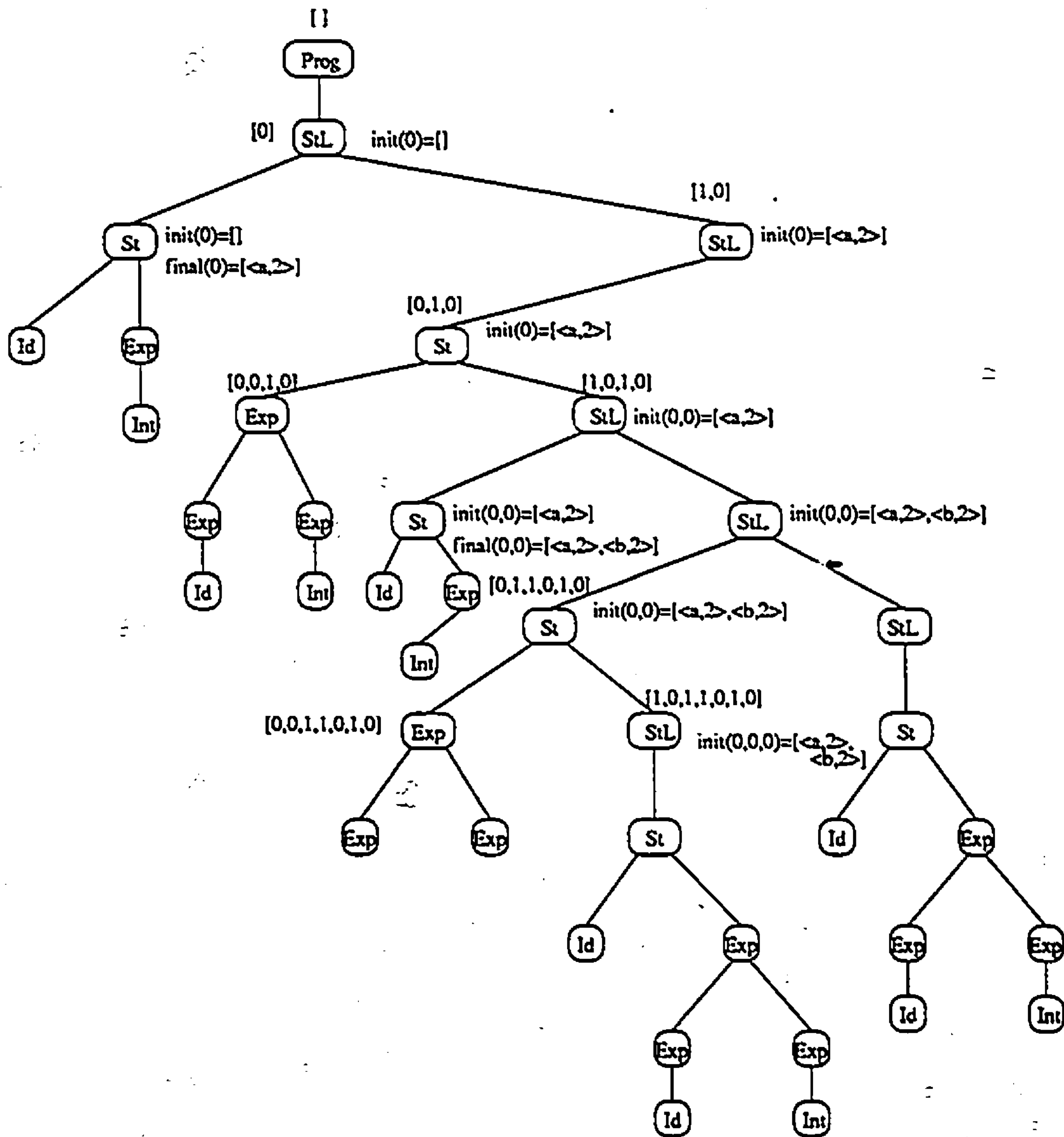

Figure 6.10: Branching new time dimensions for nested iterations

Figure 6.11: Reducing a time dimension when the inner iteration is finished

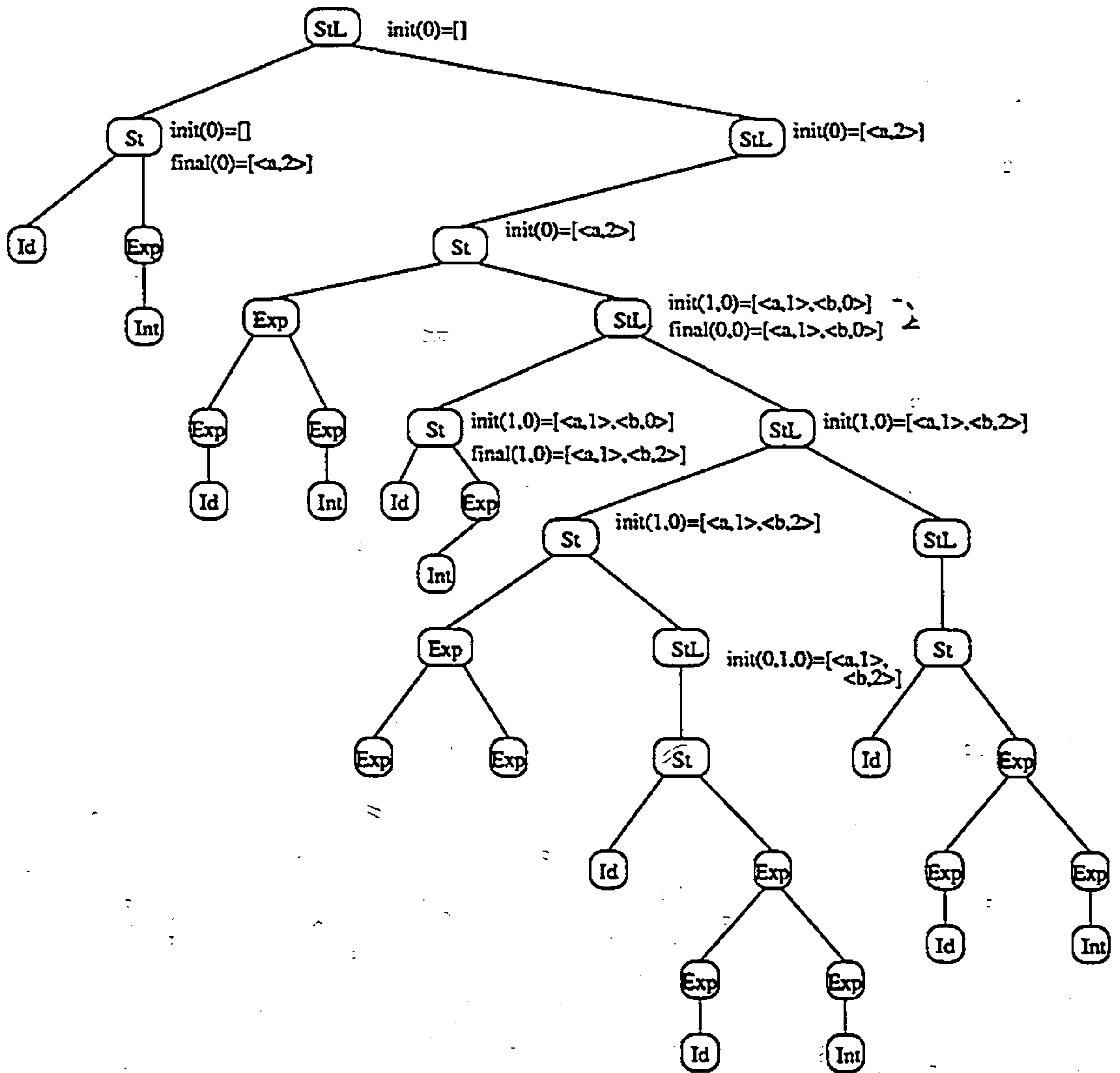

Figure 6.12: Increasing a time point for outer nested iteration

attribute at the outer *while*-statement which depends on the values of the inner *while*-statement will be started. Since at time (1.0) the value of *val* at node [0,0,1,0] is still true, the second iteration for the inner loop is started. Similarly, at time (2,1,0), the inner iteration is finished and at time (2,0) the value of *val* at node [0,0,1,0] becomes *false*. Eventually, the value of *finalenv* has its value [(a,0),(b,0)] at time (0).

6.3 Defining Aggregate Attributes

As we have introduced in Chapter 2, because of the locality of attribute definitions, conventional attribute grammars lack the power to specify the relationship between aggregate attributes and their elements directly. Using IAG, we can define the elements of an aggregate attribute as a sequence of values indexed by identifiers in a new dimension – the *identifier* dimension on the context space. Using identifier switching operators, the aggregation of the elements of an aggregate attribute value from other attribute values can be defined directly in the attribute definition.

A compiler typically consists of four basic phases: lexical analyzer, syntax analyzer, semantic analyzer, and code generator [ASU88]. There is also another very important component – symbol-table manager, which is a component of the semantic analyzer. An essential function of a compiler is to record identifiers used in the source program and to collect information about various attributes of each identifier, such as its type, scope, arguments, and so on.

When attribute grammars are used as compiler description languages, the structure of a compiler is slightly different. That is, the symbol-table manager is eliminated since attribute grammars do not allow global attributes. Symbol-tables in attribute grammars can only be defined as aggregate attributes whose values are monolithic data structures such as lists, which is what we did in the previous examples.

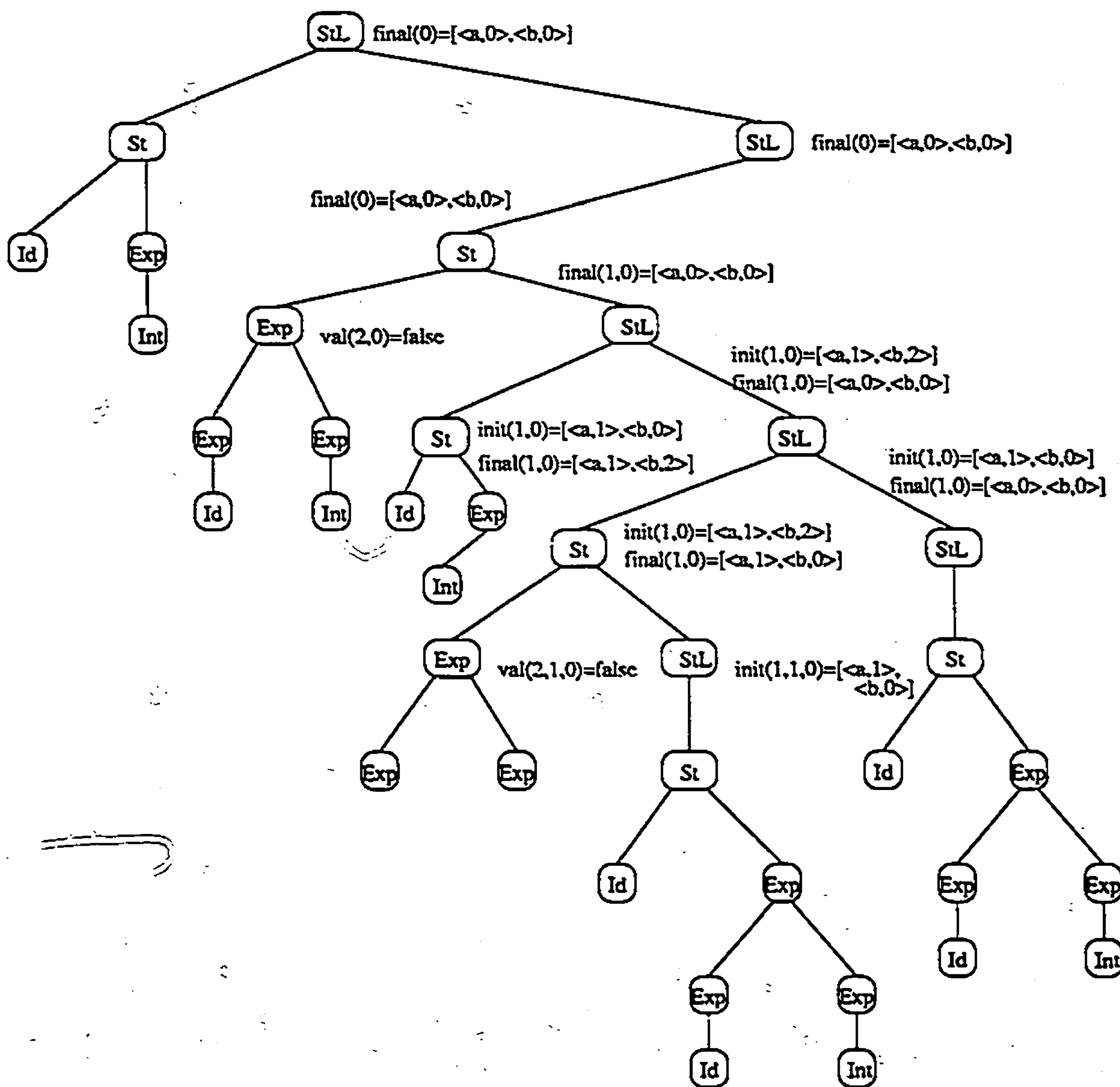

Figure 6.13: Reducing time dimensions when all the nested iterations are finished

6.3.1 Aggregating Attributes from an Attribute Value Tree

By using identifier switching operators, indexical attribute grammars can define aggregate attributes such as symbol tables directly. In the following we first define a special attribute *parsetree* whose tree shape is the reflection of a given parse tree. Then we use the indexical function *agg* defined in Chapter 3 to aggregate values according to the tree shape of *parsetree*.

Definition 6.1 (Special attribute parsetree) parsetree *is a special attribute symbol in IAG. The value of* parsetree *is an intension*

$$parsetree : \mathcal{T} \times \omega^{\omega} \times \mathcal{I} \rightarrow \{true, eod\}.$$

Given an IAG, let π be a parse tree generated by the IAG and $\hat{T}_\pi$ be the index tree of π, parsetree is defined by

$$parsetree(n, t, i) = \begin{cases} true & \text{if } n \in \hat{T}_\pi \\ eod & \text{otherwise} \end{cases}$$

Using the operator *agg*, we can define tables or lists as aggregate values of which each element has its own entry - the identifier point. When we want to use the value of an element of an aggregate value, we can access the element by its identifier point. For example, we can collect the type information about identifiers in a program. The following is a fragment of an IAG with aggregate attribute *typetable*:

Example 6.8 (Aggregating elements using identifier switching operators)

```
root child(no-error, 1);
% Global definitions
            type = if child(type, 0) eq child(type, 1)
                      then child(type, 0)
                      else Err
                   fi;

% Productions and local definitions
Program  -> DeclList StatList
             typetable = child(agg(type, id, parsetree),0);
```

```
DeclList -> Decl DeclList
          | .

Decl     -> Id ":" Id ";"
            type = child(lex, 1);
            id = child(lex, 0);

StatList -> Stat StatList
            no-error = child(no-error, 0) and child(no-error, 1);
          | .
            no-error = true;

Stat     -> Id "=" Exp ";"
            no-error = child(type, 1) eq
                       (child(id, 0) ati (root typetable));

Exp      -> Exp "+" Exp
          | Exp "-" Exp
          | Id
            type = child(id, 0) ati (root typetable);
          | Num
            type = "int";
          | CHAR
            type = "char";
```

The attribute definitions in Example 6.8 not only allow attribute values to be aggregated from an attribute value tree, but also allow an attribute to depend on the value of an individual element in an aggregate value. Fig 6.14 shows the attribute value dependencies in the attribute value tree of the following program fragment in the language defined by the above IAG.

```
...
a : int;
b : char;
...
a = 5;
b = 'A';
...
```

The attribute evaluation starts from demanding the value of *no-error* at node [1]. When the demands are sent to node [0,1] and [0,1,1], the evaluations for *no-error* at these

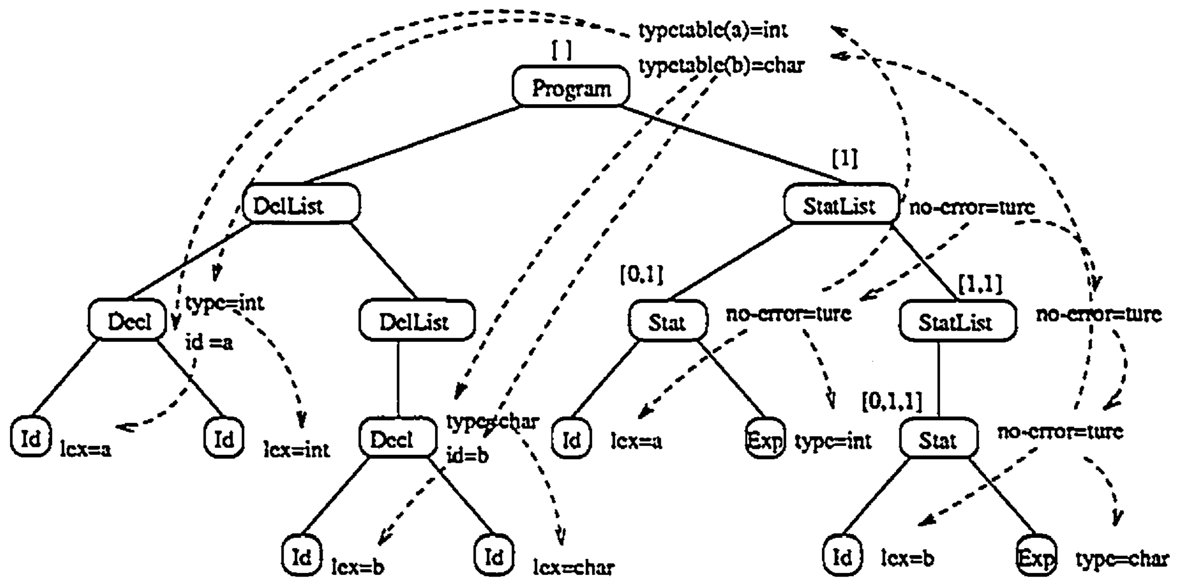

Figure 6.14: An attribute value tree with aggregate values

two nodes will send demands to *typetable* at the root node for the value of *typetable* at the identifier points *a* and *b*, respectively. Consequently, the aggregation for the type information in the declaration part of the tree is started.

From Figure 6.14 we can see that there are no partial values of *typetable* stored in the subtree of the declaration part of the program (Example 6.2 does) and no complete values of *typetable* are passed in the subtree of the statement part of the program.

The another advantage of defining aggregate value as non-monolithic values is that it has the potential to save time and space during incremental evaluation. During the first phase of evaluation, the data dependencies and demand dependencies can be detected dynamically. For example, we can keep the information of demanding attributes for *typetable* at an identifier point *a*. After the type declaration of *a* is modified, only the attributes which depend on the type of *a* will be reevaluated.

6.3.2 Aggregating Attribute Values from an Attribute Value Tree with Nested Structure

Since some programming languages allow nested blocks in their program structures, indexical attribute grammars should also allow the creation of symbol tables according to scope rules. That is, the elements collected from an outer block are not necessary to be included in the collection of an inner block. For example, in Figure 6.15, the information we collect from the outer block does not include the information in the inner block. Instead, the inner block should inherit the information in the outer block(s).

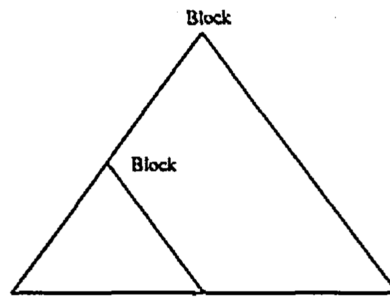

Figure 6.15: Nested structure of a program

To exclude the elements from the nested scope, we define a function *aggscope* (for "*aggregate in scope*"). The function *aggscope* only collects the values of elements from the current block, but not inner block(s).

aggscope is a 4-ary function $aggscope(x, y, tree, scope)$. The first three arguments are the same as the arguments of *agg*. The last argument is an intension that represents a nested scope. The function *aggscope* is basically the same as *agg* except that it does not collect the values from the subtree rooted by a node where *scope* has a non-*eod* value at the node.

aggscope can be defined in terms of the primitive indexical operators as follows.

```
aggscope(x, id, tree, scope) = child(element, 0)
  where
    element = if iseod(tree) or not iseod(scope) then eod
              elseif idindex eq id then x
              elseif not iseod(k) then k
```

```
                elseif child(element, head(index)+1)
                fi
                where k = child(element, 0) end;
  end;
```

The following example (extended from Example 6.7) shows how an indexical attribute grammar defines symbol tables as aggregate attributes using the identifier switching operators in nested block structures.

Example 6.9 (Aggregating elements from nested scopes)

```
root child(no-error,0)

% global definitions
              type = if child(type,0) eq child(type,1)
                        then child(type, 0)
                        else Err
                     fi;
              no-error = true;

% context-free grammar and local attribute definitions
Program  -> Block

Block    -> DeclList StatList
                no-error = child(no-error,0) and child(no-error,1);
                typelist = union(aggscope(type, id, parsetree, typelist),
                                 upasa typelist);

DeclList -> DeclList Decl
                no-error = child(no-error,0) and child(no-error,1);
              | .

Decl     -> Id "{" Block "}"
                no-error = child(no-error,1);
                type = "block";
                id = child(lex, 0);
              | Id ":" Id
                type = child(lex, 1);
                id = child(lex, 0);

StatList -> StatList Stat
                no-error = child(no-error,0) and child(no-error,1);
```

```
             | .

Stat      -> Id "=" Exp
               no-error = (child(lex,0) ati
                              (upasa typelist)) eq child(type,1);

Exp       -> Exp "+" Exp
             | Exp "-" Exp
             | Id
              type = child(lex, 0) ati (root typetable);
             | Num
              type = Num;
             | CHAR
              type = CHAR;
```

In the extended attribute grammar, the grammar symbol `Block` denotes a nested structure. The attribute *typelist* at a node associated with `Block` includes the information collected from its own scope and the information from its outer block. The function *union*(x, y) combines the two aggregate attributes x and y at every identifier point over the identifier dimension. If the element x_i has a non-*cod* value, *union* returns the value of x_i, otherwise *union* returns the value of y_i at the identifier point i. The process of aggregating the values of *typelist* is similar to the process in Example 6.5 except that at a node labeled by `Block` the value of *type* is aggregated from the corresponding scope, not from the entire parse tree.

In the parse tree of a program generated by the above grammar, the attribute *type*, which depends on the value of *typelist* at a node, will have the value of *typelist* at its first ancestor node on which the *typelist* has a non-*cod* value.

Figure 6.16 illustrates how the type information in different scopes is aggregated for the following program fragment, and how the identifiers obtain their type information from the corresponding scopes in the language defined by the above IAG.

```
...
a : int;
b : char;
...
a = b;
...
```

```
nest {
b : int;
...
a = b;
...
}
...
```

In Figure 6.16, the value of *typelist* is not a list of pairs. The type information of identifiers is stored as values at different identifier points in the identifier dimension of *typelist* at the node [0] and [*s*]. At node [0], *typelist* aggregates the value of *type* from the subtree rooted at node [0] except for the nodes in the subtree rooted at node [*s*] which denotes a nested block structure. At node [*s*], *typelist* aggregates the value of *type* from the subtree rooted at [*s*] and also gets the type information of these identifiers which are not defined in the inner block but the outer block. This process can be viewed as passing global information to a local environment.

6.3.3 Temporal Aggregate Attribute Values

The combination of the tree, time, and identifier dimensions and their associated node, time and identifier switching operators makes indexical attribute grammars more concise and expressive. The following example is a version of Example 6.3. In this example, the values of identifiers that denote constants are not stored at the root node as a list. The temporal definition is involved by the definition of the aggregate attribute *env* at the root node. At time 0, the values of *env* at the root node are initialized as *eod* at all the identifier points. At time t where $t > 0$, the values of *env* are the values of the constant identifiers aggregated at time $t - 1$. When the values of *env* do not change in time, the iterative aggregation terminates.

Example 6.10 (Type checking using all three kinds of indexical operators)

```
root finalenv

DefModule  -> ConstDecls
              finalenv = env asa (env eq next env);
```

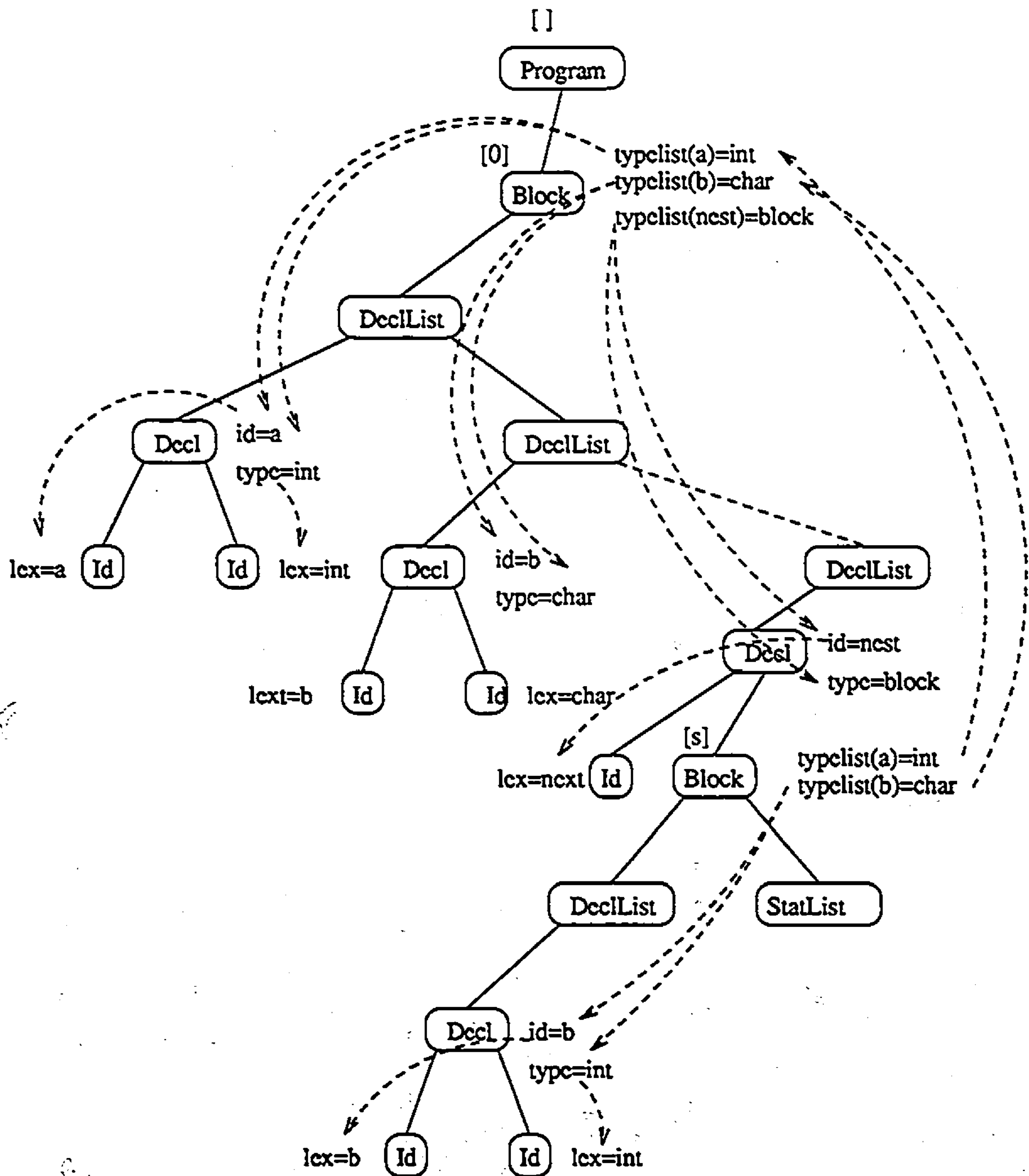

Figure 6.16: The nested structure of a program

```
                env = eod fby agg(id, val, parsetree);

ConstDecls -> ConstDecl ConstDecls
              | .

ConstDecl  -> Id "=" Expr ";"
              val = child(val, 1);
              id = child(lex, 0);

Expr       -> Expr "+" Expr
              val = child(val,0) + child(val,1);
              | Expr "-" Expr
              val = child(val,0) - child(val,1);
              | Id
              val = child(lex,0) ati (root env);
              | Num
              val = child(val,0);
```

For the same program fragment:

```
a = b + c;
b = c + 1;
c = 100;
```

in the following, we show the attribute value trees and the attribute dependencies at different time points in Figure 6.17 - 6.20. In these figures, the attribute values which are constants, such as the lexical values of identifiers, are not labeled by the time points.

At time 0, the aggregate values of *env* at the root node are all initialized as *eod*. Consequently, the attribute *val* at the nodes which depend on *env* will also have the value *eod*. Since the identifier c does not depend on any other identifiers, the value for identifier c is evaluated, which is 100.

At time 1, the *env* at the root node aggregates the values of *val* at time 0 from these nodes on which the value of *id* is not *eod* and stores the values at different identifier points according to their *id* values. Also, since the value of identifier c at time 0 is 100, at time 1, the value of *env* at identifier point c is 100. The attribute *val* at the nodes which depend on *env* at identifier point c will have the value 100 also. Consequently, the value for the identifier b, which is 101, is evaluated.

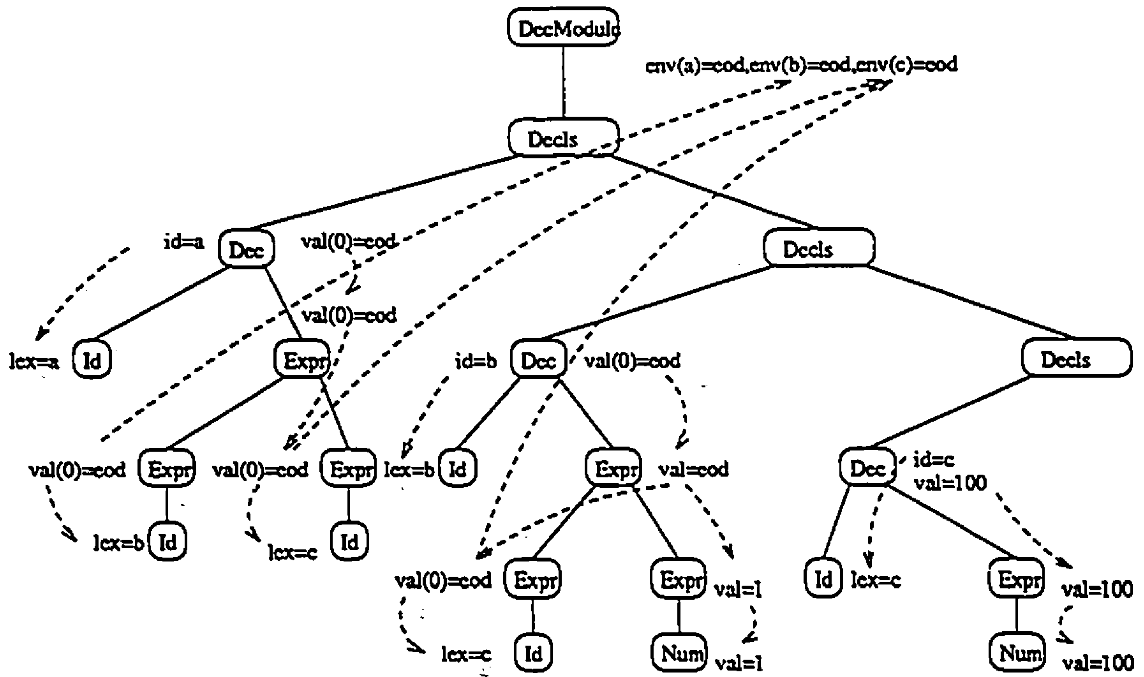

Figure 6.17: The aggregation at time 0

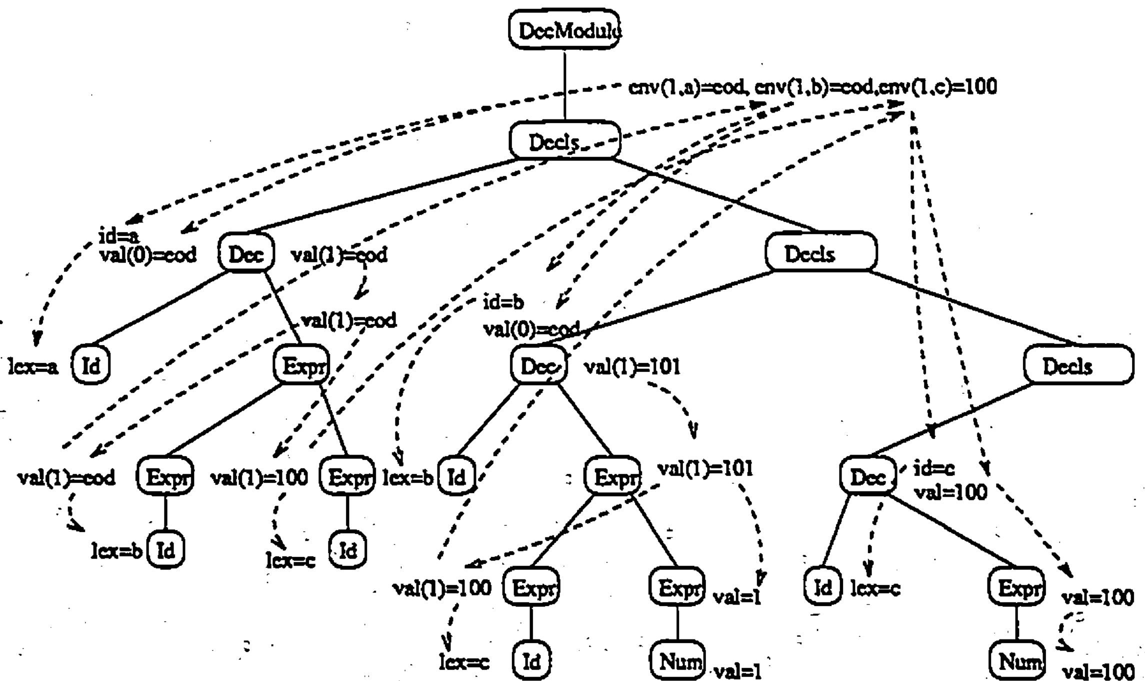

Figure 6.18: The aggregation at time 1

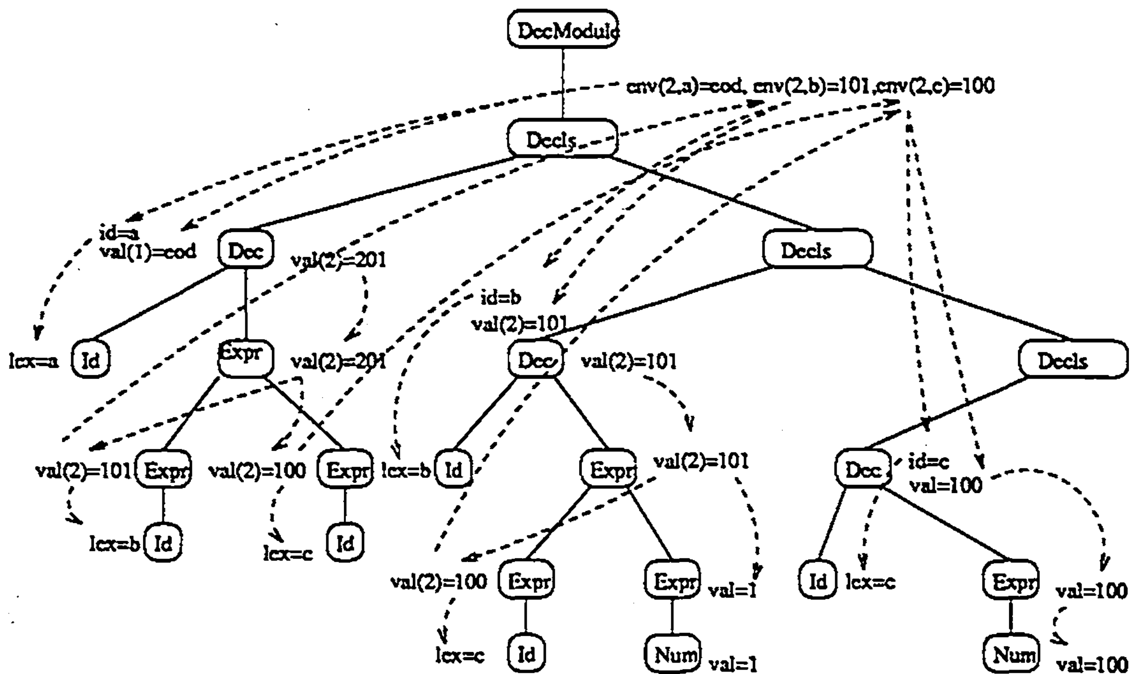

Figure 6.19: The aggregation at time 2

Similarly, at time 2, the *env* at the root node aggregates the values of *val* at time 1 from these nodes at which the value of *id* is not *eod* and stores the values at different identifier points according to their *id* values. Also, since the value of identifier b at time 1 is 101, at time 2, the value of *env* at identifier point b is 101. The attribute *val* at the nodes which depend on *env* at identifier point b will have the value 101 also. Consequently, the value for the identifier a, which is 201, is evaluated.

At time 3, the *env* at the root node aggregates the values of *val* at the nodes, which have non-*eod* values. After time 3, the value of *env* will not change. Therefore, the final value for *finalenv* is obtained, which has three non-*eod* values:

$$< 201_{([\,],3,a)}, 101_{([\,],3,b)}, 100_{([\,],3,c)}, \ldots > .$$

In the next chapter, we describe an implementation strategy for evaluating indexical attribute grammars.

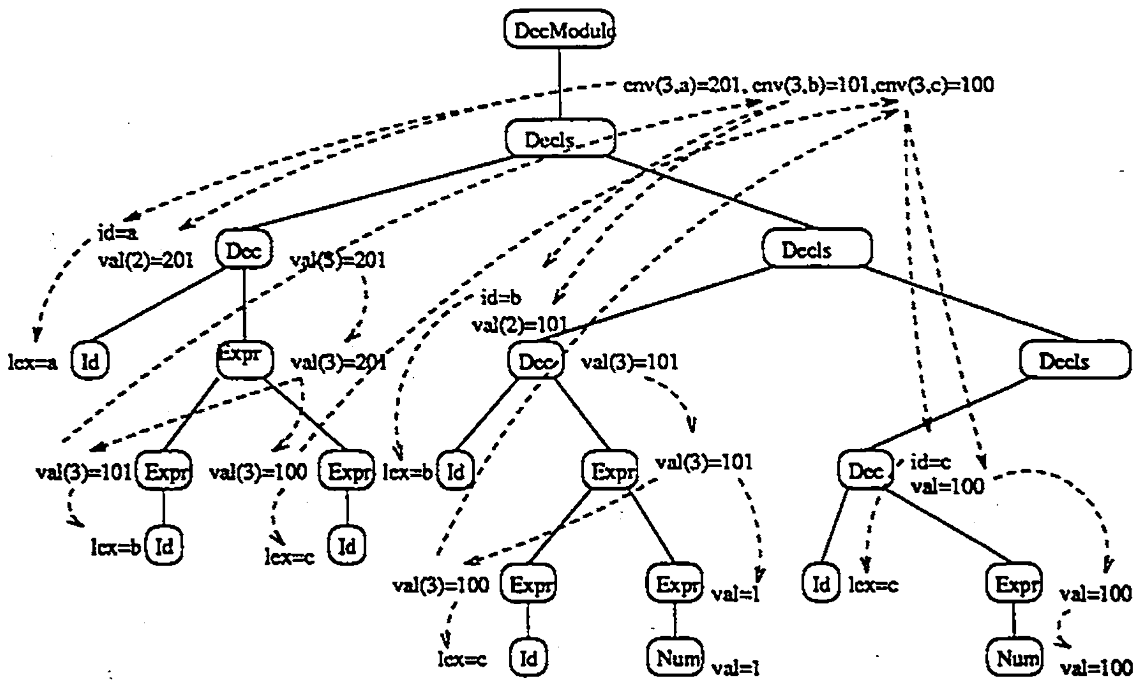

Figure 6.20: The aggregation at time 3

Chapter 7

AN IMPLEMENTATION STRATEGY

In this chapter we describe an implementation strategy for an indexical attribute grammar based compiler generator using the demand driven dataflow computation model. Many ideas presented in this chapter were implemented in my MSc thesis [Tao87].

7.1 The Demand-Driven Dataflow Computation Model

In a dataflow network (or graph) the nodes represent the operations and the arcs represent the data dependencies among the operations. The input values flow into the network through the input arcs and the result of the computations on the network flows out through an output arc.

In the demand-driven dataflow model, the evaluation for a dataflow network starts from demanding the value on the output arc. An operation is performed if and only if its result is demanded and all the input values required for performing the operation are available. If there is a demand for an operation but some of its operands are not ready on the incoming arcs, then demands for the data items will be sent out along the reverse direction of the incoming arcs.

Let us see how the demand-driven method works through a simple Lucid program

example.

```
R
where
  R = Q + S;
  Q = X + P;
  S = Y + P;
  P = X + Y;
end;
```

Here all the variables are constant in time.

The corresponding dataflow network for the above program is shown in Figure 7.1. At the initial time, all the arcs are empty, except the two input arcs labeled by X and Y. The evaluation starts with the demand for the output R. To compute R, the operation node needs two inputs - Q and S. Since Q and S have no valid values yet, two demands are sent out to request the values of Q and S. In turn, demands for P are sent out. The operation for P can be executed immediately, as its input data items are already available. After P is evaluated, the evaluation for Q and S can be started. Finally, the evaluation of R is performed.

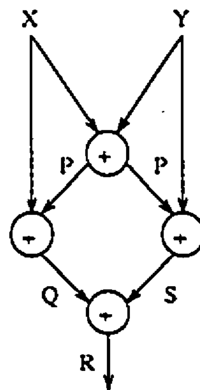

Figure 7.1: Demand driven computation on a dataflow network

In a demand-driven dataflow network, there are two kinds of information flow in opposite directions. One is the demand flow, which moves backwards through the network, and the other is the flow of data items which moves forwards through the network.

The demand-driven dataflow computation has two distinct characteristics. First, there

is no static evaluation order required. The evaluation order is created dynamically at evaluation time. Secondly, an operation is performed only if it is required; no unnecessary operations are performed in this model. These two properties of the demand-driven approach strongly support attribute evaluation.

The attribute instances in a given parse tree defined by IFADL expressions can be represented by a dataflow network. Recall the example given in Chapter 5.

```
root child(b, 0)
        %%
        a = 5
        %%
        X -> Y Z
             a = child(a, 0) + child(a, 1);
             b$0 = parent a;
```

If we consider a parse tree with only three nodes labeled by X, Y and Z, we can construct a corresponding dataflow network shown in Figure 7.2.

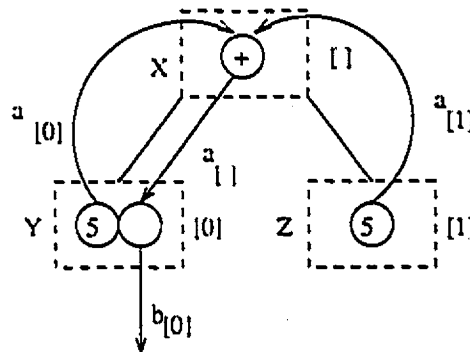

Figure 7.2: A dataflow network for the attribute instances in a parse tree

Here the dashed rectangles denote the nodes in the parse tree. The arcs represent the data dependencies among the attribute values. Since the node switching operators do not perform any computation other than switching contexts (or describing the data dependencies), node switching operators are not treated as operation nodes in a dataflow network. The evaluation for the attribute values on the network starts by demanding the value of the defining expression `root child(b, 0)`. Suppose a demand arrives for the value of b at [0], the definition of b at [0] is found, which is `parent a`. The node operator

parent switches the context from [0] to [ ]. Then the value of a at [ ] is required. To evaluate a at [ ], the values of a at [0] and [1] are demanded. Since a at [0] and [1] has value 5, the operation for a at [ ] is performed. Then the result flows out of the network at [0].

7.2 Structure of IAGCG

Using the demand driven approach, we can build a compiler generator IAGCG (for Indexical Attribute Grammar Compiler Generator) for languages defined by indexical attribute grammars. The inputs to IAGCG are language specifications, i.e. indexical attribute grammars, and the outputs are compilers for the languages defined.

IAGCG consists of two parts: an *indexed parse tree generator* and a *definition warehouse generator*. The parser generator generates a parser for a given indexical attribute grammar. The definition warehouse generator builds a definition warehouse for attribute definitions and the subject expression.

A compiler generated by the IAGCG consists of four components:

- a parser to generate parse trees for input language strings,
- a definition warehouse DW,
- an attribute value warehouse AW, and
- an eductive attribute evaluator EAE that is also a general purpose IFADL interpreter.

The parser constructs a parse tree for a language string. The parse tree is a concrete representation of the production tree $\mathcal{P}$ defined in Chapter 5. A node in a parse tree (production tree) has four fields: (*index*, *pnumber*, *firstchild*, *nextsibling*). The *firstchild* and *nextsibling* are pointers to link the first child and the next sibling of the node respectively. The *pnumber* contains an integer to indicate the production associated with the node, which can be obtained at parse tree building time. The number is used to find the local definition for an attribute instance at a node at evaluation time. After a parse tree

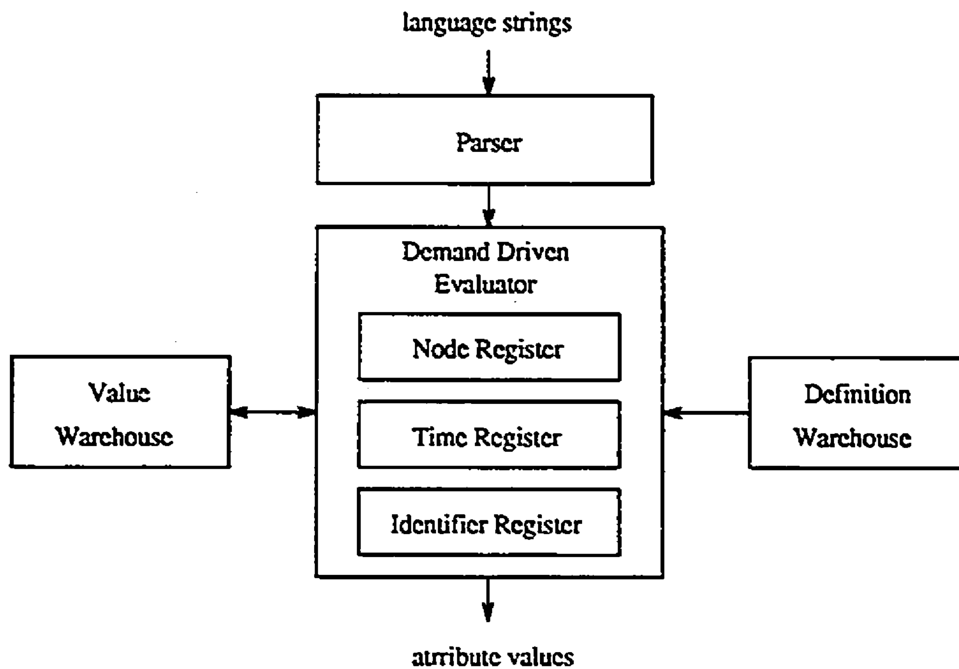

Figure 7.3: The structure of a compiler generated by IAGCG

is built, the parser assigns an index for each node in the parse tree following Definition 3.2. The algorithm to index a parse tree is as follows.

```
index-tree(Tn: a parse tree rooted by node n, i: index of n)
m: node;
j: integer;
begin
     assign index i to n
     for all child nodes m of n do
         index-tree(Tm, cons(j,i)) where m is n's jth child
end
```

The indexing starts from the root node of a given parse tree.

```
main().
T: root node of the parse tree;
begin
     index-tree(T, □);
end
```

The attribute definitions are stored in the *definition warehouse* *DW*. The definition

warehouse consists of three parts:

- a defining expression warehouse DEW,
- a global definition warehouse GDW, and
- a local definition warehouse LDW.

The DEW has only one cell that stores the defining expression whose value is the semantics of a given parse tree.

The GDW stores global attribute definitions for an indexical attribute grammar. Each cell of the GDW stores one of the global attribute definitions. Since the definitions are valid for all the nodes of a parse tree, a table entry or a tag of a cell of the GDW only consists of an attribute symbol.

The LDW stores local attribute occurrence definitions for an indexical attribute grammar. A definition of an attribute occurrence in a production is stored in a cell of the LDW. An entry or a tag of a cell of the LDW therefore consists of three elements: an attribute symbol a, a production number p, and a position number of the non-terminal symbol x in the production p.

There is also an attribute value warehouse (AW) which stores the values of attributes. For a given parse tree π, the AW can also be viewed as implementing the attribute value tree $\mathcal{V}_\pi$. At initial time, all the cells of the AW are initialized with value $\perp$. The tag associated with each cell of the AW consists of four components: (*attribute symbol, node, time, identifier*) where *time* is a sequence of integers that represent the relevant time points in the multi-time dimensions. When a cell of AW has a non-$\perp$ value, the value is not allowed to be changed.

7.3 The Eductive Attribute Evaluator EAE

EAE is the heart of IAGCG. The evaluation for a given parse tree starts by evaluating the defining expression E, whose value is the semantics of the language string corresponding

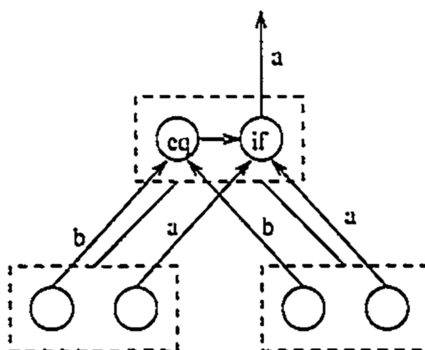

Figure 7.4: A dataflow network

to the tree. The following is an algorithm for evaluating a parse tree, where we assume that the E is evaluated only at context ([], 0, " ").

The evaluation of attribute instances on a given parse tree π is considered as the evaluation of the corresponding value tree $\mathcal{V}_\pi$. Since attribute occurrence definitions of an attribute grammar are stored in the definition warehouse with tags, the occurrence definition tree $\mathcal{O}_\pi$ and the attribute definition tree $\mathcal{D}_\pi$ do not have to be built. To obtain the definition for an attribute instance, we first give proper tags to the local definition warehouse LDW. If there is no definition stored with the given tags, then we look at the global definition warehouse GDW.

Consider an attribute a at a production that is defined by

```
X -> Y Z
     a = if (child(b, 0) eq child (b, 1))
         then child(a,0)
         else child(a,1);
```

We also assume that a is a constant in time. The definition can also be viewed as a dataflow network (Figure 7.4).

When the value of a is demanded at a context (n, t, i) and the definition of a at n is the above `if` expression, a demand to its first operand (or a demand to the output of the operator eq) is issued. Since eq is a strict operator, two demands for evaluating b at the contexts `((cons(head(n)+1),tail(n)),t,i)` and `((cons(head(n)+2),tail(n)),t,i)` are sent out. If the result of the first operand of the `if` expression is true, the demand for

the value of a at the context ((cons(head(n)+1),tail(n)),t,i) is send out. Otherwise, the demand for the value of a at the context ((cons(head(n)+2),tail(n)),t,i) is issued.

The EAE will repeat the process until the value of the defining expression is evaluated.

Chapter 8

CONCLUSIONS

8.1 Summary

This dissertation defines an attribute grammar system – Indexical Attribute Grammars (IAG), based on the indexical programming paradigm.

In IAG, an attribute is an intension – a function from the indexical context space to the domain of data values. The indexical context space is defined as

$$\mathcal{C} : \mathcal{T} \times \omega^{\omega} \times \mathcal{I}$$

where $\mathcal{T}$ is the tree dimension, ω^{ω} is the multitime dimension, and $\mathcal{I}$ is the identifier dimension. We say that attributes in IAG vary over the indexical context space $\mathcal{C}$. They may have different values at different contexts.

An attribute grammar in IAG is defined in three parts: a defining expression that denotes the semantics of a parse tree generated by the grammar, an optional set of global attribute definitions, and a context-free grammar with local attribute definitions associated with each production.

To define relationships among attribute values on the context space $\mathcal{C}$, IAG provides a set of primitive indexical or context switching operators. The operators can be divided into four subsets: node switching operators, time switching operators, identifier switching operators, and multidimension switching operators. There are also operators which switch

contexts into different dimensions.

Node switching operators allow the user to define *non-local* attributes, in terms of the values of attributes at different, remote nodes of the parse tree.

Time switching operators allow the user to define *temporal* attributes, in terms of the values of other attributes or itself at different time points. If a temporal attribute is defined by an iterative algorithm, the value of the attribute at different time points can be viewed as the value at different steps in the iteration.

Identifier switching operators, combined with node switching operators, allow the user to define aggregate attributes. Aggregate attributes can aggregate their elements (other attribute values), indexed by identifiers directly from any part of the parse tree. Individual elements of an aggregate attribute can also be referred to directly by the definitions of other attributes on the parse tree.

The indexical attribute evaluator is based on the eductive computation model. Given a parse tree, the evaluation starts by sending demands for the value of the defining expression of the tree. To compute the initially demanded attribute value, demands for values of other attributes on the tree will be issued and the demanded values will be computed, until the initial demand is satisfied. In the eductive computation model, no value is computed until it is needed. Consequently, no unnecessary attribute evaluation is performed.

8.2 Contributions and Advantages

- **Denotational semantics**

  The first advantage of IAG is that it has clearly defined denotational semantics based on least fixed point theory. The formal semantics of an attribute grammar in IAG is the denotation of each parse tree π generated by the grammar, that is, the least fixed point of the function that maps the intension $\mathcal{V}_\pi$ (the attribute value tree) to itself.

- **AG-independent attribute definition language**

  IFADL is designed to be an *attribute definition language*. Attributes defined by

IFADL expressions are IFADL variables. The meaning of an attribute is an intension on the indexical context space C. The semantics of IAG directly corresponds to the IFADL program formed by the attribute definitions of IAG. Thus the denotational semantics of IAG can be defined clearly in terms of IFADL's semantics.

IFADL is independent of the concept of AG. There are several advantages to using an AG-independent attribute definition language. Firstly, we do not need to build special attribute evaluators. The implementation of IFADL on a given system can be adapted as an evaluator of IAG. Secondly, it makes IAG executable. That is, given a parse tree generated from an AG, its attribute definitions determine an executable IFADL program. Thirdly, the expressive power of IAG increases as IFADL evolves. On other the hand, IAG provides an important type of application for IFADL, so that the language can be developed and specialized according to the needs of IAG.

- **Improved expressive power**

  - **Solution for the communication attribute problem**

    Indexical attribute grammars allow us to remove most of the communication attributes without losing the declarative property. IAG relaxes the locality restriction of attribute definitions by introducing indexical semantics to attributes. Using indexical operators (in particular, node-switching operators) to specify attribute definitions, an attribute can directly depend on other attributes. Meanwhile, we still keep the declarative property of attribute definitions and give them precise denotational semantics.

    By eliminating communication attributes, we can also improve the space efficiency, since there is no need to store duplicated attribute values.

  - **Solution for the aggregate attribute problem**

    We solve the aggregate attribute problem within the declarative semantics and without introducing side-effects in the attribute grammar definition level. We consider that an aggregate attribute value is distributed, that is, its elements can be collected and referred to individually. Thus, both conceptually and

practically, we do not need to carry the entire aggregate structure around the parse tree.

- **Time varying attributes**

  As a formal specification tool, IAG can specify systems whose values involve data streams. IAG can specify a system with the structure of *a tree of data streams*. That is, given a parse tree generated by an attribute grammar in IAG, the tree structure is fixed and does not change with time, but the attribute values at nodes of the tree change with time.

  Using the time varying attributes, IAG allows the user to define the circular attributes in conventional AGs non-circularly. Such definitions are independent of the underlying implementation.

  IAG can also specify a system with the structure of *a stream of trees*. That is, we can define attributes in IAG to associate their tree-shapes to the structure of the system. At different times, the attributes may have different tree-shapes. The tree-shapes of one or a union of a set of attributes at all the time points constitute a stream of trees.

- **Parallel attribute evaluation**

  AGs have great potential for parallel execution since an attribute dependency graph is a special instance of a dataflow graph [Far83][KG92]. The IAG system can be implemented based on the eduction model – a parallel implementation model for indexical languages and systems. In this implementation, no other special parallel attribute evaluation method is needed, besides eduction which is suited to handle fine-grained parallelism. The correctness of the parallel evaluation of IAG is guaranteed by the model, no matter what order attributes are evaluated, because of their declarative semantics without side effect. Also, because of elimination of communication attributes and monolithic aggregate attributes, these parallel evaluations could be efficient.

8.3 Future Work

- **Incremental evaluation**

  By adding a *version* dimension in IAG, we can consider the value of an attribute as a function of *version*. That is, after each modification indexed by a version point, the attributes whose values have been changed at certain nodes by the modification, will have the updated values at these nodes at the version point corresponding to the modification. Such IAGs will not only allow incremental evaluations, but will also have the power to simultaneously represent different versions of a program. In particular, the *history* of the modifications to a program can be represented by the sets of attribute values, each of which corresponds to a version of the program.

- **Higher-order attribute grammars**

  In higher-order attribute grammars [VSK89][TC90] the value of an attribute can be a structured tree. This can be done in IAG by adding an extra tree dimension to the context space in IAG to represent the nested tree structures in the nested tree dimensions. For any tree structure can be represented by the tree shape of an attribute.

  A more general higher order extension of IAG is to allow attributes at nodes to have function values of arbitrarily high orders. There are two kinds of possible function values for attribute instances at nodes. The first is pointwise functions that are pure higher order functions in terms of functional languages without any indexical operators and context switching involved. The second is so-called synchronic functions whose meaning depends on the underlying context but which do not involve any context switching operations. Since Wadge has solved the problem of evaluating higher order functions in Lucid-like indexical languages [Wad91], the two kinds of higher order attributes can be implemented in higher order indexical attribute grammars using the same scheme. To allow attribute values to be general indexical functions that involve context switching, we need further to investigate their semantics and implementation strategies. If such functions are associated only with

individual nodes, the question is what are their meanings after contexts have been switched, and how to evaluate the values of these functions at different nodes.

- **Extendible expressive power with customized context space**

  The context space of IAG with the associated context-switching operators can be designed according to the application. Thus, there is only one base dimension in all IAG, that is, the tree dimension. Other user-defined dimensions with certain structures of IAG can be introduced into the context space, according to the problem it solves.

- **Natural language processing**

  In Montague Grammars [DWP80], the meaning of a natural language sentence is a truth-value that is determined under certain conditions. These conditions may depend on the sentence in various possible states of affairs, at various times, in various places, for various speakers, and so on.

  It is natural to think that we may be able to specify Montague grammars in terms of IAG, because both are based on possible world semantics. To do so, we need to extend or generalize IAG in several aspects. We need to extend the context-free grammar of indexical attribute grammars to cover the syntax of Montague grammars. We need to define the context space to consist of, at least, all the dimensions that are required by Montague grammars plus the general tree dimension. We also need to allow attributes at nodes of a structure tree generated by a Montague grammar to have function values. Using such IAGs, we may be able to build an interpreter that could evaluate the semantics of natural languages sentences based on Montague grammars.

Bibliography

[AFH85] E. A. Ashcroft, A. A. Faustini, and B. Huey. Eduction – a model of parallel computation and the programming language lucid. In *the Phoenix Conference on Computers and Communications, IEEE*, pages 9–15, 1985.

[AJ86] E. A. Ashcroft and R. Jagannathan. Operator nets. In Woods J. V., editor, *Fifth Generation Computer Architecture*, pages 177–202. 1986.

[Alb89] H. Alblas. Iteration of transformation passes over attributed program tree. *Acta Informatica*, 27:1–40, 1989.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, 1988.

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis, part 1: Exhaustive analysis, part 2: Demand analysis. *Acta Informatica*, 10(3):245–272, October 1978.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communication of the ACM*, 19(2):55–62, 1976.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 274–284, 1988.

[Den80] J. B. Dennis. Data flow supercomputer. *Computer*, 13(11):48–56, 1980.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars.* Springer-Verlag, 1988. Lecture Notes in Computer Science **323**.

[DWP80] D. Dowty, R. Wall, and S. Peters. *An Introduction to Montague Semantics.* Dordrecht, Holland, 1980.

[Far83] R. Farrow. Attribute grammar and dataflow language. Technical Report Technical Report, Department of Computer Science, Columbia University, 1983.

[Far84] R. Farrow. Generating a production compiler from an attribute grammar. *IEEE Software*, 1(4):77–93, 1984.

[Far86] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *SIGPLAN Notices*, 21(7):85–98, 1986.

[Fro92] R. A. Frost. Constructing programs as executable attribute grammars. *The Computer Journal*, 35:376–387, 1992.

[FW86] A. A. Faustini and W. W. Wadge. Intensional programming. Technical Report DCS-55-IR, Department of Computer Science, University of Victoria, 1986.

[GF82] M. Ganapathi and C. N. Fischer. Description-driven code generation using attribute grammars. In *the Ninth ACM Symposium on Principles of Programming Languages*, pages 108–119, 1982.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *ACM SIGPLAN' 84 Symp. on Compiler Construction*, pages 172–184, 1984.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[HK86] S. E. Hudson and R. King. Implementing a user interface as a system of attributes. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 143–149, 1986.

[Hoo86] R. Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 14–25, 1986.

[JF84] G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *the Eleventh ACM Symposium on Principles of Programming Languages*, pages 141–151, 1984.

[Joh87] T. Johnsson. *Attribute Grammars as a Functional Programming Paradigm.* Springer-Verlag, 1987. Lecture Notes in Compter Science, 274.

[Jon90] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transaction on Programming Langages and Systems*, 12(3):429–462, 1990.

[JS86] Larry G. Jones and J. Simon. Hierarchical vlsi design systems based on attribute grammars. In *the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 58–69, 1986.

[Kai89] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, 1989.

[Kat84] T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, 1984.

[KG92] A. Klaiber and M. Gokhale. Parallel evaluation of attribute grammars. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):206–220, 1992.

[Knu68] D. E. Knuth. Semantics of context-free language. *Math. System Theory*, 2(2):127–145, 1968.

[Knu71] D. E. Knuth. Semantics of context-free language: Correction. *Math. System Theory*, 5(1):95–96, 1971.

[Kos85] K. Koskimies. A note on one-pass evaluation of attribute grammars. *BIT*, 25:438–450, 1985.

[KS76] K. Kennedy and Warren S. K. Automatic generation of efficient evaluators for attribute grammars. In *the Third ACM Symposium on Principles of Programming Languages*, pages 32–49. 1976.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[Mue85] E. M. M. Mueghal. Controlled natural language interfaces: the best of three worlds. In *ACM Computer Science Conf.*, 1985.

[Pla86] W. J. Plath. Request: a natural language question-answering-system. *IBM Journal of R&D*, 20(4):326–335, 1986.

[Rep84] T. Reps. *Generating Language-Based Environments*. The MIT Press, 1984.

[RMT86] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language–based editors. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 1–13, 1986.

[RPJ94] G. Roussel, D. Parigot, and M. Jourdan. Coupling evaluators for attribute coupled grammars. In *Proceedings of Compiler Compilers*, pages 52–67, 1994.

[RT86] K. Raiha and J. Tarhio. A globalizing transformation for attribute grammars. In *ACM SIGPLAN' 86 Symp. on Compiler Construction*, pages 74–83, 1986.

[SEFR89] S. Sagiv, O. Edelstein, N. Francez, and M. Rodeh. Resolving circularity in attribute grammars with applications to data flow analysis. In *the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 36–45, 1989.

[Tao87] S. Tao. AGCG – an attribute grammar compiler generator. Master's thesis, University of Victoria, B.C., Canada, 1987.

[TC90] T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 197–208, 1990.

[Tho74] R. Thomason, editor. *Formal Philosophy*. 1974. Yale University Press.

[TR81] T. Teitelbaum and T. Reps. The cornell program synthesizer a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[TW89] W. Thome and R. Wilhelm. Simulating circular attribute grammars through attribute reevaluation. *Information Processing Letters*, 33:79–81, 1989.

[van88] J. van Benthem. *A Manual of Intensional Logic*. Springer-Verlag, 1988. Lecture Notes.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 131–145, 1989.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, England, 1985.

[Wad81] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[Wad91] W. W. Wadge. Higher order lucid. In *4th International Symposium on Lucid and Intensional Programming*, pages 62–69, 1991.

[WJ88] J. A. Walz and G. F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 209–221, 1988.

[Yag84] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, University of Warwick, Coventry, UK, 1984.

Appendix 1

Syntax of IFADL

```
program    -> expr

expr       -> constant
              | identifier
              | "(" expr ")"
              | expr infix-op expr
              | prefix-op expr
              | list-expr
              | if-expr
              | case-expr
              | fun-call
              | w-clause
              | "eod"
              | "error"

if-expr    -> "if" expr "then" expr
              | "else" expr "fi"

case-expr  -> "case" expr "of" cbody "end"

cbody      -> cbody defacase
              | expr ":" expr ";" cbody
              | expr ":" expr ";"

defacase   -> "default" ":" expr

fun-call   -> identifier "(" act-list ")"

act-list   -> expr
```

```
               | expr "," act-list

w-clause    -> expr "where" defn-list "end"

defn-list   -> defn
               | defn defn-list

defn           -> identifier "=" expr ";"
               | identifier "(" formals ")" "=" expr ";"

formals     -> identifier
               | identifier "," formals
```

Appendix 2
An Indexical Attribute Grammar for Type Checking

```
%
% This is an indexical attribute grammar for a simplified Modula-2
% language. It tests variables within a module with only simple types.
%

root accept

% Global attribute definitions
        accept = child(accept,0);
        type = child(type,0);

% Context-free grammar and local attribute definitions

program: "MODULE" ident ";" block ident "."
        accept = if child(id,0) eq child(id,2)
                 then child(accept,1)
                 else FALSE
                 fi;
        % typetable at the node labeled by program equals the
        % typetable at the first child of the node labeled by block.
        typetable = child(child(typetable,0),1);

block : decl_list_opt begin_and_stmts_opt "END"
        accept = child(accept,0) and child(accept,1);
        % collect type information for typetable at the node labeled
```

```
        % by decl_list_opt.
        typetable$0 = agg(decl_type, id, parsetree);

decl_list_opt :
        accept = TRUE;
        | decl_list

decl_list : decl
        | decl_list decl
        accept = child(accept,0) and child(accept,1);

decl :  "VAR" var_decl_list_opt
        accept = TRUE;
        | proc_decl ";"

var_decl_list_opt :
        | var_decl_list

var_decl_list : var_decl
        | var_decl_list var_decl

var_decl : field_decl ";"

field_decl : ident_list ":" type
        decl_type$0 = parent child(lex,1);

ident_list_opt:
        | ident_list
        decl_type$0 = parent decl_type;

ident_list : ident
        decl_type$0 = parent decl_type;
        | ident_list "," ident
        decl_type$0 = parent decl_type;
        decl_type$1 = parent decl_type;

proc_decl : proc_heading ";" block ident
        % if the identifier in the proc_heading is equal to the
        % identifier of the block then accept is true.
        accept = if child(child(id,0), 0) eq child(id,2)
                then child(accept,1)
                else FALSE
                fi;
```

```
        % union the type information from the proc_heading,
        % the body of the procdure, and the upper block.
        typetable = union(union(child(typetable,0),
                    child(child(typetable,0),1)),
                    upasa typetable);

proc_heading :  "PROCEDURE" ident formal_para_opt
        % collect type information of parameters.
        typetable = child(agg(decl_type, id, parsetree), 1);

formal_para_opt :
        | formal_para

formal_para : "(" fp_section_list_opt ")"

fp_section_list_opt :
        | fp_section_list

fp_section_list : fp_section
        | fp_section_list ";" fp_section

fp_section : var_opt ident_list ":" type
        decl_type$1 = parent child(lex, 2);

var_opt :
        | "VAR"

begin_and_stmts_opt :
        accept = TRUE
        | "BEGIN" stmt_list
        accept = child(accept,1);

stmt_list : stmt
        | stmt_list ";" stmt
        accept = child(accept,0) and child(accept,1);

stmt :
        accept = TRUE;
        | assignment
        | if_stmt
        | while_stmt
        | repeat_stmt
        | loop_stmt
```

```
        | for_stmt

assignment : designator ":=" expr
        accept = child(type,0) eq child(type,1);

if_stmt : "IF" expr "THEN" stmt_list elsif_list_opt else_opt "END"
        accept = if child(type,0) eq BOOLEAN
                 then child(accept,1) and
                      child(accept,2) and
                      child(accept,3)
                 else FALSE
                 fi;

elsif_list_opt :
        accept = TRUE;
        | elsif_list

elsif_list : elsif
        | elsif_list elsif
        accept = child(accept,0) and child(accept,1);

elsif : "ELSIF" expr "THEN" stmt_list
        accept = if child(type,0) eq BOOLEAN
                 then child(accept,1)
                 else FALSE
                 fi;

else_opt :
        accept = TRUE;
        | else

else :  "ELSE" stmt_list

loop_stmt : "LOOP" stmt_list "END"

while_stmt : "WHILE" expr "DO" stmt_list "END"
        accept = if child(type,0) eq BOOLEAN
                 then child(accept,1)
                 else FALSE
                 fi;

repeat_stmt : "REPEAT" stmt_list "UNTIL" expr
        accept = if child(type,1) eq BOOLEAN
```

```
                then child(accept,0)
                else FALSE
                fi;

for_stmt : "FOR" designator ":=" expr "TO" expr "DO" stmt_list "END"
        accept = if child(type,0) eq child(type,1) and
                   is_simple_type(child(type,2))
                 then child(accept,3)
                 else FALSE
                 fi;

expr_opt :
        type = NON_TYPE;
        | expr

expr :  simple_expr
        | simple_expr relation_op simple_expr
        type = checktyperel(child(type,0), child(lex,1),
                            child(type,2));

simple_expr : sign_opt term
        type = checktypeop1(child(op,0), child(type,1));
        | simple_expr add_operator term
        type = checktypeop2(child(type,0), child(lex,1),
                            child(type,2));

sign_opt :
        op = NON_OP;
        | sign_op
        op = child(lex,0);

term :  factor
        | term mul_operator factor
        type = checktypeop2(child(type,0), child(lex,1),
                            child(type,2));

factor : number
        | ident
        type = child(id,0) upasa typetable;
        | string
        type = STRING;
        | "(" expr ")"
        | "NOT" factor
```

```
        type = if child(id,0) eq BOOLEAN
               then BOOLEAN
               else Err;

designator : ident
        type = child(id,0) upasa typetable;

ident : identifier
        id = child(lex,0);

number : integer
        type = INTEGER;
        | real
        type = REAL;
```

VITA

Surname: Tao	Given Names: Senhua
Place of Birth: Beijing, China	Date of Birth: June 27, 1955

Educational Institutions Attended:

Beijing Polytechnical University	1979 to 1983
University of Victoria	1985 to 1990

Degrees Awarded:

B.Sc.	Beijing Polytechnical University	1983
M.Sc.	University of Victoria	1987

Honours and Awards:

B.C. Advanced Systems Scholarship	1987
University of Victoria Fellowship	1985-1990

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my dissertation to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this dissertation for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying of this dissertation for financial gain shall not be allowed without my written permission.

Title of Dissertation:

INDEXICAL ATTRIBUTE GRAMMARS

Author: ______________________________

Senhua Tao

September 14, 1994