HARRY LEGS
A Syntax-Directed Programming Environment Generator

By

Jeffrey Palm

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE

UNIVERSITY OF FLORIDA

2000
Copyright 2000

by

Jeffrey Palm
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** ........................................................................................................ Error! Bookmark not defined.

**LIST OF TABLES** ............................................................................................................... vii

**LIST OF FIGURES** ........................................................................................................... viii

**ABSTRACT** ........................................................................................................................ xi

**CHAPTER 1** ...................................................................................................................... 1
Introduction .......................................................................................................................... 1

**CHAPTER 2** ...................................................................................................................... 4
Background and Review of Previous Work ........................................................................... 4

- 2.1 Definitions ....................................................................................................................... 4
- 2.2 Editing Paradigms .......................................................................................................... 4
- 2.3 Text Editors ................................................................................................................... 4
- 2.4 Structured Editors ......................................................................................................... 5
  - 2.4.1 Generative mode ..................................................................................................... 5
  - 2.4.2 Analytic mode ....................................................................................................... 6
  - 2.4.3 Hybrid mode ....................................................................................................... 6
  - 2.4.4 Design Issues of Structured Editors .................................................................... 6
  - 2.4.5 Arguments for Structured Editors ...................................................................... 8
  - 2.4.6 Arguments against Structured Editors ............................................................... 9
- 2.5 Context-Free Grammars ............................................................................................. 9
- 2.6 LR Parsing .................................................................................................................... 10
  - 2.6.1 General LR Parsing ............................................................................................ 10
  - 2.6.2 Selective LR Parsing ......................................................................................... 11
  - 2.6.3 Incremental Parsing ............................................................................................ 11
- 2.7 Evaluation Environments and Type Systems ............................................................ 12
  - 2.7.1 Strong Typing Versus Weak Typing .................................................................... 12
  - 2.7.2 Static Typing Versus Dynamic Typing ............................................................... 13
  - 2.7.3 Monomorphic Versus Polymorphic Types ......................................................... 13
  - 2.7.4 Static Binding Versus Dynamic Binding ............................................................ 14
  - 2.7.5 Treatment of Functions ..................................................................................... 15
  - 2.7.6 Other Issues in Type Systems ............................................................................ 16
- 2.8 Run Time Function Dispatch ...................................................................................... 17
  - 2.8.1 Table-Based Dispatch ......................................................................................... 18
  - 2.8.2 Compressed n-dimensional tables ..................................................................... 18
  - 2.8.3 Multiple row displacement ................................................................................. 19
  - 2.8.4 Cache-Based Techniques .................................................................................... 19
  - 2.8.5 Handling Cross-Type Operations ...................................................................... 20
**CHAPTER 3** ...................................................................................................................... 23
The Original Parse Tree Library and Modifications Made to this Library .............................. 23
6.2 Implementation of the Environment System ...................................................... 56
   6.2.1 The HashNode Class..................................................................................... 56
   6.2.2 The HashTable Class ................................................................................... 59
   6.2.3 The Env Class............................................................................................... 60
   6.2.4 The EnvException Class and Subclasses.................................................... 61
   6.2.5 Evaluation of a Parse Tree using the Env Class ......................................... 62

CHAPTER 7 ................................................................................................................. 64
Multiple Dispatch System.............................................................................................64
   7.1 Goals of the Dispatch System ............................................................................. 64
   7.2 Implementation of the Dispatch System............................................................. 64
      7.2.1 The AbstractDispatcher Class................................................................. 65
      7.2.2 FADD Classes .............................................................................................. 66
      7.2.3 SADD Classes .............................................................................................. 67
      7.2.4 The UserDefinedClosure Class ................................................................. 68
      7.2.5 The Dispatcher Class ................................................................................. 69
      7.2.6 Generating Dispatchers with the Maker Module ......................................... 69
   7.3 Automatic Generation of Dispatchers and Functions ......................................... 70
      7.3.1 Function Specification ................................................................................. 70
      7.3.2 Dispatcher and Coercion Specification ..................................................... 74

CHAPTER 8 ................................................................................................................. 78
Original Generation System and Modifications to this System................................. 78
   8.1 The Original Specification Grammar .................................................................. 78
      8.1.1 Lex text section ............................................................................................ 78
      8.1.2 Prefix specification ...................................................................................... 79
      8.1.3 Non-editable Terminals ............................................................................... 79
      8.1.4 Editable terminals ........................................................................................ 80
      8.1.5 Nonterminals ................................................................................................ 81
      8.1.6 Right hand sides ........................................................................................... 82
      8.1.7 Additional scripts ......................................................................................... 82
   8.2 Changes Needed in the Generation System........................................................ 83
      8.2.1 Condensing the output files ......................................................................... 84
      8.2.2 Specifying action routines in right hand sides ............................................. 85
      8.2.3 Accessing right hand side entities ............................................................... 86
      8.2.4 Assigning nonterminals and right hand sides return values ......................... 86
      8.2.5 Specifying Value subclasses ........................................................................ 87
      8.2.6 Describing editable terminals ...................................................................... 88
      8.2.7 Including needed files .................................................................................. 91
      8.2.8 Transporting the new information via the actions files ............................... 91
      8.2.9 Supporting operators .................................................................................... 93
      8.2.10 Hiding right hand sides .............................................................................. 94
      8.2.11 Adding precedence and associativity to terminals and operators .............. 95
      8.2.12 Allowing nullable nonterminals ................................................................. 96
      8.2.13 Comments .................................................................................................. 96
   8.3 Final Generation Process .................................................................................... 96
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1. Example coercion scheme.</td>
<td>22</td>
</tr>
</tbody>
</table>


**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1. An integer-generating context-free grammar.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2. A Lisp program showing dynamic binding.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3. Static binding example in C.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4. High-order function example in Lisp.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 5. Original LEGS class overview.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 6. Example grammar fragment for a polynomial.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 7. Changing the operators in the abstract syntax tree for ‘1+2*3’</td>
<td>38</td>
</tr>
<tr>
<td>Figure 8. Example grammar generating a language for polynomials.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 9. Tree generated from the polynomial grammar in Figure N.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 10. Bad copy and paste on a tree.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 11. UnopDispatcher constructor</td>
<td>66</td>
</tr>
<tr>
<td>Figure 12. Unary coercion and operation.</td>
<td>67</td>
</tr>
<tr>
<td>Figure 13. Binary coercion and operation.</td>
<td>67</td>
</tr>
<tr>
<td>Figure 14. The populate signature.</td>
<td>69</td>
</tr>
<tr>
<td>Figure 15. Example equal function.</td>
<td>71</td>
</tr>
<tr>
<td>Figure 16. Conversion specification.</td>
<td>71</td>
</tr>
<tr>
<td>Figure 17. A Value-domain function from the equal function.</td>
<td>71</td>
</tr>
<tr>
<td>Figure 18. Function specification for max and min functions.</td>
<td>72</td>
</tr>
<tr>
<td>Figure 19. Resulting max and min Value-domain functions.</td>
<td>73</td>
</tr>
</tbody>
</table>
Figure 20. Example conversion signature

Figure 21. A coercion .spec file for addition and division

Figure 22. Valid type definitions

Figure 23. Coercion specification

Figure 24. A coercion/operation tuple array entry

Figure 25. Lex portion specification

Figure 26. Prefix specification

Figure 27. Non-editable terminal specification

Figure 28. Perl call from shell script to generate a non-editable terminal

Figure 29. Editable terminal specification for an integer

Figure 30. Perl call to in the shell script to generate an editable terminal for an integer

Figure 31. Nonterminal specification for an expression

Figure 32. Perl call in the shell script to generate a nonterminal

Figure 33. Right-hand side specification

Figure 34. Perl call in the shell script to generate a right-hand side

Figure 35. New header Perl scripts

Figure 36. New skeleton header files

Figure 37. New Perl scripts to produce the new classes

Figure 38. Action routine for a print command

Figure 39. Fragment showing a nonterminal return type Value pointer

Figure 40. Editable nonterminal with return type int

Figure 41. Forced Value directives creating intValue and MyPointValue

Figure 42. Editable terminal, INT, action routine

Figure 43. Private declaration in an editable terminal

Figure 44. An asstring declaration for an editable terminal
Figure 45. Setval declaration for an editable terminal ...........................................................90

Figure 46. Include and includedir directives for a header MyPoint.h .................................91

Figure 47. Opgroup with string representation ‘?’ ................................................................93

Figure 48. Perl call to generate an opgroup head CALCBINOP ........................................93

Figure 49. Addition operator added to the opgroup BINOP .............................................94

Figure 50. Perl call to generate an operation CALCPLUS in the opgroup CALCBINOP ....94

Figure 51. A fragment example of hiding a right-hand side ............................................94

Figure 52. Giving the addition operator a higher precedence than the multiplication operator. ...............................................................................................................95

Figure 53. An example empty right-hand side .................................................................96
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

TITLE OF THE WORK, CENTERED, SINGLE SPACED, IN ALL CAPITAL LETTERS, EXACTLY AS ON TITLE PAGE

By

Your Name as on the Title Page

Month and year of graduation
(NOT month the work is submitted)

Chairman: Name
Major Department: Name of Graduate Department

The double spaced text of the abstract begins here. All paragraphs beyond this point should use the Body Text style. Note that the headings above should be typed single spaced as illustrated in this sample. Single space the title if it runs more than one line. Single space between the words Chairman and Major Department.

The margins are set. The text above the abstract uses the Normal style with centered justification. The first line (which is blank) uses the One Inch Spacer style.

The month listed is the month of Commencement, NOT the month the manuscript is submitted. The abstract should be a concise summary of the content and purpose of the dissertation, fully understandable without reference to the text. It should not contain parenthetical or bracketed references. Signature lines are no longer used for master's abstracts.

Use the statement and format above, substituting the correct designation (thesis or dissertation) and the official name of the degree as listed in the Graduate Catalog.
Master's abstracts cannot exceed 2 pages and College of Nursing master's abstracts cannot exceed 150 words. Doctoral abstracts cannot exceed 350 words.
CHAPTER 1
Introduction

This work began under the University Scholars Program at the University of Florida to create algorithms in the Image Algebra for the Adaptive Image Manager (AIM). The Image Algebra is a mathematical notation developed to provide a standard description for image processing and computer vision algorithms. This notation is supported by the Image Algebra C++ (iac++) class library began at the University of Florida in 1992. AIM proposes to radically enhance the Parallel Image Manager (PIM) server developed at University of Florida's Center for Computer Vision and Visualization (UF/CCVV), to map image and signal processing (ISP) algorithms around faulting or failing components in a heterogeneous network of processors. AIM's Status Manager and Multi-Level Debugger implements fault tolerance by routing the algorithm mapping process around defective hardware.

The AIM system is tested using algorithms written in iac++, but has the disadvantage that writers must be fluent in the C++ programming language, as well as others reading the programs. Therefore, there is a need for a language having syntax closer to that of the Image Algebra. This work addresses that need by creating a general programming language and hybrid editor generator called Harry LEGS and using this system to generate a programming -- the Image Algebra Prototyping Language (IAPL).

Harry LEGS also provides an interactive programming environment in which programs may be entered top down or bottom up. This facilitates not only programming by fluent Image Algebraists, but also learning the notation by others who may be
unfamiliar through an easy-to-use interface. This programming environment also
provides rapid prototyping of algorithms, which is very useful for testing under the AIM
project.

Chapter 2 presents background information and a brief review of previous work.
Chapter 3 discusses the original tree node system, and Chapter 4 discusses the original
editor system. Chapters 5, 6, and 7 discuss three new systems: the type, environment,
and dispatch systems respectively. Chapter 8 explains changes made in the generator.
Chapter 9 reviews features of the IAPL. Chapter 10 presents conclusions, and Chapter 11
provides suggested future work.
CHAPTER 2
Background and Review of Previous Work

Definitions are presented in this section followed by background information and a summary of previous work for the main topics in this paper.

2.1 Definitions

The target language refers to the final language generated by Harry LEGS.

The user refers to the user of the target language.

The programmer refers to the user of Harry LEGS.

The specification grammar or grammar refers to the annotated context free grammar describing the syntax and semantics of the target language.

2.2 Editing Paradigms

There are two main paradigms of program editing: text editing and structured editing. A text editor is a tool that facilitates the manipulation of textual entities that are organized into a collection of character, words, and lines [AHW90]. A structured editor – or syntax-directed editor – acknowledges the highly structured syntactic forms of computer languages and uses this awareness to the creation and alternation of programs [KU93].

2.3 Text Editors

Text editors do not ensure syntactically correct programs, because the user is free to manipulate characters in any way [Wel91]. However, such an editor can highlight based on syntax and format the body of the program (possibly by way of indentation) to
aide the user in visualizing the program [SBA92]. This and other aspects give text editors the following advantages:

- They require little training because they are commonplace in computers today.
- Users can quickly input programs because of text editors’ free form input.
- Users prefer direct input of programs

There are disadvantages of text editors, though, and they include the following:

- When the editor does not format the text, programs can become very complex and disorganized.
- The user must have an accurate knowledge of the syntax of programs being entered because this syntax cannot be fully checked by the editor.

Because of these advantages and disadvantages text editors can be useful for experience programmers, but can also be a hinderence for novices. To help these programmers enter syntactically correct programs a new type of editor was developed – the structured editor.

2.4 Structured Editors

Structured editors aim to ensure syntactically correct programs by presenting the user with templates that depict the structure of programming constructs and contain placeholders at positions where user-insertions are allowed [AHW90]. Degano et al. presents three modes in which these editors operate: generative, analytic, and hybrid.

2.4.1 Generative mode

In generative mode the user defines productions by editor commands in a top-down, incremental way. Examples of this mode include EMILY [Han71] and MENTOR [GHK84].
2.4.2 **Analytic mode**

In addition to traditional text editors, analytic editors can incrementally analyze program fragments and detect errors. When using this mode, the user works in a bottom-up manner. Examples of analytic editors are PDE1L [MW80] and CAPS [WD76].

2.4.3 **Hybrid mode**

Hybrid mode editors mix the generative and analytic modes, allowing the user to enter programs in both a top-down and bottom-up manner. Examples of hybrid editors are the PSG system [BS86] and the Cornell Program Synthesizer [TR81] & Generator [Rep84].

2.4.4 **Design Issues of Structured Editors**

Khwaja and Urban [KU93] offer the following design issues of structured editors:

- **Choice of Editing Unit**: The editing unit is the part of a document that can be inserted, deleted, or changed otherwise, and should be simple while satisfying the language requirements.

- **Syntax and Static Semantic Errors**: Errors must be handled in a non-obtrusive fashion. Issues relating to error handling are determining what types of errors are handled by the editor, because the type of error handled by the language should be consistent with the errors checked by the underlying language. At the same time structured editors should check for errors, they should also allow for intentional errors by the user, such as leaving an open space to be filled in later.

- **Internal Structure Design**: All user-level operations should be abstract operations linked to the internal structure.

- **Abstractness Level**: The abstractness held by terminal nodes should be considered.
In addition, Welsh et al. [WBK91] presents the following issues:

- **Structure of the Document:** The user should be able to structure the document to his liking, while keeping good internal structure. This should be done through viewing the document as a tree, sequence of character, or functions. To accomplish this, the editor must allow the user to interact through many ways.

- **View of the Document:** Because there may be multiple structural interpretations of the document, there should be a flexible view corresponding to each interpretation. For example, if the user is entering the program textually in bottom-up manner, the document should contain a character-oriented view. But, if the user views the program in tree form, the view should be structured as a tree.

- **Node Navigation:** The ability to traverse any representation of the program is important. If the view is character-oriented, keywords and token should be navigable, but, if the program is presented as a tree, there should be a mechanism to move both depth-wise and breadth-wise in the tree.

- **Error Handling:** As Khwaja and Urban pointed out, handling errors must be done in a way that ensures the program is entered correctly while not being obtrusive to the user. In addition to what Khwaja and Urban presented, Welsh et al. mentions that errors should not propagate, but they do a mechanism to traverse these errors should be provided.

- **Comments:** Comments are desirable in the maintenance of code. Hood stresses the level of abstractness and suggests that efficient recursive data structures and operations in the underlying language can ease the burden of the user
Lastly, Stelovsky et al. reiterates what Welsh mentioned by saying a structured editor should offer multiple views ranging from pure text to parse tree [SAC87].

Overall, the design issues of structured editors involve the flexibility that a user has to write a program. A flexible editor provides the user with multiple views of the program and ways to manipulate this view. Also, the operations on the view should be at the correct level of abstractness for the user. Because this level of abstractness varies between users, this aspect must also be flexible.

2.4.5 Arguments for Structured Editors

Arefi et al. offer three advantages of structured editors [AHW90]:

- Typing effort is reduced and typing errors are minimized.
- Programs are syntactically correct, so syntax error detection is eliminated
- Editing operations are abstractions of the underlying language.

Stelovsky et al. also comment that the range over which structured editors can represent a program as well as the visualization of the relationships and dependencies of modules and procedure blocks can enhance the mental model of that program [SAC87]. Because the structured editor must detect and prevent errors with one interface, all editing and debugging can be performed in a single interface [Wat82]. Lastly, programs entered through structured editors are syntactically correct, and they aid inexperienced programmers by offering choices of completion [Min92].

Structured editors provide abstractness and flexibility to users that cannot be gained from a normal text editor. This is often useful for novices, but sometimes

---

¹ For simplicity all pronouns with gender are masculine in this paper but are meant to encompass both genders. So, every occurrence of his should be taken as his/her, every occurrence of him should be taken him/her, as so on.
becomes evasive for the experienced programmer. Next, the arguments against these editors are examined.

2.4.6 Arguments against Structured Editors

Minor offers the following disadvantages [Min92]:

- Some constructs require more actions to enter than simple text editors.
- Transforming constructs can be cumbersome. Arefi et al. gives the WHILE-DO statements as an example.
- Immediate syntactically incorrect states may cause many errors.
- Experienced users have little to gain from these editors.
- The internal representation is usually not portable to other editors.
- Text textual representation may not be accurately depicted by the structured view.

2.5 Context-Free Grammars

A context-free grammar describes the hierarchical structure of programming languages constructs and is a tuple contains a set of terminal symbols, nonterminals, productions, and a start symbol [ASU88]. Terminal symbols appear in the actual language generated from a grammar. Nonterminals represent a sequence of tokens. Productions consist of a nonterminal called the left side and sequence of terminals or nonterminals called the right side. The right sides of productions are separated by the ‘|’ symbol. The following example is a context-free grammar that generates the language of integers.

\[
\text{number ::= digit number | } \varepsilon \\
\text{digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
\]
Figure 1. An integer-generating context-free grammar.

The ε represents an empty nonterminal. Throughout this paper, nonterminals will be represented in italic font, terminals will be represented in a plain text font, and left and right hand sides will be separated using the ‘::=’ symbol.

2.6 LR Parsing

This section presents the basics of LR parsing. The underlying parser in Harry LEGS is generated from a modified version of the Bison parser generator that is available under GNU public license. Because the Bison parser is based on the LR parser generator YACC [LMB92] it is advantageous to understand the principles of LR parsing, but a detailed description is found in Compilers: Principles, Techniques, and Tools by Aho et al.

2.6.1 General LR Parsing

A LR parser is left-to-right (L) and constructs a rightmost derivation bottom-up in reverse (R). Aho et al. gives the following four advantages of LR parsers.

- LR parsers can recognize mostly any programming language construct.
- The method is the most general non-backtracking shift-reduce parsing method known, yet is as efficient as any other shift-reduce methods.
- LR parsers can parse any grammar that predictive parsers can.
- LR parsers can detect syntactic errors as soon as possible on a left-to-right input scan.

A general LR parser consists of the input, a stack, and a parsing table. Characters from the input are read one at a time. The stack stores states and grammar symbols.
Each state symbol describes the symbols below it, and the current input symbol and these states index the parsing table to determine the next shift-reduce action.

The parsing table is comprised of a parsing action and a goto function. To decide the next parsing action the LR algorithm compares the state on the top of the stack and the current input. The indexed entry of the parsing table can have one of the following options:

- Shift on the state
- Reduce by a grammar production
- Accept, or
- Error

Initially the initial state is put on the stack and the first input character is read. The above routine is repeated until either the input is accepted or an error is found.

2.6.2 Selective LR Parsing

Selective parsing is the act of starting a parse at any given production of a grammar instead of the start symbol and allows one to syntactically analyze fragments of text [Mur92]. Slama explains how the original version of LEGS used elaboration tables to push initial and accepting states onto the stack before initiating a parse in the middle of the program [Sla99]. By pushing the states on the stack, LEGS ‘tricks’ the Bison parser into thinking it has reached an accepting state in the program.

2.6.3 Incremental Parsing

Incremental parsing parses a prefix of a sentential form [Sne90]. A placeholder is placed on the remaining input, and this allows the parsing of parse tree with unelaborated terminal nodes. For example, the input ‘a –’ is a prefix of the sentence ident ‘-‘ ident, where ident is a nonterminal representing alphabetic words. So, this sentence would be
accepted with a placeholder positioned after the ‘-‘. Incremental parsing was accomplished in the original version of LEGS by representing nonterminals of the target language as terminals in the Bison parser. Consequently, in performing a top-down construction of the program, nonterminals were recognized just as terminals, so no errors were thrown, and the structure of the program was maintained [Sla99].

2.7 Evaluation Environments and Type Systems

A type system describes the types that can be used to annotate programs and the relationship between programs and types: common program types include Booleans, arrays, objects, and procedures [Car96]. A major purpose of type system is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up [CW85]. There is much research now on type systems, and how different approaches affect program security, efficiency, and correctness. Factors considered in developing the type system for Harry LEGS included the typing strength, type determination, and support for polymorphism. In the following section prevalent topics in type systems are discussed.

2.7.1 Strong Typing Versus Weak Typing

Languages in which every expression is type consistent are strongly typed [CW85]. Examples of such languages are Java, ML, and Haskell. Weakly typed languages allow expressions to have ambiguous types. Examples of these languages are early C and Lisp. Strongly typed languages provide more security from a language because it can be determined whether and when a type clash occurs. Weakly typed languages, on the other hand, offer a more relaxed programming environment, without the burden of ensuring type consistency.
2.7.2 Static Typing Versus Dynamic Typing

Programming languages in which the type of every expression can be determined by static program analyses are *statically typed* [CW85]. Examples of such languages are Java and ML. On the other hand, *dynamically typed* languages delay type assurance to runtime. An example dynamically typed language is Lisp. Static typing has the advantage of assuring that no errors will occur due to type clashes before compilation and runtime, where dynamic typing allows for generic functions by relaxing type constraints.

2.7.3 Monomorphic Versus Polymorphic Types

Languages, such as Pascal, that are based on the idea that every function, procedure, and operand has a unique type is said to be *monomorphic*. Conversely, *polymorphic* languages, such as C++ and SmallTalk, support types whose operations are applicable to more than one type [CW85]. Cardelli and Wegner further divide polymorphism into the following two categories: *universal* and *ad-hoc*.

2.7.3.1 Universal and Ad-hoc polymorphism

Universal polymorphic functions work on an infinitely many number of types. Two types of universal polymorphism are *parametric* and *inclusion* [Str67]. Parametric functions have an implicit or explicit type parameter that determines the type of argument for each function application, where as inclusion polymorphism allows objects to belong to many classes of types.

Ad-hoc functions work on a number of unrelated types. Like universal polymorphism, ad-hoc polymorphism can be subdivided further into *overloading* and *coercion*. In overloading, a name is used in different contexts to denotes different entities, and in coercion a semantic operation is needed between different types to ensure type compatibility.
2.7.3.2 Four ways of extending a monomorphic language

Cardelli and Wegner give four ways of extending a monomorphic language into a polymorphic one.

- **Overloading**: Defining functions for every combination of operands.
- **Coercion**: Converting the type of operands in order to apply a function defined on other types. This can be done at compile time or at run time, as in Lisp.
- **Subtyping**: Every type in a subtype can be used in a supertype context. This is a special case of inclusion polymorphism.
- **Value sharing**: This occurs when multiple types can take on a common value. For example, the value `nil` in Pascal may be shared between many types. This is a special case of parametric polymorphism.

2.7.4 Static Binding Versus Dynamic Binding

*Static binding* makes an association between two entities before runtime, while *dynamic binding* waits until runtime. Usually, early binding times are associated with great flexibility and later binding times are associated with greater efficiency [Sco00].

The effects of the two are best seen in an example. Consider the Lisp program showing dynamic binding followed by a C program showing static binding:

```lisp
(setf a 3)
(defun e () (print a))
(defun f () (setf a 1) (e))
(defun g () (setf a '(hello (2,3), #'e)) (e))
  (e)
  > 3
(f)
  > 1
(g)
  >(HELLO (2 (COMMA 3))
    (COMMA (FUNCTION E)))
```
In function `e` the variable `a` is dynamically bound to either the `a` in expression `f` or `g`, depending on the calling environment. The following C fragment gives an example of static binding.

```c
int a = 3;
void e() { printf("%d ", a); }
void f() { int a = 1; e(a); }
void g() { int a = 2; e(2); }
void main() { e(); f(); g(); }
> 3 3 3
```

In this C fragment the `a` in function `e` was bound at compile time to the global `a`, so any call to `e` will refer to this `a`.

2.7.5 Treatment of Functions

Typically in programming languages a value has first-class status if it can be passed as a parameter, returned from a function, or assigned into a variable. A second-class value can be passed as a parameter, but not returned from a function or assigned into a variable. A third-class value cannot be passed as a parameter [Sco00]. Common r-values such as numbers and strings typically are first-class values, but functions have varying status in different languages. Typically function languages treat functions as high-order values, while imperative languages view them as low-order values. In Lisp functions (or S-expressions) are treated as first class values. Take the following code fragment as an example.
In the above Lisp fragment \( f \) is (1) assigned the value of a function that (2) takes a function \( \text{fun} \) as a parameter and (3) returns a function that applies the function \( \text{fun} \) to the two primitives 1 and 2. Next \( g \) is assigned the value of the function returned from evaluating \( f \) with the addition function then the multiplication function. Notice that the respective results of 3 and 2 differ.

In C functions may be passed as a parameter and assigned into a variable, but not returned from a procedure. For this discussion, this will be considered second-class.

Lastly, an example of a language that treats functions as third-class entities is Java.

2.7.6 Other Issues in Type Systems

Scott presents three issues in type systems that must be addressed by that system’s implementation -- they are type equivalence, type compatibility, and type inference [Sco00].

2.7.6.1 Type equivalence

Type equivalence can take the form of structural equivalence or name equivalence. Structural equivalence concerns the content of type definitions and says that two types are equivalent if they consist of the same components. Name equivalence
concerns the lexical occurrence of type definitions, so it requires two types to have the same definition.

2.7.6.2 Type compatibility

Type compatibility arises when two or more objects have differing types (i.e. are not equivalent), so the system must decide if the types are compatible with each another. This definition may vary between languages, but, within a strongly typed language, there must be a sound and complete method for determining type compatibility. Different ways of determining compatibility exist, as well as multiple ways to deal with inequivalent, compatible types. These methods will be examined in following sections.

2.7.6.3 Type inference

Type inference involves determining what type an object is as well as determining the type of an operation of one or more objects. Determining the type of a single object varies between languages. In Java every object has a declared type, and because every operation must have a declared return type, the type of an operation is determined by the final return value of the overall operation [GJS96]. However, in ML, objects (variables and functions) are not required to declare a type, so the ML system has to infer an expression’s type based on the operands and operator. If there is not enough information to determine this when declaring a function, a parameter or return value from a function is given the generic type `a, `b, `c, and so on [Rep96].

2.8 Run Time Function Dispatch

Function (or method) dispatch is the process that selects a function to execute based on run-time types [DAS98]. Programming languages, and object-oriented languages in particular, can be separated into single receiver and multiple dispatch
languages [HSP98a]. Single-receiver languages base function dispatch on the dynamic
type of one argument of a function – called the receiver – and the function name.
Examples of such languages are Java [GJS96] and C++ [ES92]. Multiple dispatch
languages base function dispatch on one or more arguments and the function name by
way of late binding [AGS94]. Examples of such languages are Cecil [Cha92] and the
Common Lisp Object System (CLOS) [BDG88]. Holst et al. divides multi-method
dispatch into two major categories: table-based and cache-based [HSP98b].

2.8.1 Table-Based Dispatch

Table-based dispatch involves holding a statically determined table of functions
for every call site. A call site is defined as the combination of the function name and its
arguments (including the receiver). This technique can be done in constant time as
opposed to non-constant time required of cache-based techniques [AGS94]. Two
techniques for efficient implementation of table-based techniques are presented.

2.8.1.1 Compressed n-dimensional tables

Amiel et al. present a method to compress tables by grouping identical dimension
lines into a single line and eliminating null values [AGS94]. Once the table is
compressed dispatch must be performed differently because the unique index of each
type cannot be directly used to access entries in compressed tables. The solution is to
map, for every generic function \( m \) and every argument is position \( i \), a type to index in the
\( i \)th dimension of the dispatch table. The methods are organized into single-dimensional
argument arrays, where the \( i \)th argument array of \( m \) holds the positions of every type in
the \( i \)th dimension of the compressed dispatch table.
This technique is useful when lines of types can be eliminated, and this occurs when one or more types call the same method or have null entries. When this does not occur this technique cannot be used.

2.8.1.2 Multiple row displacement

Holst et al. suggest a constant-time dispatch technique that extends single-receiver tables to multi-method tables [HSP98a]. In single-receiver table dispatch, the method address is found statically and stored in a selector table [HSP98b]. This technique uses row displacement to reduce the number of empty entries in the selector table by compressing the two-dimensional table into a one-dimensional array. To accomplish this each row in the two-dimensional table is shifted until only one entry per column is full, then each shift amount is stored in an array. At run time, the type is used to find the shift index and dispatch the correct method.

Multiple row displacement first compresses the tables by combining call sites with the same methods. It then uses the techniques from single-receiver tables to further compress these tables. The limitation of this technique is that not every table can be shifted so there exists only one full method entry per column. When this criterion is not met neither single-receiver nor multiple dispatch displacement may be used.

2.8.2 Cache-Based Techniques

Cache-based techniques look in global or local caches to determine if the method for a particular call site has been resolved. One cache-based implementation is based on multiple-row displacement [HSL98]. In single-receiver languages using cache-miss algorithms each type maintains a method dictionary mapping names to methods, but this technique does not transfer to multi-methods well. Instead of dividing methods into equivalence classes based on the types where they are defined, as is done in standard
cache-miss algorithms, this technique divides methods based on the name and arity – grouping the methods into behaviors. This technique turns out to be non-constant time, but still better than the common cache-miss algorithm.

2.8.3 Handling Cross-Type Operations

Even with an efficient method for storing and accessing multiply dispatched methods, there is an improvement that can be made to the amount of methods needed. There are two major categories for handling cross-type operations – one involves creating a method for every operation and combination of operands called pure multi-methods, and the other uses coercion to reduce the number of methods needed. In real life these two techniques often must be combined into a hybrid version.

2.8.3.1 Multi-methods

This approach could be used for a small number of types, but as this number grows, even the compression techniques presented above cannot create a manageably sized table. This technique is $O(n^k)$ for each operation for $n$ types and $k$ arguments per operation. So, given $m$ operations this technique is $O(m \cdot n^k)$. By reducing the number of methods needed, the table size can be greatly reduced.

2.8.3.2 Coercion

Often different data types are not completely independent, and there exist ways to change objects of one type into object of a similar type – this is called coercion [ASS95]. This technique can be used in multiple dispatch by coercing similarly typed objects into a common type and creating a method for each operation only over that common type instead of creating $n^k$ methods per operation for $n$ types with $k$ arguments per operation. For example, consider binary addition and binary subtraction over the integers, reals, and
complex numbers. Given there are three types and two operations; the conventional table-based approach would produce 18 methods – one for each operation and combination of two operands. Using a coercion scheme the number of methods can be reduced from 18 to four – two coercions and two operations. The two coercions transform integers and reals into complex numbers, and the two operations add and subtract complex numbers.

This example has introduced an aspect of type sets that is useful in coercion – tower hierarchies.

2.8.3.2.1 Tower hierarchies

Tower hierarchies exist when the coercion within a set of types is fully contained within that set (called a closed type set), and there always exists a coercion that coerces two types within the set into a single type [ASS85]. When these structures exist, introducing a function raise can redesign the coercion scheme. The function raise applies successive coercions to a subset of a type set until all objects have the same type. A raise function, together with on operation, can cover every possible operation on a closed type set.

2.8.3.2.2 Limitations of tower hierarchies

The limitation of tower hierarchies occurs when there do not exist close type subsets over which a raise function is valid.

2.8.3.3 Multi-method/coercion hybrids

Often in real life one cannot group types into homogenous sets or it is not clear how types should coerce, but the advantages of coercion may still be used by combining multi-methods will coercion. This can be done by using coercion in every instance that
the type set permits and using multi-methods in other situations. As an example, consider the addition operation over the set of types containing real numbers, complex numbers, and alphabetic strings. Any add operation involving a string should act semantically as an append operation, thus coercing any non-alphabetic operand into a string. But those operations involving just numbers should arithmetically add the two operands. This leads to the creation of two add functions – one between two strings and one between two numbers. Additionally, not all coercions converge to one type, because is some instances real numbers coerce into strings and in other instances they coerce into complex numbers. Table 1 displays the coercion scheme for this example as tuples containing two coercions and one operation, where ↑C, ↑S, and nil denote coercion to Complex, coercion to String, and no coercion, respectively, and addC and addS refer to the add function for type Complex and String, respectively.

<table>
<thead>
<tr>
<th>Type</th>
<th>Real</th>
<th>Complex</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>↑C, ↑C, addC</td>
<td>↑C, nil, addC</td>
<td>↑S, nil, addS</td>
</tr>
<tr>
<td>Complex</td>
<td>nil, ↑C, addC</td>
<td>nil, nil, addC</td>
<td>↑S, nil, addS</td>
</tr>
<tr>
<td>String</td>
<td>nil, ↑S, addS</td>
<td>nil, ↑S, addS</td>
<td>nil, nil, addS</td>
</tr>
</tbody>
</table>
CHAPTER 3  
The Original Parse Tree Library and Modifications Made to this Library  

3.1 Overview of the Original Parse Tree Library  

This section introduces the original parse tree library as implemented by Skander Slama [Sla99]. This library consists of C++ classes representing parse tree nodes. There are two groups of tree node classes – base classes and subclasses of these base classes. The base classes represent the general elements of a parse tree, and the derived classes extend these classes to implement language specific features needed by the target language.  

3.1.1 Parse Tree Node Base Classes  

The following is a brief description of the original base classes. A more detailed description is found in Skander Slama’s Master’s Thesis [Sla99]. The following figure gives an object overview of the system.  

![Object Overview](image)

Figure 5. Original LEGS class overview.
3.1.1 The `treeNode` class

This abstract base class is extended by all parse tree nodes and provides functions to derived classes for accessing a node’s parent. Additionally virtual functions are declared for accepting a `nodeVisitor` and allowing a `nodeVisitor` to visit a node’s children. These methods should be overloaded according to the constraints of a node.

3.1.1.2 The `viewableNode` class

This class is extended by `nonTerminalNode` and `terminalNode`, and represents those nodes in the parse tree that should be viewable to the programmer.

3.1.1.3 The `nonTerminalNode` class

This class is derived from `viewableNode` and represents nonterminal tree nodes. It contains an `RHSNode` (representing a right hand side in the tree), a list of possible right hand sides that could elaborate this node, and a method to return initial and accepting states for use in selective parsing. It was intended that every nonterminal in the grammar specification would produce a class derived from `nonTerminalNode`.

3.1.1.4 The `terminalNode` class

This class is derived from `viewableNode` and represents terminal tree nodes. Instances of `terminalNode` add functionality to `viewableNode` by only identifying themselves as being terminal. It was intended that every terminal symbol in the grammar specification would produce a class derived from `terminalNode`.

3.1.1.5 The `RHSNode` class

This class is derived from `viewableNode` and represents the right hand sides of nonterminal nodes. Instances of this class contain child objects derived from `nonTerminalNode` and `terminalNode` and an interface to access these children. It was
intended that every sentential form in the grammar would produce a class derived from RHSNode.

3.1.1.6 The `viewableNodeList` class

This class implements a fixed size list of `viewableNodes`. It is used when iterating through a node’s children and constructing the tree.

3.1.1.7 The `RHSNodeList` class

This class is derived from `viewableNodeList` and may only contain instances of RHSNode. Instances of `nonTerminalNode` will return an `RHSNodeList` containing all possible completions for that node, and this mechanism is used for presenting the user with a menu of possible completions for a nonterminal node.

3.1.1.8 The `nodeVisitor` class

Instances of this class were intended to travel through the tree to perform a particular function, such as constraining or execution. Every `treeNode` should implement a method that accepts a `nodeVisitor` and performs the correct function.

3.1.2 Generated Parse Tree Node Classes

These classes are generated according to the specification grammar. All generated class begin with a common prefix, and this prefix will be denoted `<PREFIX>`.

3.1.2.1 The `<PREFIX>NonTerminalNode` classes

These classes derive from `nonTerminalNode` and implement a unique string representation of the form “<name>” – where name is the name of the nonterminal given in the specification grammar. Also, instances return a unique instance of `RHSNodeList` containing `RHSNode` objects that represent possible completions for this node.
3.1.2.2 The `<PREFIX>TerminalNode` classes

These classes derive from `terminalNode` and implement a unique string representation that is specified in the specification grammar.

3.1.2.3 The `<PREFIX>RHSNode` classes

These classes, derived from `RHSNode`, are created for every sentential form in the specification grammar and contain instances of `<PREFIX>NonTerminalNode` and `<PREFIX>TerminalNode` as children according to the grammar. For each `<PREFIX>NonTerminalNode` $N$ in the grammar with $n$ right hand sides, the following subclasses of `RHSNode` were created: `<PREFIX>N_RHS0`, `<PREFIX>N_RHS1`, ..., `<PREFIX>N_RHSn`.

3.1.2.4 The `<PREFIX>NodeVisitor` classes

Only one class derived from `nodeVisitor` was created, and this class contains member functions for each `<PREFIX>NonTerminalNode`, `<PREFIX>TerminalNode`, and `<PREFIX>RHSNode` created each named `visit name` where `name` was the name of the class. This method was intended to implement semantic checking, code generation, or another node specific operation.

3.2 Modifications Needed in the Parse Tree Node System

This section details the changes needed in the original parse tree system and the design issues that affected these changes. The base parse tree nodes appear in a static library `libbasefiles.a`. This library is used to extend the base classes to create new classes defined by the specification grammar.
3.2.1 Methods of Evaluating Tree Nodes and the Chosen Method

The original system did not include a method to evaluate a tree once it was constructed, and this feature was definitely needed in order to produce a functioning programming language. There were three possibilities that were considered, and these are now presented.

3.2.1.1 Visitors for evaluation

One possibility to evaluate a parse tree was the original design that passed an instance of nodeVisitor down the tree, and this instance would perform some semantic function on each node. The positive aspect of this technique was that all the semantic properties of the language would be centralized in the methods of the nodeVisitor class.

There were two disadvantages, though. First, the current visitor implementation required knowing the exact class of a viewableNode instance in order to visit it because the subclass of nodeVisitor generated had a method for each generated subclass of nonTerminalNode, terminalNode, and RHSNode named visit name, where name is the class name. Two workarounds were proposed for this problem. The first involved polling the type of a node before calling it via an if-then-else or switch-case construct, but this technique was both inefficient and inelegant. The second workaround was to keep an array of function pointers $A_T$ for each group of RHSNodes that formed completions for a particular nonTerminalNode $T$, and each of these classes would have a unique index from 0 to $n-1$, where $n$ was the number of RHSNodes for that nonTerminalNode $T$. At run time the correct visit method would be the function found by indexing the array $A_T$ by the index returned from the current RHSNode. This technique was more time efficient than the previous, but required more space and was cumbersome.
The second disadvantage of using visitors for evaluation was that it violated the notion of object-oriented programming set forth by Skander and Buckner. A true object-oriented approach would be to encapsulate the unique functionality of each node within that particular node. With this in mind other possibilities were explored.

3.2.1.2 Templates for evaluation

The second possible evaluation scheme was to define the `treeNode` class as a template class, and subclasses would define concrete classes based on this template\(^2\). The problem with this approach was that templates cannot contain pure virtual functions in ANSI C++, and many `treeNode` subclass methods were best implemented as virtual functions.

3.2.1.3 Evaluation method inside each subclass

The last possibility was to have `viewableNode` classes declare a pure virtual `evaluate` method that each subclass was required to define. This `evaluate` method would take an instance of the class `Env` and return an instance of the class `Value`\(^3\). This technique upheld the notion of object-oriented programming as well as allowing `treeNode` to declare pure virtual functions.

3.2.1.4 The chosen evaluation technique

Because the third method was the most object-oriented and encapsulated the unique action of the tree nodes within the nodes, this was the chosen technique. The new evaluation method now introduced a return value to each `viewableNode`, hence these nodes now had a value attribute. However, as it will be shown later, this evaluation

---

\(^2\)A template in C++ is a generic class with place holders for specific class information. Readers unfamiliar with C++ templates should refer to the Annotated C++ Reference Manual [ES92].

\(^3\)The `Value` and `Env` classes are introduced in a later section.
cannot be declared in the treeNode or viewableNode classes, but must be declared in
derived classes.

3.2.2 RHSNode Wrapper Classes

   The original system generated all <PREFIX>RHSNode classes as direct subclasses
of RHSNode. This implementation would not work now because all classes extending
viewableNode implemented a unique evaluation method that returned (in general) a
different type. For this reason, a set of wrapper classes was created so that, in general,
RHSNode classes could return different types, but all RHSNode classes associated with a
particular nonTerminalNode-derived class were required to return the same type.

3.2.3 The HarryLegsException Class

   A general exception mechanism was needed to deal with errors in the system, and
the base class for these errors was the HarryLegsException class.

3.2.4 The operatorNode Class

   The original implementation had two types of terminal nodes: editable and
noneditable terminals. The former was used for variables and functions and the latter for
language tokens and operators. Common productions in programming languages are
binary and unary operations, and in the original system modifying a binary operation
required rewriting the entire production, thus destroying the operands. A more attractive
solution is to edit just the operator without consuming any other nodes. For this reason a
new subclass of terminalNode was needed—operatorNode—that offered limited editing
capability.

   Operators in a language also often appear in groups. For example, one associates
the arithmetic operators +, -, *, and /, the bit wise operators <<, >>, ||, and &&, and the
relational operators <, <=, >, >=, ==, != into three groups. Likewise, the subclasses of
operatorNode appear in groups, and these groups are called opgroups. Each opgroup has a unique string representation, and any of the operators in one group may be rewritten as another operator in the same group. It was the intention that in the editor only the head of the opgroup would be presented in nonterminal menus, and this head operator could be elaborated by clicking on it on the canvas. This would clean up the editing unit by presenting short, more manageable menus to the user.

3.2.5 The emptyNonTerminalNode class

The emptyNonTerminalNode encapsulates nullable productions in the specification grammar. Because nullable productions are commonplace in context-free grammars, it was deemed imperative to allow them in the specification grammar. Without an explicit class, nullable productions could be incorporated very quickly into the specification grammar and implemented by simply not creating a node when a nullable production was encountered. Therefore, these nodes were not evaluated and would not be visible in the editor. This posed a problem with the original implementation, so it was necessary to create a new subclass of nonTerminalNode to hold a place for nullable productions. Furthermore, new subclasses of emptyNonTerminalNode were generated from the specification grammar.

3.3 Implementation of Modifications to the Parse Tree Node System

Modifications to the original parse tree node system involved creating new classes for the base parse tree node library as well as modifying the classes generated from the specification grammar⁴.

⁴ The generation system is discussed later in this paper.
3.3.1 Method of Evaluating Tree Nodes

The scheme to evaluate the parse tree was to define a unique evaluation routine within each sentential form of the specification grammar, and this method has the form

\[ \text{return}\_\text{type} \ \text{evaluate}(\text{Env} *e); \]

where \text{return}\_\text{type} denotes the return type of that class, and the \text{Env} class encapsulates an evaluation environment and is discussed later in the paper. Every \text{viewableNode} would contain a current \text{Env} instance and the first action of any \text{evaluate} method would set the current \text{Env} instance to the passed in argument. Because the return type for each class derived from \text{viewableNode} could be different, the declaration of \text{evaluate} could not appear in the definition of \text{treeNode}, \text{viewableNode} or even \text{nonTerminalNode}. Hence the declaration had to appear in the definition of the subclasses of \text{nonTerminalNode} (\text{terminalNode} are not evaluated so all can have return type void). However, the return type of \text{nonTerminalNode} must match the return type of all of its member \text{RHSNode}-derived instances, so the evaluation method for subclasses of \text{nonTerminalNode} was just to return the evaluation of its right hand side. The code for this method follows.

\[ \text{return}\_\text{type} \ \text{evaluate}(\text{Env} *e) \ { \text{return}\ get\text{RHS}() ->\text{evaluate}(e); } \]

An \text{evaluate} method with no arguments is provided, also, that allocates a new environment and passes this environment to above evaluate method. Additionally, the class \text{nonTerminalNode} defines a pure abstract method \text{Value *execute(Env *)}, that is defined in all generated classes to return the result of its evaluate method. This was needed so that the \text{Closure} class\textsuperscript{5} could be able to hold a lambda expression in the target language and execute a portion of the parse tree. The return type need not be known statically, so the generic \text{Value} return type suffices. It is left to the programmer’s
discretion to enforce return type declarations in the target language. If done so, the programmer must implement a system to enforce this rule.

This solves the problem of nonTerminalNode-derived class evaluation, but getRHS as defined in nonTerminalNode return an RHSNode, and there is no evaluation method defined in this class. This leads into the next modification of creating RHSNode wrapper classes.

3.3.2 RHSNode Wrapper Classes

Each nonTerminalNode-derived call to getRHS must return an RHSNode-derived instance that has the same return type for its evaluation method as the nonTerminalNode. To ensure this, a wrapper class $W_N$ was created for each nonTerminalNode-derived class $N$, and every possible RHSNode $R_i$ that could complete $N$ had to extend class $W_N$. Every class $W_N$ declared an evaluate method that returned the same type as $N$'s evaluate method, and this ensured that the call

\[
\text{return getRHS() \rightarrow evaluate(e);}
\]

could be placed inside the nonTerminalNode-derived evaluate method.

3.3.3 The HarryLegsException Class

This class was part of the base node library and allowed errors to be reported with the file name, line number, and error message string. A number of subclasses were defined to deal with specific errors, such as parse errors, invalid input errors, and evaluation errors.

3.3.4 The operatorNode Class

This class was part of the base node library and extended terminalNode. In adding this class a new virtual method isTerminal in viewableNode was declared, and

footnote{The Closure class is discussed in Chapter 5.}
it returned 0 in every class except operatorNode and its subclasses. Treated outside the context of the generated files the operatorNode class was very similar to the terminalNode class, but operators are editable, so the isEditable method was defined to return 1. On the other hand, the generated operatorNode subclasses differed greatly from regular terminalNode subclasses. Classes derived from terminalNode did not contain RHSNode instances, so they could not be rewritten. Classes derived from operatorNode, however, could be rewritten, but only in a limited fashion. The three groups containing arithmetic, logical, and relational operators were presented previously, and these groups formed opgroups. An opgroup consists of head node with an initial string representation and zero or more operators to which this head node can transform. To implement this, an RHSNode subclass was created for the head node and every opgroup member, and each opgroup member could only be rewritten as one of these RHSNode-derived classes. This ensured that, for example, an arithmetic operator could only be changed into another arithmetic operator, and a logical operator could only be changed into another logical operator.

3.3.5 The emptyNonTerminalNode Class

This class and its subclasses were not part of the base tree library, but were, instead, automatically generated from the specification grammar. An emptyNonTerminal is derived from nonTerminalNode and differs from ‘regular’ subclasses of nonTerminalNode by throwing a HarryLegsException instead of returning a value. These nodes were created from empty sentential forms in the specification grammar.
CHAPTER 4
The Original Editing System and Modifications to this System

The original editing system, HELP, is introduced, but Bucker gives a detailed description in his Master’s thesis [Buc99]. The needed changes in the system are, then, presented, followed by the implementation of these changes.

4.1 Overview of the Original Editing System

The original HELP editor classes were divided into three categories: document classes, view node classes, and view classes. The document classes manipulated the parse tree itself, the view node classes encapsulated a visible node to the user, and the view classes manipulated these view nodes.

4.1.1 Document Classes

The CDocument class contained the parse tree and was responsible for changes to the tree. Any operation on the tree from the user went through the CDocument class, and this class accessed elements of the tree only through the viewableNode interface and supporting classes.

The CDocumentManager class created, destroyed, altered the focus of, and modified any open documents (instances of class CDocument). This class also managed mouse and keyboard events, because it kept track of the current document in focus.

Lastly, the CDocumentContainer class provided methods to compare and manipulate instances of CDocument. Likewise, every CDocument instance was contained inside an instance of CDocumentContainer, and a CDocumentManager manipulated these containers instead of directly interacting with the CDocuments.
4.1.2 View Node Classes

The view node classes provided an interface to the actual text widget displayed on the canvas. Events handled by this class include highlighting/unhighlighting, keyboard and mouse button presses, and position changes due to reparsing or right hand side completion. The original system included three subclasses of CViewNode.

A CNonTerminalViewNode encapsulated a viewable version of the nonTerminalNode. It could be selected, activate a menu for completion, and have text entered directly into it. When entered directly, text triggered a CNonTerminalViewNode to change into a CUnParsedTextViewNode -- these nodes are discussed below.

A CUnParsedTextViewNode was much like a CNonTerminalNode except it could not activate a menu so could not rewrite into another CViewNode.

A CNonEditableTerminalViewNode was the viewable version of a terminalNode. It could neither activate a menu nor allow editing.

A CEditableTerminalViewNode could not activate a menu and rewrite itself, but could be edited. This node was the viewable version of the editableTerminalNode. Its text color was blue and allowed text manipulation inside of it.

4.1.3 View Classes

The top-level class of this group was CView and controlled the overall display seen by the user. Every CView was attached to a document, and every operation performed on the tree or the program was directed through the attached CDocument instance. The tree could be displayed in normal or parse tree form, depending on the current widget manager.

The class CViewWidgetManager was responsible for displaying the parse tree in an instance of CView. Widget managers completely encapsulated adding and removing
subtrees to and from the view, but the method of displaying the program depended on the subclass.

The CNormalViewWidgetManager class displayed the program as would be seen on a text editor and maintained a list of CViewNodes. These nodes were terminal if the corresponding node in the parse tree could be derived to a terminal symbol and was otherwise a nonterminal node. This view provided the easiest way to view a program, but often the parse tree view offered a better environment to edit and expand the program.

The CParseTreeNodeViewWidgetManager class displayed the program in parse tree form by placing each node on a separate line and propagating tabs for each level of children. The view nodes are maintained in a list, but, unlike the CNormalViewWidgetManager, all nodes were present, regardless of whether they could derive to a terminal node or now. This presented the entire parse tree view and was useful for editing.

Additionally, the following classes enhanced the view and offered the user control over the program:

- CMenuBar
- CScrollBars
- CStatusBar

Class CMenuBar provided a dynamically generated menu that was constructed from the possible completions of a viewableNode found inside a CViewNode on the canvas. The choices for the menu were constructed from the viewableNode method listRHS that lists the possible completions that a viewableNode could be rewritten as. Selection of the menu caused the selected CViewNode on the canvas to be rewritten as the
completion chosen from the menu. Classes CScrollBars and CstatusBar provided scrollbars to the editor window and status bars at the bottom of the window respectively.

4.2 Modifications Needed in the Editing System

Changes in the system stemmed from changes in the parse tree node system as well as improvements discovered while testing the original system. These needed changes are presented below.

4.2.1 Forced Parsing

In the original system the precedence and associativity of terminal symbols were enforced through the grammar rules, but the addition of a new operator node type in the parse tree node library allowed the precedence and associativity to be expressed with directives in the specification grammar rather in the nonterminals. This change brought about the need to reparse parts of the tree whenever an operator was changed into another operator with a different precedence. The following example verifies the need for this change.

Consider the grammar fragment in the following figure.

```plaintext
%left '+'
%right '*'

term : term '+' term 
     | term '*' term 
     | integer 
     ;
```

Figure 6. Example grammar fragment for a polynomial.
The `%left` and `%right` directives denote associativity, the `+` appearing before
the `*` denotes that `+` has a lower precedence than `*`. Now, consider changing the
sentence `1+2*3` to `1*2+3`. The abstract syntax trees for this change are shown in the
next figure.

```
+  *  
1   2 3
=⇒ 1 2 3
```

Figure 7. Changing the operators in the abstract syntax tree for `1+2*3`.

The precedence between the `+` and `*` operators has been lost, so to ensure the
intended precedence of the terminals in the grammar was upheld, there needed to be a
way to reparse the tree after every action that could have upset this precedence. This
mechanism is referred to as force parsing. Likewise nodes that require forced parsing
will be referred to as forced parse nodes.

**4.2.2 Support for the new operatorNode Class**

A viewable version of the class operatorNode was needed. This class had to
implement the following characteristics.

- Encapsulate visual characteristics such as text font and text color.
- Allow a new form of insertion that would allow a new operator to be inserted in
  place of another. Previously tree nodes, could only be inserted below another tree
  node as a child. A forced parse should follow this insertion, also.
• Be capable of initiating a forced parse when inserted into the program.

4.2.3 Improved Cut, Copy, and Paste

The original cut, copy, and paste facilities allowed the user to cut and copy single nodes, but this operation didn’t always travel all the way down the tree. A true cut or copy should begin at the desired node and travel to the leaf nodes of the tree (these could be terminals or non-elaborated nonterminals). Additionally the cut and copy operations should not disrupt the syntactic correctness of the program, so these operations should not be allowed on unparsed text or user-selected text within a node. If the user could cut or copy unparsed text, there would be no assurance that the text was syntactically correct, hence a following paste would invoke a parse error. Likewise, there is no assurance that selected text within a node is syntactically correct, so the selection facility should be disallowed.

Just as cut and copy operations could disrupt the syntactic correctness of a program, the paste operation could do the same. The design decision was made to check the correctness of a cut-paste or copy-paste combination with the paste, and this would be done by gathering the terminal text of a node and all its subtree nodes and invoking the parser to reparse that text into the program. This solution had two advantages: By reusing the generated parse functionality and error-handling modifications to the HELP structure would be minimal. Additionally, the user could handle paste errors in the same manner he handled parser errors, because paste errors were treated as parse errors within he generated parser.

There were still possibilities for parse errors, so, to minimize these, a paste operation would be invoked upon the highest valid node in the branch containing the desired node instead of on that node. This was needed because, even though the terminal
text of a subtree seemed syntactically correct in the normal program view, it could have been inserted under a nonterminal that could not derive that text. The following example motivates the need.

Consider the grammar in the following figure.

```
term  :  term  ' + ' factor  
    |  factor  

factor  :  integer  ' * ' factor  
    |  integer  
```

Figure 8. Example grammar generating a language for polynomials.

A parse tree constructed for the string ‘1+2’ is shown in Figure 9.

```
    term
      /   \
  term   factor
  /     \
factor   integer
 /     \
integer 2
  \
B 1
```

Figure 9. Tree generated from the polynomial grammar in Figure 8.

The user could copy node A and try to paste on node B. In the normal this is syntactically correct, but as Figure 10 shows, the resultant tree is not found in the grammar specified in Figure 8.
There needs to be a way to safely cut, copy, and paste.

4.2.4 Better Error Handling

The original system handle errors by presenting an error message in the lower right hand corner of the display, and, while the method, met Buckner’s criterion of non-obtrusive error handling [Buc99], it carried this notion too far in the author’s opinion. Parse errors should be presented to the user immediately with the opportunity to correct these errors. The system, therefore, needed a mechanism to present the user with an error message, containing the incorrect text that allowed him to modify this incorrect text.

Additional errors that needed reporting included attempted evaluation of non-elaborated nonterminals\(^\text{6}\) and target language errors.

---

\(^6\) This paper discusses the possible ways to handle non-elaborated nonterminals in the implementation section
4.2.5 Unification of HELP and LEGS

The original intent was to make to systems that were independent [Buc99, Sla99], yet solved a common problem, but this separation was not needed anymore. In fact, this detachment hindered target language development, because it forbade target language-specific error handling through the HELP error handling system and other aspects of the target language-specific that greatly improved the final product.

4.3 Implementation of Modifications to the Editing System

Modifications to the original system took the form of additional classes as well as inline modifications to the original code.

4.3.1 Forced Parsing

As, a previous example illustrated forced parsing should occur when an operator of one precedence in replaced by an operator of a different precedence. To implement this the user needs to identify the nonterminals that contain operator with differing precedence in the specification grammar\(^7\). All \texttt{RHSNode}-derived classes of these identified forced parse nonterminals define the method \texttt{isForcedParse} to return 1, instead of the default 0. The HELP classes need this information, so these classes can decide where to start a forced parse.

A forced parse can be invoked from the menu or implicitly from the system – either way, the method \texttt{MenuActionForceParse} is called. This method will travel up the tree to find the highest forced parse node (node whose class returns 1 from the method \texttt{isForcedParse}), find the position of the node with current focus, collect all the terminal text and reparse it, insert the resulting new tree node, and return the focus to the original

\(^7\) The facility to identify these nonterminals is discussed with modifications to the generation system.
node on which the parse was forced. The next section discusses each step of forcing a parse on a node \( N \) in more detail.

4.3.1.1 Finding the highest forced parse node

The first step in forcing a parse is to find the highest nonterminal in the tree that could be affected by the parse. Starting with node \( N \), the tree is traveled upward while saving the most recent nonterminal with a forced parse right hand side as node \( S \). When this traversal reaches the top of the tree node \( S \) holds the highest node needing a forced parse.

4.3.1.2 Finding the position of forced parse node \( N \)

After the terminal text of a subtree is reparsed and a new subtree is inserted all information from the former subtree is lost, including the identifying information of the node that had the current focus before the parse. In order to return the focus to the node that originally had the focus, that node’s position in the tree must be identified. This position \( pos \) is obtained by counting all the terminal nodes in a pre-order traversal of the subtree with tree head \( S \) (i.e. the forced parse subtree). This is a valid method because the order in which these nodes appear in a pre-order traversal will not change following a parse, only the level in which they appear may change. A previous example in section 4.2.1 showed this property.

This algorithm was implemented very simply because the normal view managers already contain a list of all the viewable terminal nodes, so counting from node \( S \) to node \( N \) involved searching linearly for the node \( S \) and counting to node \( N \). This was done in
O(n) time for n nodes, but could be improved by threading the viewable tree with the parse tree nodes so that a viewable node could be retrieved in O(1) time\(^8\).

4.3.1.3 Collect the terminal text of forced parse node \(N\) and reparse it

The terminal text was collected by beginning with an empty \texttt{std::string}\(^9\) named \texttt{str}, performing an pre-order traversal of the subtree of node \(S\), and appending the text of any terminal nodes encountered to string \texttt{str}. The resulting string \texttt{str} then contained the text that would be passed to the generated parser to create a new tree node to be inserted into the program. These actions were done in the first part of the method \texttt{ParseUnParsedTextOfParseTreeNode} found in the \texttt{CDocument} class.

4.3.1.4 Inserting the resulting tree node from the parsed string \texttt{str}

After the string \texttt{str} was obtained it was passed to the generated parser in the second half of the method \texttt{ParseUnParsedTextOfParseTreeNode} by calling the method \texttt{PartialParse}. This method was implemented by the generated parser and returns a new \texttt{RHSNode}-derived node to be inserted into the tree. If an error occurs the appropriate action is taken by the error handling system discussed later in this chapter. If an error does not occur the \texttt{RHSNode}-derived node is inserted into node \(S\).

4.3.1.5 Return focus to the node \(N\)

Now that the newly parsed subtree was inserted into node \(S\), another pre-order traversal is performed counting from 0 to \texttt{pos}, the position of node \(N\) relative to node \(S\), and then focus is given to the node with position \texttt{pos}. This node is node \(N\), and that concludes the forced parse.

\(^8\) The Future Work section elaborates on the threading idea.
\(^9\) The class \texttt{std::string} is found in Silicon Graphics Standard Template Library.
4.3.2 The `COperatorNode` Class

Class `operatorNode` differs slightly from class `nonTerminalNode` in the base parse tree node library, but this class’ corresponding viewable class `COperatorNode`, has many differences from viewable class of `nonTerminalNode`, `CNonTerminalNode`. The design and implementation of this class is now presented.

4.3.2.1 Visual aspects of the class `COperatorNode`

An operator should appear differently than other terminal and nonterminal nodes in the program, and, for this reason, a distinct color was chosen for operator nodes in the program to distinguish them from other nodes. Additionally, operators in target languages (such as the one implemented for the Image Algebra) often have special characters, so a facility was put in place so that the programmer can specify a special font to implement these characters. This font has the name `PopupMenuFont` and can be found in the Tcl\Tk start-up script `iagui.tcl`.

4.3.2.2 Facilitating a new form of insertion

Previously tree nodes were inserted into the tree as children, but did not replace other tree nodes. In order to achieve the desired behavior of replacing operators with operators found in the same opgroup, there needed to be a way to replace a tree node in addition to giving that node a child. This was accomplished by creating a method in the `CDocument` class called `insertText` that inserted a string into the node with current focus. This method first queried for the currently focused `CViewNode` instance, and then replaced the text of the underlying `viewableNode` via the `CViewNode` method `UpdateNodeAndWidgetText`. The `insertText` method was later used in various components of the system and proved to be very useful.
4.3.2.3 Support for forced parsing

A forced parse only occurs when it is initiated from the menu or when a new operator is inserted; hence a forced parse should follow every operator insertion. This was implemented in the CPopupMenu class by modifying the method PerformPopupMenuChoice that originally took in a menu list and prepared the next view node to be inserted into the parse tree. This change queried the selected view node, and instead of simply inserting this view node into the tree, if the node was an operator, it first inserted the new operator via the insertText method, and then called a forced parse via the MenuActionForceParse method.

4.3.3 Improved Cut, Copy, and Paste

To ensure that a cut or copy operation on a node traveled to all of its children, the makeNew method of viewableNode was altered so that it not only returned a new copy of itself, but also recursive copies of its children. To ensure that a cut or copy couldn’t occur on user-selected text the selection was just ignored when performing the cut or copy.

To paste node A on node B (from the previous cut-copy-paste example) the original implementation of paste would simply set node B’s right hand side to the current right hand side of A. This obviously has drawbacks, because it doesn’t take any steps to uphold syntactic correctness. The solution to this problem was to first retrieve the terminal text of Node A into a std::string str. As the example in section 4.2.3 illustrates it is necessary to travel to the highest possible node in the tree to insert this text, and this was accomplished by traversing up the tree starting from node B using a traversal node T until the parent of T had more than one child. This technique ensured
that the text \texttt{str} would be placed at the highest nonterminal that could derive the nonterminal node \textit{B}.

This reduces parse errors greatly, but there still exists the possibility to have a parse error. Because the text inserting was carried out by the same mechanism to insert user-entered text, the same error handling routine would handling any paste errors, and the user would have a chance to correct his error.

4.3.4 Better Error Handling

One error handling mechanism has been mentioned previously, and that is the class \texttt{HarryLegsException}. Any instance of \texttt{HarryLegsException} will be caught at runtime and the error message will be printed to the screen, but this procedure could be improved. To improve the error handling and reporting, a class \texttt{ErrorMsg} was created that encapsulated an error catcher/reporter. This class featured a number of Tcl\Tk message boxes for error reporting, including a parse error box that gave the user a chance to correct his incorrectly input text. The figure below shows a parse error widget.

![Figure 11. A bad parse widget.](image)

Other events that triggered errors were trying to evaluate unelaborated nodes, target-language independent errors thrown by the programmer, and errors within the evaluation environment.
4.3.5 Unification of HELP and LEGS

To homogenize the two systems a common file `helpLink.h` was created that contained LEGS information needed by HELP, and this file included the name of the start symbol, the name of the grammar parser, and extra include files. Additionally, the error handling system of HELP was available for use by the target language.
CHAPTER 5
Type System

This chapter introduces the goals, design, and implementation of the type system provided to the programmer. The programmer is not required to use this system, but its use is supported by the entire Harry LEGS system.

5.1 Goals and Overall Structure of the Type System

The main goal of the type system was to provide a flexible, easy-to-use system with very few limitations so the programmer could coordinate with the environment system\(^\text{10}\) and customize this system to suit the needs of the target language. These flexibilities include, but are not limited to, choices between strong/weak and static/dynamic typing. To accomplish this the ultimate system was polymorphic with one layer of inheritance. There was an abstract class, `Value`, from which all type were derived, but any further inheritance could be implemented by the programmer by interpreting objects in the target language differently. For example, in the target language for the Image Algebra, the type system was one-level polymorphic, but appeared polymorphic through the use of coercion\(^\text{11}\).

This system is dynamically non-extensible – all possible types are determined at the time the specification grammar is processed. The decision to disallow new types was made to simplify the system, but the programmer could implement extensions allowing new types to be introduced. Also, type system implementations should provide a way to

\(^{10}\) The environment system is discussed in the next chapter.

\(^{11}\) This coercion scheme is described in a later chapter.
uniquely identify the type of an object at run time, and this was implemented by statically assigning each type an integer type identifier at that generation of the types.

Each generated subclass of `Value` contains an instance of a C++ type or primitive `val`, and this `val` is accessible through an interface described below. Additionally, utility classes were provided that allow lambda expressions, parameters to be passed, and arrays – these classes are also introduced in the following section.

5.2 Implementation of the Type System

This system is linked to Harry LEGS as a static library written in C++. The following describes the classes composing the system.

5.2.1 The Superclass `Value` and Generated Subclasses

The abstract class `Value` is the class from which all types are derived. It defines utility methods for use by subclasses, declares methods to access the underlying value of the subclass and global constants to dynamically identify subclass instances.

Values can be linked in two ways – as `Params` or as arrays. The `Param` class provides a mechanism for passing `Values` as arguments to instances of class `Closure`. Single-dimension arrays of `Values` are implemented in a similar fashion using a singly linked list, but another link – `arrayLink` – is used.

5.2.1.1 Utility methods

The following methods allow `Values` to be passed as arguments and arrays:

- `void setLink(Value *)` – links two `Values` together.
- `void setArrayLink(Value *)` – links an array `Value` to another `Value`.
- `void setLength(int)` – assigns a linked of `Values` a length.
- `Value *getLink()` – returns a linked `Value`’s link.
- `Value *getArrayLink()` – returns an array-linked `Value`’s link.
int getLength() – returns the length of a linked Value list. Primarily the class Param uses this length.

int getArrayLength() – return the length an array-linked Value list. This length is used for linking arrays of Values.

bool isArray() – return whether a Value represents an array-linked list of Values.

friend ostream &operator <<(ostream &, const Value &) – prints the class name to an ostream object (STDOUT by default).

5.2.1.2 Overridden methods

The following methods are overridden by subclasses:

virtual const char *getClassName() – returns the name of the subclass.

virtual int getType() – returns the integer type of the subclass.

virtual Value *evaluate(Value *) – Closure defines this method to evaluate the passed in Value on a subtree.

virtual void out(ostream &) – prints the class name to an ostream object (STDOUT by default), but can be overridden, unlike the overloaded operator << method above.

virtual bool isEnd() – returns 0, but the class EndValue class overrides this method to return 1.

virtual bool isClosure() – returns 0, but the class Closure overrides this method to return 1.
5.2.1.3 The **EndValue** class

The class **EndValue**, derived from **Value**, terminates a list of **Values** and differs from its parent class only by defining `isEnd` to return 1 instead of 0.

5.2.1.4 **Generated is and get methods**

For every generated subclass of **Value** there are generated **is** and **get** methods. Each generated class has the name `typeNameValue`, where `type` is the name of the underlying value. For example, for the three types **int**, **double**, and **char**, **Value** subclasses `intValue`, `doubleValue`, and `charValue` would be generated. The **is** method has the form `virtual bool isType()`. By default this method returns 0, and returns 1 only by the class that contains a value `type`. From the above example, the following methods would be present in the **Value** class and all subclasses:

- `virtual bool isint()`
- `virtual bool isdouble()`
- `virtual bool ischar()`

The class `intValue` would return 1 for **isint**, class `doubleValue` would return 1 for **isdoule**, and class `charValue` would return 1 for **ischar**. Likewise, methods of the form `get type` are created for each generated class, and these throw a `HarryLegsException` by default. Only the class `typeNameValue` will actually return a `type` from the call `getType`, all other class instances will throw an exception. From the above example, the following **get** methods would be present in the **Value** class and all subclasses:

- `virtual int getint()`
- `virtual double getdouble()`
- `virtual char getchar()`
When obtaining the underlying value of a `Value` instance, one may either poll for the type via the `getType` method or `is` methods or use a try-catch block to catch a `HarryLegsException`.

### 5.2.2 The `Param` and `EndParam` Classes

The class `Param` encapsulates a container for a `Value` and a name to be passed as an argument to a `Closure` object. `Param` instances may be linked to form a linked list via the member `Param` `link` and also contain a `Value` instance and string `name`. This `Param` may be the end of an argument list, in which case this object would be of the `Param`-derived class `EndParam`. Class `Param` provides the following functionality:

- `int getType()` — returns the type of the member `Value` instance.
- `const char *getName()` — returns the name given to the member `Value` instance.
- `Value *getValue()` — returns the member `Value` instance.
- `virtual bool isEnd()` — return 0, but the class `EndParam` overrides this method to return 1.
- `void setLink(Param *)` — links `Param` instances together to construct a linked list of `Params`.
- `Param *getLink()` — returns a `Param`’s link.
- `friend ostream &operator << (ostream &, const Param &)` — passes a string representation of the passed in `Param` to an `ostream` (STDOUT by default).

As mentioned previously, instances of class `EndParam` are used to terminate a list of `Params`. The only different method is `isEnd`, because `EndParam` defines this method to return 1 instead of 0.
5.2.3 The Closure Class

This class encapsulates a procedure or function that can be evaluated and is implemented by containing a member `nonTerminalNode` instance body to execute and `Env env` in which to execute body. Unlike other `Value` subclasses, the class contains an `evaluate` method, analogous to that found in `treeNode`-derived classes, and this method evaluates by passing `env` into the `execute` method of `nonTerminalNode`. It was noted before that `nonTerminalNode` did not contain an `evaluate` method, but instead the pure virtual `execute` method. This method will evaluate body and return the result.

This class is useful in the language and provides the capability of implementing a target language with first-class functions, because `Closure` is derived from `Value`, and, thus may be passed around as an argument, returned from functions, and assigned to a variable.

5.2.4 Support for Arrays

Support for arrays was mentioned earlier, but this section describes this feature in more detail. An array is constructed by using a `Values setArrayLink` method. This method applied repeatedly will construct a singly linked list of `Values`. Each addition to an array list increments the length of the head node, so that node contains the length of the entire list. This length is accessed using the `getArrayLength` method. The `getArrayLink` method accesses the values of a value array. This method will return the next item in an array list and should be applied until an `EndValue` instance is encountered, because this class is used to denote the end of the array.
5.2.5 Name Conversion

Types supplied by the programmer that are not valid C++ class names are converted into valid C++ class names using the following table. All systems that use the type system use this conversion technique.

Table 2. Conversion table for class names.

<table>
<thead>
<tr>
<th>PROGRAMMER SUPPLIED STRING</th>
<th>CONVERTED STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Lt</td>
</tr>
<tr>
<td>&gt;</td>
<td>Gt</td>
</tr>
<tr>
<td>*</td>
<td>Star</td>
</tr>
</tbody>
</table>
CHAPTER 6
Environment System

The environment system provides the capability to bind names to Value instances in the target language. First, the goals and overall structure are presented, followed by the implementation of this system.

6.1 Goals and Overall Structure of the Environment System

The main goal of the environment system, like that of the Value system, was to provide the programmer a generic, flexible system in which to evaluate a target language program using the type system described in the previous chapter. This system should offer choices such as static/dynamic binding and weak/strong typing, and these features should be provided both efficiently both in time and space. Additionally a robust error handling system should be in place to support a flexible evaluation environment.

6.2 Implementation of the Environment System

This system is composed of linked hash tables that provide a number of access and insertion methods. Each component of this system is now discussed.

6.2.1 The HashNode Class

The core of the environment system is a linked list of hash tables, and each hash table stores and retrieves HashNode instances. Each HashTable bucket points to a linked list of HashNode instances where the HashNode class embodies a Value value indexed by the key string name. The HashNode class provides the following access methods:

- const char *asString() – returns a string representing the value-name pair.
- const char *getName() – returns a duplicate string of name.
• Value *getValue() returns value.

Additionally, the HashNode class provides the following methods for insertion into a HashTable. The methods parallel those found in the HashTable, so that an insertion method \( M_{HT} \) in HashTable will simply call the corresponding method \( M_{HN} \) method of HashNode.

• void insert(HashNode *node) inserts the passed in HashNode node into a list without checking the type of the underlying Value of node. If a HashNode in the list already contains the same name as this instance, this instance’s value is overwritten. If this instance is not the end of the list, the node is passed to the linked HashNode by calling insert of the link. Otherwise, this instance is the end of the list, and this instance’s link is set to node.

• void assign(HashNode *node) assigns value the Value node->value if name is the same as node->name. The same technique for traversing the list is performed as in insert, except that instead of inserting node if there is no HashNode with node->name, an IdentifyNotFoundException is thrown. Additionally, if there is a HashNode such that its name equals node->name, but the type of its value and node->value differ, a BadValueException is thrown. This method would be used for a strongly typed target language that required the user to declare variables before using them.

• void writeOverValue(HashNode *node) performs the same function as assign, except this method does not check whether a HashNode already exists with name that equals node->name. This method would be used is a strongly
typed target language that did not require variables to be declared before being used.

- void writeOverType(HashNode *node) — this is the most flexible insertion method. If performs the same function as writeOverValue, except it does not check that node->value has the same type as a node with the name node->name. This method would be used in a weakly typed target language that did not require variables to be declared before begin used.

HashNodes provide the following mechanism for retrieval of a value:

- Value *lookUp(char *in_name) — if this node is empty a IdentifierNotFoundException is thrown. If in_name equals name then value is returned, otherwise if this node is not the end of a list lookUp is called on the next node, and, finally, an IdentifierNotFoundException is thrown if this is the end of the list.

Lastly, every HashNode is empty by default so that a HashTable can initially contain empty buckets. To enforce the emptiness notion, only one constructor is provided that will create a HashNode with non-null value and name members, and this constructor also accepts a bool flag to designate this HashNode as empty or non-empty. It is desirable to create empty HashNode that still have names and values, and that was the reasoning behind this design decision. The following methods are provided for construction and utility:

- HashNode() — creates an empty HashNode with null value and name.
• HashNode(char *in_name, Value *in_value, bool isEmpty) – creates a HashNode with name in_name and value in_value whose emptiness depends on isEmpty.

• bool isEmpty() – return whether a HashNode is empty or not.

• void out(ostream& o) – passes the string returned from asString to the ostream o and calls out on the next node if this node is not the end of the list. This technique allows the method out to print out a list of HashNodes rather than just one.

6.2.2 The HashTable Class

The HashTable class is implemented as a variable-length array of HashNodes. The hash function used is hashpjp found in Abelson et al. [ASU88], and initially all buckets are filled with empty HashNodes. The same insertion methods are provided to the user as HashNode provides to HashTable, except these methods accept a string and Value instead of a HashNode as arguments. Each of the following methods $M_{HT}$ create a non-empty HashNode $N$ from the string and Value, hashes to that bucket, and calls the HashNode method $M_{HN}$ on the node to which was hashed. This procedure shows the usefulness of creating empty hash nodes, because insertion into the HashTable is analogous to insertion into a HashNode:

• void insert(char *in_name, Value *in_value)

• void assign(char *in_name, Value *in_value)

• void writeOverValue(char *in_name, Value *in_value)

• void writeOverType(char *in_name, Value *in_value)

Additionally, class HashTable provides the following methods for retrieval from and utility.
• Value *lookUp(char *name) -- following the insertion structure, first a HashNode is created from name and an empty Value, then lookUp is called on the HashNode bucket indexed by the hash function.

• void out(ostream &o) – the method out is called on all of the HashNode buckets.

• void clear() – clears all the buckets of the HashNode array, and inserts new empty HashNodes.

6.2.3 The Env Class

This class has appeared previously in this paper and implements a linked list of HashTables by extending HashTable, hence providing the same interface, and including an extra member, envLink, that is linked to another Env instance. This class does implement two methods differently than HashTable and a new method; these follow:

• void assign(char *in_name, Value *in_value) – performs the HashTable::assign method on itself that will search only this table and catches any EnvException. If an exception is caught and a linked Env instance does not exist that exception is thrown again. If there is a linked environment Env instance assign is called on that instance, so this allows the user to assign names found in linked environments. The other three insertion methods were not changed because none of those methods searched for a name to which a value was to be bound. Instead, insert, writeOverType, and writeOverValue all bind a value to a name if that name exists in the current environment, otherwise inserts a new binding.
• **void assignProtected(char *in_name, Value *in_value)** – this method performs just as the above assign, except it will throw a BadValueException and will not assign to an Env instance that has no link. Such an instance is the global environment, and this method is used to implement primitive functions and variables in the target language.

• **Value *lookUp(char *in_name)** – following the technique of assign, this method first looks up the string in_name in the current environment, and catches any EnvExceptions. If an exception is thrown and there does not exist a linked Env instance, then that exception is thrown again. If there is a linked Env instance, lookUp is called on this instance, and this allows the user to look up names in more than one scope.

An instance of Env is passed in parse tree node evaluate methods, and this provides the parse tree evaluation an environment in which to execute.

6.2.4 **The EnvException Class and Subclasses**

The main exception class thrown in this system is EnvException, and this class represents a message given to the user due to an error in the environment system. It provides the following constructor and utility method:

• **EnvException(char *in_message)** – constructs a new exception that contains the string message in_message.

• **const char *asString()** – returns a duplicate of the message string.

The following are classes derived from EnvException that each encapsulate a common error that can occur in an evaluation environment:
• DuplicateIdentifierException – thrown when one tries to insert HashNode with a name already existing in the current environment.

• IdentifierNotFoundException – thrown when tries to look up a name in an environment, but no HashNode exists with that name.

• BadValueException – thrown when types clash and for miscellaneous errors.

6.2.5 Evaluation of a Parse Tree using the Env Class

There are two alternatives to evaluate a parse tree using the Env class: (1) begin evaluation with an empty environment, and (2) begin evaluation with an environment already populated with Values. The first alternative is done by first allocating a new Env instance and then passing this instance to the head tree node’s evaluate method. This could also be achieved by calling the node’s evaluate method with no arguments that creates allocates a new environment and passes this environment to the evaluate methods of all its children.

The second form of evaluation can be useful when the target language contains primitive functions and variables that exist in the environment before execution. Populating an environment consists of using the insertion methods provided by the class Env to insert functions (instances of Closure or UserDefinedClosure\textsuperscript{12}) or variables (instances of Value), and this population can occur at a scope accessible to the user of a global scope. Scopes accessible to the user allow the user to overwrite primitive values, and this is done by opening an environment, populating it, and passing this environment along for evaluation. Global scopes are not accessible to the user; hence the user cannot overwrite primitive values. Opening an environment $E_1$, populating it, creating a new environment $E_2$ linked to $E_1$, and passing $E_1$ along for evaluation by the parse tree
accomplishes this. When using a protected global environment, the method
\texttt{assignProtected} (introduced previously) must be called instead of \texttt{assign} to ensure
global values are not overwritten.

\footnote{The class \texttt{UserDefinedClosure} is discussed in a later chapter.}
CHAPTER 7
Multiple Dispatch System

The dispatch system is responsible for calling the right procedure or operator from an operator set $O_p$ for one or more $\text{Value}$-type arguments $\tau_1, \tau_2, \ldots, \tau_n \in T$. Dispatching a procedure in Harry LEGS consists of receiving arguments, performing the appropriate coercions on these arguments, and then performing the appropriate procedure on the coerced arguments. A number of classes implement this in different ways and are discussed later, but first the goals of the system are presented.

7.1 Goals of the Dispatch System

The first goal of this system was to provide a general mechanism for coercing and operating on one, two, or three $\text{Value}$ arguments. A secondary goal was to provide a means of describing the coercions and operations needed.

7.2 Implementation of the Dispatch System

Classes derived from $\text{AbstractDispatcher}$ perform the actual method dispatch in this system. Each instance of an $\text{AbstractDispatcher}$-derived class represents one function or operator and contains a table or list of dispatch entries, where each dispatch entry contains the appropriate coercion and operation function pointers. The types of the $\text{Value}$ arguments index the dispatch entry, and the coercion and operation methods are then applied to these arguments.

An access class $\text{Dispatcher}$ contains one static evaluation method for each operator or function $o \in O_p$ and is responsible for calling the right
AbstractDispatcher-derived instance to perform this operation. Lastly, a utility class Maker creates the tables for each AbstractDispatcher-derived instance.

7.2.1 The AbstractDispatcher Class

This class defines the following five methods that are overridden by derived subclasses:

- virtual Value *execute(Value *v1)
- virtual Value *execute(Value *v1, Value *v2)
- virtual Value *execute(Value *v1, Value *v2, Value *v3)
- virtual Value *executeWithArgs(Value *args)
- virtual int getArity()

The first three execute methods perform a certain operation on the argument or arguments based on the type of these arguments. By default these methods return null. Derived subclasses should define one of these methods to handle its underlying operation and the other two to either throw an exception or try to recover from the error. For example, a binary dispatcher may implement the two-argument execute method to perform its underlying operation, define the one-argument execute method to return that argument, and throw an exception if three arguments are passed. The forth method, executeWithArgs, receives a linked list of Values representing arguments and performs the appropriate action on the arguments. Lastly, the method getArity returns the arity (unary, binary, or ternary), depending on the subclass.

Each AbstractDispatcher subclass implements a table or list of function entries, and these entries, whose definition is found in the appendix, are defined along with AbstractDispatcher. There are two groups of subclasses: full (FADD) and sparse (SADD) AbstractDispatcher-derived classes. The FADD classes should be used for
operations that are defined over a large portion of the type set \( T \), where the SADD classes should implement operations that are defined over a limited number of types of the set \( T \). The choice of groups is subjective, but FADD classes provide faster dispatch for operations defined over a large range of the type set, where SADD provide more space efficiency for operators defined over a limited range of the type set.

7.2.2 FADD Classes

There are two FADD classes: BinopDispatcher and UnopDispatcher. As the names suggest, BinopDispatcher instances perform binary operations and UnopDispatcher perform unary operations.

7.2.2.1 The UnopDispatcher class

Each UnopDispatcher instance contains a single-dimensional array of UnopEntry instances called table and implements only the one-argument execute method, throwing an exception for the other two. An instance is constructed using the method

\[
\text{UnopDispatcher}(\text{UnopEntry} \ast t, \text{int} r, \text{char} \ast \text{name})
\]

Figure 12. UnopDispatcher constructor.

The execute method is performed by first looking up the appropriate UnopEntry \( e \) in table, performing \( e.\text{conv1} \) on the first argument \( v1 \), and then applying \( e.\text{op} \) to the result of the conversion. The following example depicts the execute method of a UnopDispatcher instance defined for negation over a number type set \( T_{\text{JRC}} = \{\text{Integer, Real, Complex}\} \), where only negation on Complex numbers is defined and the argument name is 3.
7.2.2.2 The \textbf{BinopDispatcher} class

This class is similar to \textbf{UnopDispatcher}, except it contains a two-dimensional array of \textbf{BinopEntry} instances and implements only the two-argument \texttt{execute} method.

An instance is constructed from the method

\texttt{BinopDispatcher(BinopEntry **t, int r, int c, char *name)}.

The execute method also performs similarly to the one-argument execute method of \textbf{UnopDispatcher} -- it first looks up the \textbf{BinopEntry} \texttt{e} in table, then performs \texttt{e.conv1} and \texttt{e.conv2} on \texttt{v1} and \texttt{v2}, and then applies \texttt{e.op} to the results of the conversions. The following example depicts the execute method of a \textbf{BinopDispatcher} instance defined for multiplication over the above type set $T_{JRRC}$, where multiplication is only defined over \texttt{Complex} numbers. The arguments are 4.3 and 5 +6j:

\[
(4.3, 5.6) \rightarrow_{conv1, conv2} (4.3 + 0j, 5 + 6j) \rightarrow_{op} (21.5, 25.8j)
\]

Figure 14. Binary coercion and operation.

7.2.3 SADD Classes

There are three SADD classes: \texttt{SparseUnopDispatcher}, \texttt{SparseBinopDispatcher}, and \texttt{SparseTrinopDispatcher}. As the names suggest, these each define only the one-argument, two-argument, and three-argument execute methods, and contain a single dimensional array instance of \texttt{SparseUnopEntries},
SparserBinopEntries, and SparseTrinopEntries respectively. These sparse entries differ from the entries of the FADD classes because they are not indexed according to type, but instead contain fields for the argument types over which they operate. To find an entry based on arguments types \(\tau_1, \ldots, \tau_n\), the entry list \(L = \{\text{type}_1, \text{type}_2, \ldots, \text{type}_n\}\) is searched until an entry is found such that \(\text{type}_1 = \tau_1, \ldots, \text{type}_n = \tau_n\). Once this entry is found, the same procedure that converted and operated on the FADD class arguments is performed and this result is returned. If an entry is not found an exception is thrown, because the operation desired was not defined over the passed in argument types.

The following methods construct sparse dispatchers:

- `SparseUnopDispatcher (SparseUnopEntry *t, int r, char *n)`
- `SparseBinopDispatcher (SparseBinopEntry *t, int r, char *n)`
- `SparseTrinopDispatcher(SparseTrinopEntry *t, int r, char *n)`

7.2.4 The UserDefinedClosure Class

This class, derived from `Closure`, is used to populate an environment with dispatchers to provide primitive functions to a target language\(^\text{13}\). The reader may recall from Chapter 5 that a `Closure` is derived from `Value` and defines its evaluate method to call the underlying `nonTerminalNode` body’s `execute` method. `UserDefinedClosure`, instead, contains three dispatchers -- `dispatcher1`, `dispatcher2`, and `dispatcher3` -- and defines its evaluate method to call `executeWithArgs` on one of the dispatches according to the number of arguments. For example, if an argument list were passed in of length 2, the result of `evaluate` would be `dispatcher2->evaluate`. Containing three

---
\(^\text{13}\) Chapter 6 discusses environment population.
dispatchers allows the target language to have overloaded procedure names with differing number of arguments

7.2.5 The Dispatcher Class

Access to the dispatch library is performed through static methods found in the Dispatcher class. An instance of this class contains one method $M_{op}$ and one instance $D$ of an AbstractDispatcher-derived class for each operation $op$. To dispatch the operation $op$ over some arguments, dispatcher $D$’s execute method is called and the result of this method is returned.

In addition, this class may be used to populate an environment through the static populate method whose signature is shown below

\[
\text{static void populate(Env *in_env)}
\]

Figure 15. The populate signature.

This method inserts new UserDefinedClosure instances created from the static dispatcher members into the in_env, so that these operations may be used in the global environment of a target language. This is done through the Env class method writeOverType.

Each dispatcher is created from a make function defined in the Maker module (described below), that fills an entry array with functions and then returns a new dispatcher constructed from that array.

7.2.6 Generating Dispatchers with the Maker Module

The Maker module is responsible for filling an array with coercion and operation functions and then returning a new dispatcher constructed from that array. The module
defines a number of public make functions called make\texttt{name}, where name is the name of the dispatcher, and these functions are called by a Dispatcher instance to create each dispatcher instance. This module may be defined by hand, but is easily generated using the system described next.

7.3 Automatic Generation of Dispatchers and Functions

Two Perl scripts were used to automatically generate the Dispatcher library: \texttt{j\_genFuncs.pl} and \texttt{j\_genMaker.pl}. The first script translates functions operating on arbitrary types into new functions operating on the corresponding \texttt{Value} types according to a specification file. The second script generates make functions for the \texttt{Maker} module that constructs dispatchers based on a file specifying the coercion scheme for each operator.

7.3.1 Function Specification

The purpose of this script is to take a description of the operations that are implemented over a set of types in the base domain and generate a set of wrapper functions that perform the same operations over the corresponding \texttt{Value} types in the Value domain. Every generated wrapper function takes \texttt{Value}-type arguments, extracts the underlying elements, and performs the specified operation on these elements. This is needed so that the op function in the dispatch entries can call these generated methods with its \texttt{Value} arguments. The following example motivates the need for wrapper functions and depicts the conversion from a base function into a wrapper function in the Value domain. Consider the method equal that returns a C++ primitive \texttt{bool} whose value depends on whether the int-valued arguments are equal

\begin{verbatim}
    bool equal(int a, int b) { return a == b; }
\end{verbatim}
The following line specifies a conversion into the \texttt{Value} domain according to the \texttt{j_genFuncs.pl} script:

\begin{verbatim}
%binary equal : bool int int
\end{verbatim}

Figure 17. Conversion specification.

From this specification the following \texttt{Value}-domain function would be generated:

\begin{verbatim}
Value *equal_int_int(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << ":" << __LINE__ << ":" equal(int,int) " << endl;
#endif
    return new boolValue(equal(v1->getint(), v2->getint())); }
\end{verbatim}

Figure 18. A \texttt{Value}-domain function from the \texttt{equal} function.

This function may now be called by \texttt{op} with \texttt{Value}-type arguments. The grammar for a specification file is found in the appendix along with an example called \texttt{example.spec} with generated files, but the following describes its semantics.

7.3.1.1 The \texttt{.spec} specification file

The first section contains pairs of the form \texttt{abbreviation ':' name}, where \texttt{abbreviation} will represent \texttt{name} for the rest of the specification. The next section contains the function statements. A function statement may begin with one of the following directives:

- \texttt{%unary} – denotes a unary function statement.
- \texttt{%binary} – denotes a binary function statement.
- \texttt{%member} – denotes a unary member method.
Following each of these directives is one or more operation identifier. An operator identifier is either a valid C++ operator enclosed in single quotes or a function name. A function will be created for each of these operator identifiers whose name will be the name of the operator identifier followed by the arguments types separated by underscores. If the operator identifier is a C++ operator its name will be converting using the conversion table found in the appendix. Lastly, following the colon is the return type followed by the argument type or types. A star (’*’) may precede these types to denote a pointer to that type. In the resulting function, the pointer will be dereferenced before being used in the function. The following example specifies maximum and minimum functions on a float scalar and a pointer to a float image that returns a pointer to a float image:

```
%binary max min: *FloatDI *FloatDI float
```

Figure 19. Function specification for max and min functions.

The resulting functions are then generated:

```c
Value *max_FloatDI_float(Value *v1, Value *v2) {
    #ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: max(*FloatDI, float)" << endl;
    #endif
    return new IA_ImageLtIA_PointLtintGtCommafloatGtStarValue
        (new IA_Image<IA_POINT<int>,float>(
            max(*(v1->getIA_ImageLtIA_PointLtintGtCommafloatGtStar()),
            v2->getfloat()));
    }

Value *min_FloatDI_float(Value *v1, Value *v2) {
    #ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: min(*FloatDI, float)" << endl;
    #endif
    return new IA_ImageLtIA_PointLtintGtCommafloatGtStarValue
        (new IA_Image<IA_POINT<int>,float>(
            min(*(v1->getIA_ImageLtIA_PointLtintGtCommafloatGtStar()),
            v2->getfloat()));
    }
```

Figure 20. Resulting max and min Value-domain functions.

7.3.1.2 The .skel files

Not all functions can be specified in this straightforward manner, so the
j_genFuncs.pl script will append the generated functions to two files
Functions.skel.{h,cpp}. These files should contain any other functions needed, so the
resulting files Functions.{h,cpp} will contain every function needed by the dispatchers,
and the next section discusses the generation of these.

7.3.1.3 Additional functions and requirements

The functions no and id are automatically added to the Functions.{h,cpp} files
and act as “place holders” in dispatch entries. The function no throws an exception
denoting that a function cannot be performed on certain types, but is allowed in a
dispatch entry because it is defined with the correct signatures. These function signatures
are shown below.

- Value *no(Value *v1)
- Value *no(Value *v1, Value *v2)
- Value *no(Value *v1, Value *v2, Value *v3)

The function id with the signature Value *id(Value *v) returns the argument v
and holds the place of a coercion function in dispatcher entries that does not need a
conversion. Like the function no, this function has the correct signature, so may appear
in dispatcher entries.

Lastly, all conversion functions should appear in the .skel files, and these
functions must have the signature Value *type1_to_type2(Value *v), where type1 is
converting into \textit{type2}, and both appear as abbreviations in the first part of the .spec file. The following is an example of a valid signature:

\begin{verbatim}
Value *int_to_float(Value *v);  /* int -> float */
\end{verbatim}

Figure 21. Example conversion signature.

7.3.2 \textbf{Dispatcher and Coercion Specification}

The purpose of the \texttt{j\_genMaker.pl} script is to read a description of the coercion the dispatcher will implement. The input to this script is the .tab specification file (described below), and the output of this script are two files \texttt{Maker.\{h,cpp\}} that define make functions returning FADD and SADD instances implementing the specification file. These \textit{make} functions are used by the \texttt{Dispatcher} to allocate its dispatchers. The grammar for this specification files is found in the appendix along with an example .tab file, but the following describe its syntax.

7.3.2.1 The .tab specification file

A .tab specification begins with the directive \texttt{%default_dim} followed by the default dimension of all \texttt{BinopEntry} and \texttt{UnopEntry} arrays. Following this is zero or more dispatcher declarations, and each declaration contains a \textit{directive line}, followed by a \textit{definition section}, followed by the directive \texttt{%end}. The directive line contains a \textit{dispatcher directive} that specifies the type of dispatcher returned followed by one or more operator identifiers (as described in .spec file). Additionally, the size of the dispatch entry array may be specified at the end of the line. The result of each dispatcher declaration is a make function for each operator identifier in the dispatcher directive whose signature is \texttt{AbstractDispatcher *make\_name()}, where \textit{name} is the operator identifier or converted
operator identifier. The following lists the valid dispatcher directives and the dispatcher
type each creates:

- %binop – BinopDispatcher
- %unop – UnopDispatcher
- %sparse_unop – SparseUnopDispatcher
- %sparse_binop – SparseBinopDispatcher
- %sparse_trinop – SparseTrinopDispatcher

The definition section is of one or more definition lines. A definition line consists of
the argument types followed by the types to which these arguments are coerced. Each
coerced type must be preceded by a ‘!’, and the special coerced type id denotes no
coercion. One dispatcher entry is created for each definition line, and an optional ‘*’ may
follow a definition line in a binop dispatcher declaration denoting that two entries should
be created – each a permutation of the other. In addition, a function name may follow a
definition declaring that the following function should be called.

For example, consider a dispatcher system over the fictitious type set $T_{JRC} = \{\text{int}, \text{real}, \text{comp}\}$ introduced earlier. For this dispatcher, only operations on complex
numbers (type comp) are permitted so ints and reals are both coerced to complex
numbers. The following dispatcher declaration in file example.tab found in the
appendix would create two functions – makeplus and makediv – each returning
BinopDispatchers over $T_{JRC}$:

The resulting report would be printed in example.tab.out, Maker.h would hold
the header, and Maker.cpp would hold the source. These files are listed in the appendix, also.
7.3.2.2 Make functions

There are three parts of a make function. The first part fills a dispatch entry array with no functions. These all throw exceptions and represent types that are not defined over the operation for which the dispatcher is being created. The next step is to fill this array with the coercion and operation functions supplied by the programmer. This array is indexed by the integer type assigned to each Value subclass, and this information must be provided by the programmer in the .skel.h file in the form of #define statements. These definitions have the form v_abbrev, where abbrev is the abbreviated type specified in the .spec file. The following are examples of these definitions:

```c
#define V_int   0 /* int        */
#define V_IA_PointLtintGtStar 1 /* IA_Point<int> * */
```

Figure 22. Valid type definitions.

Additionally all conversions taken from the .tab file will be translated using the method described in section 7.3.1.2. So the binary definition line for addition

```
int   float !float    id
```

Figure 23. Coercion specification.

would produce the following array entry:

```c
    t[V_int][V_float].conv1 = int_to_float;
    t[V_int][V_float].conv2 = id;
    t[V_int][V_float].op    = plus_float_float;
```

Figure 24. A coercion/operation tuple array entry.
The last step is to return a new dispatcher with the array \( t \) as its dispatcher entry array.
CHAPTER 8
Original Generation System and Modifications to this System

The original treenode generation system included a grammar parser written in Lex
and Yacc to parse the specification grammar that output a shell script. This generated
shell script, whose name was the grammar name with an appended ‘.sh’, then called a
number of Perl scripts to generate the C++ treenode classes. These classes were built
from skeleton files (.skel files).

8.1 The Original Specification Grammar

An original grammar specification began with a text section copied directly into
the generated Lex file, and this section was useful for creating regular expressions used in
specifying editable terminals. Following this section the programmer was able to specify
the prefix for all generated classes, non-editable terminals, editable terminals,
nonterminals, and right hand sides. The following sections describe how each was
specified, and annotate an example grammar specification, spec.gram, found in the
appendix.

8.1.1 Lex text section

This section was copied directly into the Lex file, <PREFIX>lex.l, generated
from the grammar parser. The following gives an example of a definition that could be
used to specify a digit and white space later in the grammar:
8.1.2 Prefix specification

The prefix for the generated grammar was specified with the %prefix directive followed by the desired prefix. For example, the following line in spec.gram would cause all classes to begin with the string ‘STD’.

```
%prefix 'STD'
```

Figure 26. Prefix specification.

This prefix would appear in many of the generation scripts for the remainder of the generation process.

8.1.3 Non-editable Terminals

Non-editable terminals were specified with the %term directive followed by the name of the terminal, the string representation, a pre-formatting specification, and a post-formatting specification. Valid pre- and post-formatting characters consisted of the following:

- **n** – new line
- **t** ’ tab
- **NULL** – no formatting

Pre-formatting preceded terminals in the editor and post-formatting followed the terminals. Each non-editable terminal line created a call to the Perl script
The following example specifies a terminal semicolon named ‘SEMI’ that would have a new line character appear after it when printed to the screen:

```
%term SEMI ‘;’ ‘NULL’ ‘n’
```

Figure 27. Non-editable terminal specification.

This line would cause the following call to appear in `spec.gram.sh`:

```
genNonEditableTerminal.pl STDSEMI ‘;’ ‘NULL’ ‘n’
```

Figure 28. Perl call from shell script to generate a non-editable terminal.

The script `genNonEditableTerminal.pl` took as arguments the terminal name, string representation, pre-formatting, and post-formatting and would then use the skeleton files `nonEditableTerminal.{cpp,h}.skel` and these arguments to generate the files `STDSEMI.{cpp,h}`. The pre- and post-formatting characters are ignored if omitted or labeled ‘NULL’, so to specify post-formatting and no pre-formatting one would label the pre-formatting ‘NULL’.

### 8.1.4 Editable terminals

Editable terminals were specified with two directive lines beginning with `%eterm` and `%etermpat`. The first line consisted of the `%eterm` directive followed by the terminal name and initial string representation. The second line consisted of the `%etermpat` directive followed by the terminal name and a single-quoted regular expression describing the editable terminal. The lines in the below figure:

```
%eterm       INT ‘0’
%etermpat    INT ‘(DIGIT)+’
```
Figure 29. Editable terminal specification for an integer.

specify an integer using the definition made in the top of the grammar and would create the following call in `spec.gram.sh`:

```bash
    genEditableTerminal.pl STDINT '0' '{DIGIT}+'
```

Figure 30. Perl call in the shell script to generate an editable terminal for an integer.

The script `genEditableTerminal.pl` took as arguments the editable terminal name, initial string representation, and regular expression and would create the files `STDINT.{cpp,h}` from these arguments and the skeleton files `editableTerminal.{cpp,h}.skel`.

8.1.5 Nonterminals

A string followed by a semicolon and zero or more right hand sides specified a nonterminal. The lines in the below figure specify a nonterminal named `expression` with four right hand side children

```plaintext
expression: expression PLUS expression
           |   expression MULT expression
           |   PRINT LPAREN expression RPAREN
           |   INT
;
```

Figure 31. Nonterminal specification for an expression.

and would produce the following line in `spec.gram.sh`:

```bash
    genNonTerminal.pl STDexpression "<expression>" 4 STDexpression_RHS0 \
                   STDexpression_RHS1 STDexpression_RHS2 STDexpression_RHS3
```

Figure 32. Perl call in the shell script to generate a nonterminal.
The script `genNonTerminal.pl` would take as arguments the nonterminal name, string representation, number of child right hand sides, and the children names and would create the files `STDprogram.{cpp,h}` from these arguments to and the skeleton files `nonTerminal.{cpp,h}.skel`. The textual representation of the nonterminal would be ‘<program>’, and this would appear as a nonterminal to the user, but as a terminal symbol to the generated lexical analyzer. Additionally, the first nonterminal in the grammar denoted the start symbol.

8.1.6 Right hand sides

Right hand sides were specified in the same group as nonterminals and included one or more nonterminal names that would be the children nonterminals of the generated right hand side. From the above nonterminal example, the line

```
| PRINT LPAREN expression RPAREN
```

Figure 33. Right-hand side specification.

specifies a right hand side to expression that has four children and would create the following call in `spec.gram.sh`:

```
  genRHS.pl STDexpression_RHS0 4 STDPRINT LPAREN expression RPAREN
```

Figure 34. Perl call in the shell script to generate a right-hand side.

The script `genRHS.pl` would take as arguments the right hand side name, number of children, and children names and would generate the files `STDprogram_RHS0.{cpp,h}` from these arguments and the skeleton files `RHS.{cpp,h}.skel`.

8.1.7 Additional scripts

The grammar parser also invoked the following scripts:
• **genBaseVisitor.pl** – generated the visitor file `nodeVisitor.h`.

• **genVisitor.pl** – generated the visitor subclasses.

• **genStart.pl** – generated the start symbol in the `<PREFIX>grammar.{cpp,h}` files.

• **genClassID.pl** – generated the file `classID.h` that created a enumeration (enum in C++) of all the class names.

• **genRules.pl** – generated the Flex and Bison files `<PREFIX>lex.l` and `<PREFIX>gram.y` from the skeleton files `lex.l.skel` and `gram.y.skel`.

### 8.2 Changes Needed in the Generation System

The previous example grammar, `spec.gram`, was used to create a tree node library to which a hybrid editor was linked, but it lacked functionality. Due to this lack of functionality, the modifications in the editing system, modifications in the parse tree node system, and creation of a type and environment system the following changes were needed in the generation system.

• Condense the output files.

• Specify action routines for each right hand side.

• Access right hand side entities.

• Assign nonterminals and right hand sides return values.

• Specify Value subclasses.

• Describe editable terminals.

• Include needed files.

• Transport the new information via the actions file.

• Support operators.
• Hide right hand sides.

• Add precedence and associativity to terminals and operators.

• Allow nullable nonterminals.

• Comments.

In addition, all skeleton files and scripts were not prefixed with the string ‘j_’, so that the new system could easily be distinguished from the old system. To exhibit these changes this section will follow an example specification, calc.gram, whose form is similar to spec.gram of the previous section, but uses the new features of the grammar parser. This grammar appears in the appendix.

8.2.1 Condensing the Output Files

Originally every node class was outputted to a separate file, and this achieved unneeded modularity, because the programmer never intended to view these files. Additionally, the time to compile all these files greatly exceeded the time to compile the same amount of text in a fraction of the number of files. Hence, instead of outputting every node class to a file, all generated nodes were sent to one of the files

<PREFIX>NonEditableTerminal.{cpp,h}, <PREFIX>EditableTerminal.{cpp,h},
<PREFIX>OperatorNode.{cpp,h}, <PREFIX>NonTerminal.{cpp,h}, and
<PREFIX>RHS.{cpp,h}. This greatly reduced the compile time while still preserving the right amount of modularity to develop the system.

To implement this the following creation scripts were created that output heading information to the above ten files, so that the original scripts can append generated class information to these instead of creating a new file for each class:
Figure 35. New header Perl scripts.

These scripts require the following skeleton header files:

- `j_OperatorNode.header.{cpp,h}.skel`
- `j_RHS.header.{cpp,h}.skel`
- `j_editableTerminal.header.{cpp,h}.skel`
- `j_nonTerminal.header.{cpp,h}.skel`
- `j_nonEditableTerminal.header.{cpp,h}.skel`

Figure 36. New skeleton header files.

Lastly, the following scripts were altered to append generated class information to the files begun by the header creation scripts:

- `j_genEditableTerminal.pl`
- `j_genNonEditableTerminal.pl`
- `j_genRHS.pl`
- `j_genNonTerminal.pl`
- `j_genOperatorNode.pl`

Figure 37. New Perl scripts to produce the new classes.

8.2.2 Specifying Action Routines in Right Hand Sides

The right hand sides of the parse tree held the semantics of the underlying program, so it was necessary to give the programmer the ability to specify these semantics. The chosen vehicle was action routines. These action routines consist of C++ code directly transferred via the `actions` file into the evaluate methods of the right hand sides and were delimited by the strings `'( : ')'` and `'( :: ')'` as found in the LALR parser generator CUP [Hud99].
The following example from calc.gram shown in the appendix shows the use of action routines to specify the action of a print command.

```
| PRINT LPAREN expression RPAREN
{: Dispatcher::print(@3); :}
```

Figure 38. Action routine for a print command.

The ‘@3’ allows the programmer to access elements of the right hand side and is the subject of the next section.

8.2.3 Accessing Right Hand Side Entities

A clean method of accessing a right hand side’s children was needed, so the following symbols were introduced into the specification grammar parser:

- $n^{15}$ -- returns the $n^{th}$ right hand side child.
- $@n$ -- returns the Value-derived result of calling the $n^{th}$ right hand side child’s evaluate method.
- $!n$ – returns a Value-derived result calling the $n^{th}$ right hand side child’s execute method.

These are translated into the proper C++ code by the j_genRHS.pl script.

8.2.4 Assigning Nonterminals and Right Hand Sides Return Values

As was explained in Chapter 4, every nonterminal and right hand side return values. In particular, all children of a certain nonterminal $N$ must have the same return type as $N$. A new RHSNode subclass, $C_N$, with the same evaluate return type as $N$ is created for every nonterminal, and all children right hand sides of $N$ derive from this class $C_N$. Thus, every right hand of $N$ returns the same type as $N$, but how is this specified in

---

14 This file is discussed in a later section.
15 The ‘$n$’ denotes the $n^{th}$ child number between 1 and the number of children of a right hand side.
the grammar? Preceding every nonterminal is an optional return type. When omitted, the
return type is void. The following fragment from calc.gram shows that expressions
nonterminal return a pointer to Value:

```
Value*
expression: ...
```

Figure 39. Fragment showing a nonterminal return type Value pointer.

Consequently, the nonterminal CALCexpression, RHSNode-derived class
CALCexpressionRHS, and all subclasses of CALCexpressionRHS will return type Value*
and would appear as the first line of each actions files entry. Additionally, the editable
nonterminal return types may precede the declaration of such nodes¹⁶, as the following
example shows:

```
%petermpat int INT '{DIGIT}+' {:
  return i;
 :}
```

Figure 40. Editable nonterminal with return type int.

8.2.5 Specifying Value Subclasses

The grammar parser will automatically create a Value subclass for every
nonterminal or editable terminal return type with the restriction that no Value subclass
can have an underlying Value type -- i.e. Value, EndValue, or Closure. So, no Value
subclasses would be created from the first example in the previous section, but the
subclass intValue would be created from the second example.

¹⁶ Editable nonterminal declarations are explained in a later section of this chapter.
However the programmer may also use the `@value` directive to demand a subclass be created. In `calc.gram`, two subclasses, `intValue` and `MyPointValue`, are created from the following directives:

```
@value int
@value MyPoint
```

Figure 41. Forced `value` directives creating `intValue` and `MyPointValue`.

Two points should be noted. First, the first directive wasn’t needed because it appeared as the return type for an editable terminal, but the second was needed because it did not appear as a return type. Secondly, the order of these directives dictates that the subclasses `intValue` and `MyPointValue` will return type indices 0 and 1 from the method `getType`, respectively. Recall that the type indices used by the dispatch system is generated by the programmer, so the `@value` directive gives enables the programmer to control these indices.

To generate these new types a `returns` file is created that begins with all value type specified by value directives and contains all the return types in the grammar. This file is passed to the script `j_genValue.pl` that processes it and produces the files `Value.{cpp,h}` from the skeleton files `j_Value.{cpp,h}.skel` and `j_SubValue{cpp,h}.skel`. By beginning the returns files with the explicitly declared types rather than those appearing as return types, this script is ensured to produce value type indices starting from 0 that correspond to the declaration order.

8.2.6 Describing Editable Terminals

The original grammar parser allowed to the user to describe only two aspects of an editable terminal: the initial string representation and the regular expression that defines the terminal. Additional methods already existed in the generated subclasses of
editableTerminalNode that needed to be linked to the grammar specification, and these methods were:

- `void evaluate(Env *in_env)` – analogous to the `evaluate` methods of nonterminals and right hand sides.
- `void setVal(char *val)` – called various times by the editor as a result of the user inserting text into this node.
- `char *asString()` – called various times by the editor and must return the current string representation of this node.

Two declaration lines are still needed for an editable terminal, and those are the `eterm` and `etermpat` lines. The `evaluate` method is specified using an action routine following the `etermpat` declaration. An optional return value type may follow the `etermpat` declaration and serves the same purpose as the return type of nonterminals. The following example declares an editable terminal that returns an int:

```%etermpat int INT '{DIGIT}+' {: return i; :}
```

Figure 42. Editable terminal, `INT`, action routine.

Following the `etermpat` declaration is the `private` declaration, and this consists of the directive `%private` followed by an action routine specifying added members to this class. An optional value in C-style (`/* ... */`) comments may follow each member variable, and this value is used to initialize that member. For example the following private declaration declares an `int` member `i` initialized to 0 and an access method for `i`:

```%private {
    int i; /**<0*/
    int getI() { return i; }
}
```
The member methods are visible to tree node class and the editor. Following the private declaration is the \textit{asstring} declaration that consists of the \%asstring directive followed by an action routine specifying the code to be executed for every call to this node’s asString method. This method must return a char* that is the current string representation of this node. The following declaration returns a string that is an integer:

\begin{verbatim}
%asstring {:
    char str[100];
    sprintf(str, "%d", i);
    return strdup(str);
:}
\end{verbatim}

Finally, the \textit{setval} declaration consists of the \%setval directive followed by an action routine specifying the code to execute on every call this node’s \textit{setVal} method. A variable val of type char* is passed in as the only argument, and this is the value the user has tried to insert into this node. The following setval declaration receives the argument val and updates an int member \texttt{i}:

\begin{verbatim}
%setval {:
    char *bp;
    i = (int) strtod(val, &bp);
:}
\end{verbatim}

All of the following four declarations should be highly coupled, but, in particular, the setval and asstring declarations should work together so that the return value of asstring declaration will be the current value desired by the user. Sharing data between
the two methods, updating this data in every setval declaration, and returning a string representation of this data in the asstring declaration accomplish this.

8.2.7 Including Needed Files

The programmer may need to include files in the generated classes via the C++
#include directive and specify where to find these files in the generated makefiles. The
#include directive is used to specify headers that should be included in the generated classes, and the %includedir directive specifies where these headers are found. The following shows how to include a header for points found in another directory:

%includedir /cise/homes/jpalm/research/mypoint/include
#include "MyPoint.h"

Figure 46. Include and includedir directives for a header MyPoint.h.

The directory must be a valid system directory and the header must be a valid C++ header enclosed by double quotes or angled brackets. This information will be put in the actions file and accessible to all generation scripts.

8.2.8 Transporting the New Information Via the Actions File

The actions file, as mentioned, earlier, contains the new information that can be specified in the specification grammar along with some old information that was previously passed via arguments to the Perl scripts. This file is composed of global information followed by class entries for each class to be generated, and this information is delimited by begin and end tags. These tags, referred to by the form tag level, include the strings ‘begin’ or ‘end’ and the tag name and fall into one of the following three categories:

- **First level tags**: ‘***** begin tag *****’ and ‘***** end tag *****’.
- **Second level tags**: ‘--- begin tag ---’ and ‘--- end tag ---’.
- **Third level tags**: ‘<begin tag>’ and ‘<end tag>’.
Class tags are defined as second level tags with tag equal to the class name. The grammar for this file appears in the appendix, but the following section describes the global section and each class entry section.

8.2.8.1 Global section of the actions files

The global section contains the Lex specification, prefix, include headers, and include directories delimited by second level tags. Many generation scripts may process this information.

8.2.8.2 Right hand side entries in the actions file

The rhs first levels tags enclose this section, and class tags enclose each class entry. The first item in the class entry is the return type followed by a list of the children and the action code, both enclosed by third level tags.

8.2.8.3 Nonterminal entries in the actions file

The nonterms first level tags enclose this section, and class tags enclose each class entry. The first item in the class entry is the return type followed by a second level tag-enclosed list of the possible right hand sides.

8.2.8.4 Terminal entries in the actions file

The terms first level tags enclose this section, and class tags enclose each class entry. The first item in the class entry is the return type (always void) followed by a second level tag-enclosed formatting list.

8.2.8.5 Editable terminal entries in the actions file

The eterm first level tags enclose this section, and class tags enclose each class entry. The first item in the class entry is the return type followed by second level tag-
enclosed segments containing the action, private, asString, setVal, pattern, and formatting declarations.

8.2.8.6 Operator entries in the actions file

The operator first levels tags enclose this section, and class tags enclose each opgroup. With in each opgroup entry is a list of the member of that group.

8.2.9 Supporting Operators

The information that distinguishes operators from other terminals are the opgroup to which an operator belongs and its precedence. Every operator must belong to an opgroup, and an opgroup is specified with the %opgroup directive followed by the name and single quote-delimited string representation. The following example introduces a binary operator group with string representation ‘?’:

```
%opgroup BINOP '?'
```

Figure 47. Opgroup with string representation ‘?’.

This declaration would cause the following call to the script j_genOperator.pl:
```
j_genOperatorNode.pl CALCBINOP '?' CALC CALCBINOP actions.txt
```

Figure 48. Perl call to generate an opgroup head CALCBINOP.

This script takes as arguments the operator name, string representation, prefix, opgroup, and actions file. Note, that this call has added the operator CALCBINOP to the opgroup CALCBINOP. Creating an opgroup consists of organizing a collection of operators so that these operators can only rewrite into one-element right hand sides whose one child is an operator in the opgroup. Along these lines, the head of the opgroup is just a special member of the opgroup, because it can rewrite into any of the other
member, but none of the members (including the head) can rewrite into the head operator.

To create a member of an opgroup the \texttt{%op} directive is used followed by the opgroup name, operator name, and single quote-delimited string representation. The following declaration adds an addition member to the opgroup \texttt{BINOP}:

\begin{verbatim}
%op BINOP PLUS         '+'
\end{verbatim}

Figure 49. Addition operator added to the opgroup \texttt{BINOP}.

The call to create this operator would be:

\begin{verbatim}
j_genOperatorNode.pl CALCPLUS '+' CALC CALCBINOP actions.txt
\end{verbatim}

Figure 50. Perl call to generate an operation \texttt{CALCPLUS} in the opgroup \texttt{CALCBINOP}.

Every operator must be defined in a right-hand side of a production, but presenting the user with all these right-hand sides would defeat the purpose of grouping operators together. Instead, the right hand sides of an opgroup’s member are \textit{hidden}, and only the right hand side containing the head operator is presented to the user. This is explained in the next section.

8.2.10 Hiding right hand sides

Preceding a production with the \texttt{%hidden} directive hides right hand sides. The following illustrates an addition production that will not be presented to the user; instead the generic binary operator is presented.

\begin{verbatim}
| %hidden expression PLUS  expression {: ... :}
| expression BINOP expression {: ... :}
\end{verbatim}

Figure 51. A fragment example of hiding a right-hand side.

This instructs the system to discount the first production from the editor menu, and instead present the second generic binary operation. The usefulness of this feature
isn’t seen with one production, so instead consider a system of thirty binary operations. The programmer could now group these operators into an opgroup and hide all the right hand sides containing these operators. The user would now be presented with a binary operation with the ability to rewrite just the operator instead of the entire production.

8.2.11 Adding Precedence and associativity to terminals and operators

Because the programmer may now use opgroups and hidden right hand sides to define a set of operations in terms of one nonterminal, the precedence of the operators in these operations is no longer enforced by the grammar. Additionally, the programmer should have control over operator associativity, so this is done with the `%left` and `%right` directives in a way similar to Yacc. One or more operators with the same precedence and associativity may follow either of these directives, and the relative operator precedence is determined by the order in which these declarations appear in the grammar. Declarations appearing earlier in the grammar give those operators a lower precedent than those appearing later. For example the following lines specify the multiplication and addition operators as member of the opgroup `BINOP`, but give the multiplication operator a high precedence:

```
%op BINOP PLUS `+`
%left PLUS
%op BINOP MULT `*`
%left MULT
```

Figure 52. Giving the addition operator a higher precedence than the multiplication operator.

This can also be done with non-editable terminals, also.
8.2.12 Allowing nullable nonterminals

The empty right hand side followed by an action specifies nullable nonterminals.

The following (rather dull) example illustrates a nullable nonterminal:

```
[:: printf("I'm null!"); ;]
```

Figure 53. An example empty right-hand side.

A special right hand side with one child – a subclass of emptyNonTerminal – would be created using the `j_genRHS.pl` script in the same way other right hand sides are created.

8.2.13 Comments

C-style comments (`/* … */`) may be used in the grammar specification. This can allow the programmer to better document his code.

8.3 Final Generation Process

The final generation process is still very similar to the original process. First, the programmer writes a specification grammar and places in the TreeNode subdirectory of HELP. The programmer then invokes the makefile found in the directory and this begins the generation process. First, a shell script is generated and then executed consisting of calls to the generation Perl scripts. Following this the value classes are generated, then the tree node object library is produced and HELP can now be built linking to this library. A grammar for the specification grammar is found in the appendix.
CHAPTER 9
Implementation of the Image Algebra Language

The language generated for the Image Algebra was linked to the existing iac++ library written in C++. This section briefly describes the existing implementation\(^{17}\), the goals for the Harry LEGS design, and the final Harry LEGS implementation.

9.1 Overview of the Existing IAC++ Implementation

The existing iac++ class library includes the following template classes:

- \(\text{IA\_Point}<T>\)
- \(\text{IA\_Set}<T>\)
- \(\text{IA\_Pixel}<P,T>\)
- \(\text{IA\_Image}<P,T>\)
- \(\text{IA\_Neighborhood}<P,Q>\)
- \(\text{IA\_DDTemplate}<I>\)

where \(T\) is a C++ primitive type, \(P\) and \(Q\) are of type \(\text{IA\_Point}\), and \(I\) is of type \(\text{IA\_Image}\). \(\text{IA\_Point}\)s represent homogeneous points and support various unary, binary, and relational operations. Instances of \(\text{IA\_Set}\) may be value or point sets are support typical unary, binary, and relational set operations. Instances of \(\text{IA\_Image}\) represent an image composed of instances of \(\text{IA\_Point}\) with C++ primitive values. Instances of \(\text{IA\_Image}\) may be constructed directly from point/value combinations or from instances of the class \(\text{IA\_Pixel}<P,T>\) that map points of type \(P\) to values of type \(T\). The operations on \(\text{IA\_Image}\) instances include arithmetic binary, logical binary, characteristic binary,

\(^{17}\) A thorough treatment of the class library is found in Ritter and Wilson’s *Handbook of Computer Vision Algorithms in Image Algebra* [RW96].
Unary, domain transformational, and relational operations. Template instances of

`IA/DDTemplate` allow one to convolve a template over an `IA/Image` instance. Lastly

`IA/Neighborhood` instances provide image-neighborhood operations on `IA/Image`

instances.

9.2 Design Goals

The main design goal of the IAPL was to provide a target language that people
familiar with the Image Algebra could use to prototype image algorithms without
knowing the existing generic programming language implementations in C++ and Ada.

This goal affected the grammar, environment, and type design of the language.

9.2.1 Grammar and Language Design

Algorithms written in the Image Algebra are typically imperative in nature, so
IAL should present a similar programming paradigm. Additionally, the Image Algebra
contains many operators that are not commonly found in modern programming languages
such as characteristic relations and convolutions. To allow IAPL programs to truly
mimic Image Algebra algorithms a new character set should be developed containing
these operators. Lastly, regularly used entities, such as points and sets, should be
constructed using a unique syntax for each entity. Providing these constructs along with
very high-level image operations will programs more readable and will provide another
layer of abstractness for the user.

9.2.2 Environment and type design

Image Algebra algorithms are typically between ten and twenty, so the
algorithmic complexity lies in the underlying operations, rather than the use of these
operations. Semantic debugging should, therefore, not play a large roll in algorithm
prototyping, and the evaluation environment need not restrict the user be requiring typed
variable declarations and other requisites that are needed in large-scale development.

Because very little restriction is placed on variable declaration and assignment,
the IAPL should appear nearly type-free to the user. The underlying type system will, of
course, be strongly typed due the nature of the C++ language, but the user should not be
burdened with any type restrictions. In order to accomplish this, the dispatcher system
should be as tolerant as possible to type-mismatch operations and make every effort to
coerce value types to correct for user errors.

9.3 IAL Implementation

The specification grammar for the IAPL are both found in the appendix. The
operators used exhibited typical precedence and associativity and were composed of
ASCII characters and meta-characters, where a font was developed so that each meta-
character represented an Image Algebra operator.

One editable terminal was defined, and this represented identifies, strings, and
numbers. This node contained an underlying Kstring\textsuperscript{18} kname and its return type
depended on the current value of kname. A single-quoted kname represents a string
scalar, so a KStringValue instance would be returned upon evaluation. The member
kname containing an identifier represents a binding in the current environment, so this
node would attempt to return that binding in the current environment consisting of a
primitive (i.e. number, point, etc.) or an instance of Closure. When kname contained a
numeric string it would either return an intValue, if possible, or a doubleValue.
Furthermore, typical imperative constructs such as for and while loops are included, as
well as syntactic sugar to construct points, pixels, and arrays.
The `.spec` and `.tab` files for the dispatch system are found in the appendix. These files were designed to minimize the total number of functions by performing coercion wherever possible. Additionally, every effort was made to try to recover from type mismatch operations rather than simply issuing a warning message. There are eight primitive types included in the final language design, and the following lists these along with the underlying C++ or iac++ types:

- **Numeric scalar** – `int`, `double`, `float`, and `unsigned char`
- **String scalar** – `KString`
- **Point** – `IA_Point<T>` where `T ∈ {int, double}`
- **Value sets**\(^{19}\) – `IA_Set<T>` where `T ∈ {unsigned char, float}`
- **Images** – `IA_Image<IA_Point<int>, T>` where `T ∈ {bool, unsigned char, float}`
- **Templates** – `IA_DDTemplate<IA_Image<IA_Point<int>, T>>` where `T ∈ {int, bool, unsigned char, float}`
- **Neighborhoods** – `IA_Neighborhood<IA_Point<int>, IA_Point<int>>`
- **Pixels** – `IA_Pixel<IA_Point<int>, int>`

Even though these types are apparent to the programmer, they are transparent to the user. The resulting implementation was both easy-to-use and efficient, and images from the editor are found in the appendix [TODO: Put these images in the appendix].

---

\(^{18}\) Y. Kanev developed the class KString.

\(^{19}\) Point sets were omitted because the current iac++ version did not fully implement these.
APPENDIX
Dispatch System

A1. String conversion within the dispatch system.

<table>
<thead>
<tr>
<th>String</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^</code></td>
<td>&quot;plus&quot;</td>
</tr>
<tr>
<td><code>_</code></td>
<td>&quot;sub&quot;</td>
</tr>
<tr>
<td><code>*</code></td>
<td>&quot;mult&quot;</td>
</tr>
<tr>
<td><code>/</code></td>
<td>&quot;div&quot;</td>
</tr>
<tr>
<td><code>^</code></td>
<td>&quot;pipe&quot;</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>&quot;and&quot;</td>
</tr>
<tr>
<td><code>^</code></td>
<td>&quot;caret&quot;</td>
</tr>
<tr>
<td><code>!</code></td>
<td>&quot;bang&quot;</td>
</tr>
<tr>
<td><code>*</code></td>
<td>&quot;mult&quot;</td>
</tr>
<tr>
<td><code>(</code></td>
<td>&quot;lparen&quot;</td>
</tr>
<tr>
<td><code>)</code></td>
<td>&quot;rparen&quot;</td>
</tr>
<tr>
<td><code>[</code></td>
<td>&quot;lbracket&quot;</td>
</tr>
<tr>
<td><code>]</code></td>
<td>&quot;rbracket&quot;</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>&quot;lt&quot;</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>&quot;le&quot;</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>&quot;gt&quot;</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>&quot;ge&quot;</td>
</tr>
<tr>
<td><code>=</code></td>
<td>&quot;eq&quot;</td>
</tr>
<tr>
<td><code>==</code></td>
<td>&quot;et&quot;</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>&quot;ne&quot;</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>&quot;ltlt&quot;</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>&quot;gtgt&quot;</td>
</tr>
<tr>
<td><code>%</code></td>
<td>&quot;mod&quot;</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>&quot;andand&quot;</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>i</code></td>
<td>&quot;char_161&quot;</td>
</tr>
<tr>
<td><code>e</code></td>
<td>&quot;char_162&quot;</td>
</tr>
<tr>
<td><code>ê</code></td>
<td>&quot;char_163&quot;</td>
</tr>
<tr>
<td><code>ô</code></td>
<td>&quot;char_164&quot;</td>
</tr>
<tr>
<td><code>ö</code></td>
<td>&quot;char_165&quot;</td>
</tr>
<tr>
<td><code>!</code></td>
<td>&quot;char_166&quot;</td>
</tr>
<tr>
<td><code>Ç</code></td>
<td>&quot;char_199&quot;</td>
</tr>
<tr>
<td><code>Æ</code></td>
<td>&quot;char_199&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_191&quot;</td>
</tr>
<tr>
<td><code>À</code></td>
<td>&quot;char_192&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_193&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_194&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_195&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_196&quot;</td>
</tr>
<tr>
<td><code>Å</code></td>
<td>&quot;char_197&quot;</td>
</tr>
<tr>
<td><code>Æ</code></td>
<td>&quot;char_198&quot;</td>
</tr>
<tr>
<td><code>Ç</code></td>
<td>&quot;char_200&quot;</td>
</tr>
</tbody>
</table>
A2. Dispatch entry structs

```c
struct BinopEntry {
    Value *(*conv1)(Value *);
    Value *(*conv2)(Value *);
    Value *(*op)(Value *, Value *);
};

struct UnopEntry {
    Value *(*conv1)(Value *);
    Value *(*op)(Value *);
};

struct SparseTrinopEntry {
    int type1;
    int type2;
    int type3;
    Value *(*conv1)(Value *);
    Value *(*conv2)(Value *);
    Value *(*conv3)(Value *);
    Value *(*op)(Value *, Value *, Value *);
};

struct SparseBinopEntry {
    int type1;
    int type2;
    Value *(*conv1)(Value *);
    Value *(*conv2)(Value *);
    Value *(*op)(Value *, Value *);
};

struct SparseUnopEntry {
    int type1;
    Value *(*conv1)(Value *);
    Value *(*op)(Value *);
};
```

A3. The .spec file grammar

```regex
spec ::= intro '%%' rule*

intro ::= intro abbreviation ':' name
    | ε

rule ::= b_rule | u_rule | m2_rule | m2_rule | comment

b_rule ::= '%%binary' name+ ':' name name name

u_rule ::= '%%unary' name+ ':' name name
```
\texttt{m2\_rule ::= \textquoteleft \%member\textquoteleft name+ \textquoteleft:\textquoteleft name name name;}

\texttt{u\_rule ::= \textquoteleft \%member\textquoteleft name+ \textquoteleft:\textquoteleft name name;}

\texttt{comment ::= \textquoteleft \#\textquoteleft \.*

\texttt{name ::= \textquoteleft \*\*\*[a-zA-Z_]\+}

A4. An example .\texttt{spec} file for arithmetic over ints, reals, and comps called \texttt{example.spec}.

\begin{verbatim}
# There are 3 types:   int, real, comp
# There are 2 operations:  +, *

int:int
real:real
comp:comp
%
# int
%binary \textquoteleft+\textquoteleft \textquoteleft\*\textquoteleft: int int int
%unary \textquoteleft-\textquoteleft : int int
# real
%binary \textquoteleft+\textquoteleft \textquoteleft\*\textquoteleft: real real real
%unary \textquoteleft-\textquoteleft : real real
# comp
%binary \textquoteleft+\textquoteleft \textquoteleft\*\textquoteleft: comp comp comp
%unary \textquoteleft-\textquoteleft : comp comp
\end{verbatim}

A5. The header \texttt{Functions.cpp generated from example.spec}.

/ Emacs -*- C++ -*-

//
// Generated by jpcalm @ eclipse on Fri Mar 17 22:24:30 2000.
// Don't mess with it!
//

#define V_int  Value::int
#define V_real Value::real
#define V_comp Value::comp

Value *int_to_comp(Value *v);
Value *real_to_comp(Value *v);

/*@ Begin Functions */
class Value;

/*************** T H R O W   U P ***************
Value *no(Value *);
Value *no(Value *, Value *);
Value *no(Value *, Value *, Value *);
/*************** C O N V E R S I O N S ***************/

Value *id(Value *);

///////// Headers //////////
Value *plus_int_int(Value *v1, Value *v2);
Value *mult_int_int(Value *v1, Value *v2);
Value *sub_int(Value *v1);
Value *plus_real_real(Value *v1, Value *v2);
Value *mult_real_real(Value *v1, Value *v2);
Value *sub_real(Value *v1);
Value *plus_comp_comp(Value *v1, Value *v2);
Value *mult_comp_comp(Value *v1, Value *v2);
Value *sub_comp(Value *v1);

A6. The source Functions.cpp generated from example.spec.

// Emacs -*- C++ -*-
//
// Generated by jpalm @ eclipse on Fri Mar 17 22:24:30 2000.
// Don't mess with it!
//
#include "Functions.h"

Value *int_to_comp(Value *v)
{ return new compValue((comp) v->getint()); }

Value *real_to_comp(Value *v)
{ return new compValue((comp) v->getreal()); }

_MAXIMUM Functions //////////
Value *plus_int_int(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: int + int" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_int_int(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: int * int" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_int(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - int" << endl;
#endif
    return new (- v1->); }

Value *plus_real_real(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: real + real" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_real_real(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: real * real" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_real(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - real" << endl;
#endif
    return new (- v1->); }

Value *plus_comp_comp(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: comp + comp" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_comp_comp(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: comp * comp" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_comp(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - comp" << endl;
#endif
    return new (- v1->); }

# include "Functions.h"

Value *int_to_comp(Value *v)
{ return new compValue((comp) v->getint()); }

Value *real_to_comp(Value *v)
{ return new compValue((comp) v->getreal()); }

_MAXIMUM Functions //////////
Value *plus_int_int(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: int + int" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_int_int(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: int * int" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_int(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - int" << endl;
#endif
    return new (- v1->); }

Value *plus_real_real(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: real + real" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_real_real(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: real * real" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_real(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - real" << endl;
#endif
    return new (- v1->); }

Value *plus_comp_comp(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: comp + comp" << endl;
#endif
    return new (v1-> + v2->); }

Value *mult_comp_comp(Value *v1, Value *v2) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: comp * comp" << endl;
#endif
    return new (v1-> * v2->); }

Value *sub_comp(Value *v1) {
#ifdef VERBOSE
    cout << __FILE__ << "::" << __LINE__ << ":: - comp" << endl;
#endif
    return new (- v1->); }

A6. The source Functions.cpp generated from example.spec.
Value *plus_real_real(Value *v1, Value *v2) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": real + real" << endl;
    #endif
    return new realValue(v1->getreal() + v2->getreal()); }

Value *mult_real_real(Value *v1, Value *v2) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": real * real" << endl;
    #endif
    return new realValue(v1->getreal() * v2->getreal()); }

Value *sub_real(Value *v1) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": - real" << endl;
    #endif
    return new realValue(- v1->getreal()); }

Value *plus_comp_comp(Value *v1, Value *v2) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": comp + comp" << endl;
    #endif
    return new compValue(v1->getcomp() + v2->getcomp()); }

Value *mult_comp_comp(Value *v1, Value *v2) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": comp * comp" << endl;
    #endif
    return new compValue(v1->getcomp() * v2->getcomp()); }

Value *sub_comp(Value *v1) {
    #ifdef VERBOSE
        cout << __FILE__ << "::" << __LINE__ << ": - comp" << endl;
    #endif
    return new compValue(- v1->getcomp()); }

A7. The .tab file grammar.

```
tab ::= "\%default_dim" integer dec_cnt*;
dec_cnt ::= [comment]* dec;
comment ::= "\#".*;
dec ::= binary_dec | unary_dec | ternary_dec;
ternary_dec ::= "\%sparse_trinop" op+ ternary_lines \%end;
```
$\text{binary\_dec} ::= \text{%binop} \ \text{op} + [\text{integer} \ \text{integer}]^* \ \text{binary\_lines} \ \text{%end}$

$\text{unary\_dec} ::= \text{%unop} \ \text{op} + [\text{integer}]^* \ \text{unary\_lines} \ \text{%end}$

$\text{ternary\_lines} ::= [\text{type} \ \text{type} \ \text{type} \ \text{type} \ \text{type} \ \text{type}]^*$

$\text{binary\_lines} ::= [\text{type} \ \text{type} \ \text{type} \ \text{type} \ \text{type} \ \text{type}]^*$

$\text{unary\_lines} ::= [\text{type} \ \text{type} \ \text{type}]^*$

$\text{op} ::= \text{%valid\_op} \ \text{valid\_op}^20,^*$

$\text{type} ::= \text{!}\text{[a-zA-Z]}^+$

$\text{function} ::= [\text{a-zA-Z}]^+$

A8. An example .tab file called example.tab.

%default_dim 3
%binop '+' 'div
int int !comp !comp
int real !comp !comp *
int comp !comp id *
real real !comp !comp
real comp !comp id *
comp comp id id
%end

A9. Output listing from example.tab.

#
# test.spec.out
# Fri Mar 17 22:04:04 2000
# by jpalm on eclipse
#
.Dense Binary 'plus' [03][03]: 006 / 009 = 66.67% used.
.Dense Binary 'div' [03][03]: 006 / 009 = 66.67% used.

A10. Maker header Maker.h generated from example.tab.

//
// Automatically generated file, don't mess with it!
//

\text{20} \text{Please note, valid\_op is any operator found in the conversion chart in A1.}
#ifndef MAKER_H
#define MAKER_H 1

#include "BinopDispatcher.h"
#include "UnopDispatcher.h"
#include "SparseTrinopDispatcher.h"
#include "SparseBinopDispatcher.h"
#include "SparseUnopDispatcher.h"

BinopDispatcher *makeplus();
BinopDispatcher *makediv();

#endif

A11. Source file Maker.cpp generated from example.tab.

//
// Automatically generated file, don't mess with it!
//

#include "Maker.h"
#include "Functions.h"

BinopDispatcher *makeplus() {
  #ifdef VERBOSE
  cout << "Building the binary 'plus' dispatcher..." << endl;
  #endif
  BinopEntry **t = new BinopEntry*[3];
  for (int i = 0; i < 3; i++) {
    t[i] = new BinopEntry[3];
  }
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      t[i][j].conv1 = t[i][j].conv2 = no; t[i][j].op = no;
    }
  }
  t[V_intint!comp!comp].conv1 = id;
  t[V_intint!comp!comp].conv2 = ;
  t[V_intint!comp!comp].op = plus__;
  t[V_intreal!comp!comp*].conv1 = id;
  t[V_intreal!comp!comp*].conv2 = ;
  t[V_intreal!comp!comp*].op = plus__;
  t[V_intcomp!compid*].conv1 = id;
  t[V_intcomp!compid*].conv2 = ;
  t[V_intcomp!compid*].op = plus__;
  t[V_intcomp!compid*].conv1 = id;
  t[V_intcomp!compid*].conv2 = ;
  t[V_intcomp!compid*].op = plus__;
  t[V_realreal!comp!comp].conv1 = id;
  t[V_realreal!comp!comp].conv2 = ;
  t[V_realreal!comp!comp].op = plus__;
  t[V_realreal!comp!comp].conv1 = id;
  t[V_realreal!comp!comp].conv2 = ;
  t[V_realreal!comp!comp].op = plus__;
  t[V_realcomp!compid*].conv1 = id;
  t[V_realcomp!compid*].conv2 = ;
  t[V_realcomp!compid*].op = plus__;
  t[V_realcomp!compid*].conv1 = id;
  t[V_realcomp!compid*].conv2 = ;
  t[V_realcomp!compid*].op = plus__;

  return new BinopDispatcher(t, 3, 3, "+");
}
BinopDispatcher *makediv() {
#ifdef VERBOSE
    cout << "Building the binary 'div' dispatcher..." << endl;
#else
    BinopEntry **t = new BinopEntry*[3];
    for (int i = 0; i < 3; i++) {
        t[i] = new BinopEntry[3];
    }
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            t[i][j].conv1 = t[i][j].conv2 = no; t[i][j].op = no;
        }
    }
    t[V_intint!comp!comp].conv1 = id;
    t[V_intint!comp!comp].conv2 = ;
    t[V_intint!comp!comp].op = div__;
    t[V_intreal!comp!comp*].conv1 = id;
    t[V_intreal!comp!comp*].conv2 = ;
    t[V_intreal!comp!comp*].op = div__;
    t[V_intcomp!compid*].conv1 = id;
    t[V_intcomp!compid*].conv2 = ;
    t[V_intcomp!compid*].op = div__;
    t[V_realreal!comp!comp].conv1 = id;
    t[V_realreal!comp!comp].conv2 = ;
    t[V_realreal!comp!comp].op = div__;
    t[V_realcomp!compid*].conv1 = id;
    t[V_realcomp!compid*].conv2 = ;
    t[V_realcomp!compid*].op = div__;
    t[V_compcompidid].conv1 = id;
    t[V_compcompidid].conv2 = ;
    t[V_compcompidid].op = div__;
return new BinopDispatcher(t, 3, 3, "/");
}

A12. Example original .gram file called spec.gram.

{%
DIGIT [0-9]
DELIM [ \t\n]
%%
}{DELIM}
%

%prefix STD
%term EQ '=
%left EQ
%term LT '<'
%term GT '>'
%term LPAREN '('
%term RPAREN ')'
%term COMMA ','
%term SEMI ';' 'NULL' 'n'
%term PRINT 'print'
%term INT '0'  
%termpat INT '{DIGIT}+'  

program: expressions  
  ;

expressions: expression SEMI expressions  
  | expression  
    ;

expression: expression PLUS  expression  
  | expression MULT  expression  
  | expression BINOP expression  
  | PRINT LPAREN expression RPAREN  
  | INT  
  | LT INT COMMA INT GT  
    ;

A13. New .spec file called calc.gram.

{%
DIGIT [0-9]  
DE delim [ \t\n]  
%}

// Extra stuff could go into "extraStuff.h" from here
#include "HarryLegsException.h"

void j_throw(char *msg, char *file, int line);

%

void j_throw(char *msg, char *file, int line) 
  {
    cout << "Oops..." << endl;
    throw HarryLegsException(msg, file, line);
  }

%

%prefix CALC
%includedir /cise/homes/jpalm/research/mypoint/include
%includedir /cise/homes/jpalm/research/kstring/include
#include <iostream.h>
#include "kstring"
#include "MyPoint.h"

%value int
%value MyPoint

%opgroup BINOP        '?'
%term EQ '='
%left EQ
%term LT ' '<'  
%term GT ' '>'  
%term LPAREN '('  
%term RPAREN ')'  
%term COMMA ','  
%term SEMI ':' 'NULL' 'n'  
%term PRINT 'print'  

%op BINOP PLUS '+'  
%left PLUS  

%op BINOP MULT '*'  
%left MULT  

%left BINOP  

/* We can have comments now, too! */  
%eterm INT '0'  
%etermpat int INT '{DIGIT}+' {:
  return i;
:}
%private {:
  int i; /*0*/
:}
%asstring {:
  char str[100];
  sprintf(str, "%d", i);
  return strdup(str);
:}
%setval {:
  char *bp;
  i = (int) strtod(val, &bp);
:}

program: expressions
{::
  env = new Env();
  @1;
:}

expressions:
  expression SEMI expressions {: @1; @3; :}
| {:
  /* nothing */
:}

Value*
expression:
  %hidden expression PLUS expression
{::
  return Dispatcher::plus(@1, @3);
:}
| %hidden expression MULT expression
{:
return Dispatcher::plus(@1, @3);
;
| expression BINOP expression
| : 
| KString error = "You have to fill the '?' in!";
| j_throw((char *) error, __FILE__, __LINE__);
| ;
| PRINT LPAREN expression RPAREN
| :
| Dispatcher::print(@3);
| ;
| INT
| :
| return new intValue(@1);
| ;
| LT INT COMMA INT GT
| :
| return new MyPointValue(MyPoint(@2, @4));
| ;

A14. Actions file grammar

```
actions_grammer ::= lex includes includedirs rhs nonterms eterms operators 
:

lex ::= '--- begin lex ---'
LEX21
'--- end lex ---'
;

includes ::= '--- begin includes ---'
INCLUDES
'--- end includes ---'
;

includedirs ::= '--- begin includedirs ---'
INCLUDEDIRS
'--- end includedirs ---'
;

rhs ::= '***** begin rhs *****'
rhs_entry*
'***** end rhs *****'
;

rhs_entry ::= '--- begin 'CLASS ' ---'
RETURN_TYPE
'<begin children>'
CHILDREN
'<end children>'
```

---All **CAPITILIZED ITALICIZED WORDS** in this grammar denote a value discussed earlier found in the specification grammar."
nonterms := "***** begin nonterms *****
nonterm_entry*
"***** end nonterms *****"
;

nonterm_entry ::= '--- begin 'CLASS' ---'
RETURN_TYPE
'begin rhs'
RHS
'end rhs'
'--- end 'CLASS' ---'
;

terms ::= "***** begin terms *****
term_entry*
"***** end terms *****"
;

term_entry ::= '--- begin 'CLASS' ---'
'void'
'begin formatting,'
FORMATTING
PRE_FORMATTING
POST_FORMATTING
'end formatting,'
'--- end 'CLASS' ---'
;

eterms ::= "***** begin eterms *****
eterm_entry*
"***** end eterms *****"
;

eterm_entry ::= '--- begin 'class' ---'
RETURN_TYPE
'begin_action'
ACTION
'end action'
'begin variables'
VARIABLES
'end variables'
'begin asString'
AS_STRING
'end asString'
'begin setVal'
SET_VAL
'end setVal'
'begin pattern'
operators ::= '***** begin operators *****'
   operator_entry*
   '***** end operators *****'
;
operator_entry ::= '--- begin 'CLASS '---'
   CLASS*
   '--- end 'CLASS '---'
;

A15. Grammar for the specification grammar.

spec_gram ::= preamble visitors terms grammar
;
preamble ::= '%%{
   LEX `%'
   EXTRA_STUFF_H `%%'
   EXTRA_STUFF_CPP `%%'
   %PREFIX'
   id
}
visitors ::= '%visitor' id visitors
  | ε
;
terms ::= terms defterm
  | terms defeterm
  | terms etermpat
  | terms makestuff
  | terms precedence
  | ε
;
makestuff ::= '%includedir' id
  | '%include' id
  | '%value' id
;
defterm ::= '%term' id term
  | '%term' id term term
  | '%term' id term term term
  | '%opgroup' id term
  | '%op' id id term
;
defeterm ::= '\%emterm' id  
      | \%emterm' id term term  
      | \%emterm' id term term term  
      ;
etermpat ::= '\%etermpat' id term '{: ACTION::}'  
         \%private '{: ACTION::}'  
         \%asstring '{: ACTION::}'  
         \%setval '{: ACTION::}'  
      ;
precedence ::= prec list  
      ;
prec ::= \%right  
      | \%left  
      ;
list ::= list ID  
      | \epsilon  
      ;
hidden ::= \%hidden  
      | \epsilon  
      ;
grammar ::= rule  
      ;
rule ::= id ': rightHandSide ;'  
      | id ': rightHandSide ;' rule  
      | id id ': rightHandSide ;'  
      | id id ': rightHandSide ;' rule  
      ;
rightHandSide ::= hidden idList '{: ACTION::}'  
      | hidden idList '{: ACTION::}' rightHandSide  
      | hidden '{: ACTION::}'  
      | hidden '{: ACTION::}' rightHandSide  
      ;
idList ::= id  
      | id idList  
      | term  
      | term idList  
      ;
id ::= [/_a-zA-Z.\"<>~][_a-zA-Z0-9-<>\"\'*\'\'/+.,]*  
      ;
term ::= \\[^']\+\'  
      ;
A16. Specification grammar for the IAPL

{%
DIGIT [0-9]
DELIM [\t\n]
IDENT ['a-zA-Z0-9\_].
OPERATOR [;!\*\+\-\^\&~\|\<\>=\[\&\|=]*
%%
{DELIM}
%
#include "HarryLegsException.h"

void j_throw(char *msg, char *file, int line);

%
void j_throw(char *msg, char *file, int line)
{
    cout << "Oops..." << endl;
    throw HarryLegsException(msg, file, line);
}
%
%prefix HIA

%includedir /cise/homes/jpalm/research/ia++/include
%includedir /cise/homes/jpalm/research/tab/include
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include "ia/Bit.h"
#include "ia/RGB.h"
#include "ia/Db1Point.h"
#include "ia/IntPoint.h"
#include "ia/Set.h"
#include "ia/IntPS.h"
#include "ia/Db1PS.h"
#include "ia/IntDI.h"
#include "ia/BoolDI.h"
#include "ia/UcharDI.h"
#include "ia/FloatDI.h"
#include "ia/CplxDI.h"
#include "ia/RGBDI.h"
#include "ia/IntProd.h"
#include "ia/BoolProd.h"
%include "ia/UcharProd.h"
%include "ia/FloatProd.h"
%include "ia/CplxProd.h"
%include "ia/BoolNOps.h"
%include "ia/UcharNOps.h"
%include "ia/IntNOps.h"
%include "ia/FloatNOps.h"
%include "ia/CplxNOps.h"
%include "ia/Pixel.h"
%include "Dispatcher.h"
%include "kstring.h"

%value int
%value IA_Point<int>*
%value bool
%value IA_Set<int>*
%value IA_Set<IA_Point<int>>*
%value char*
%value IA_Image<IA_Point<int>,int>*
%value double
%value float
%value u_char
%value IA_Set<u_char>*
%value IA_Set<float>*
%value IA_Image<IA_Point<int>,bool>*
%value IA_Image<IA_Point<int>,u_char>*
%value IA_Image<IA_Point<int>,float>*
%value IA_DDTemplate<IA_Image<IA_Point<int>,int>>*
%value IA_DDTemplate<IA_Image<IA_Point<int>,bool>>*
%value IA_DDTemplate<IA_Image<IA_Point<int>,u_char>>*
%value IA_DDTemplate<IA_Image<IA_Point<int>,float>>*
%value IA_Neighborhood<IA_Point<int>,IA_Point<int>>*
%value IA_Pixel<IA_Point<int>,int>*
%value KString

%term EQ '='
%left EQ

%term LT '<'
%term GT '>'

%term CEILLEFT '£'
%term CEILRIGHT '¥'
%term FLOORLEFT '¤'
%term FLOORRIGHT '¦'
%term PIPE '|'  
%term SUB_DEC 'sub'

%term LPAREN '('
%term RPAREN ')'
%term LBRACE '{'
%term RBRACE '}'
%term LBRACKET '['
%term RBRACKET ']'
%term IF 'if'
%term ELSE 'else'
%term TRUE 'true'
%term FALSE 'false'
%term COMMA ','
%term RETURN 'return'
%term SEMI ';'
NULL 'n'

%opgroup BINOP '??'
%opgroup UNOP '?'

%op BINOP _LT '<'
%op BINOP _LE '<='
%op BINOP _GT '>
%op BINOP _GE '>='
%op BINOP _NE '!='
%left _LT _LE _GT _GE _NE

%op BINOP CHI_EQ '='
%op BINOP CHI_NE '<'
%op BINOP CHI_GT '>'
%op BINOP CHI_LT '<'
%op BINOP CHI_GE '>
%op BINOP CHI_LE '<'
%left CHI_EQ CHI_NE CHI_GT CHI_LT CHI_GE CHI_LE

%op BINOP ANDAND '&&'
%op BINOP PIPEPIPE '||'
%left ANDAND PIPEPIPE

%op BINOP MAX '
%op BINOP MIN '
%left MAX MIN

%op BINOP PLUS '+'
%op BINOP SUB '-'
%left PLUS SUB

%op BINOP DIV '/'
%op BINOP MULT 'x'
%op BINOP MOD '%'
%op BINOP LPROD '©'
%op BINOP ADDMAX_PROD '¢'
%op BINOP ADDMIN_PROD '¡'
%op BINOP MULTMAX_PROD '§'
%op BINOP MULTMIN_PROD '¨'
%left DIV MULT MOD LPROD ADDMAX_PROD ADDMIN_PROD MULTMAX_PROD MULTMIN_PROD

%op BINOP GTGT '>>'
%op BINOP LTLT '<<'
%left GTGT LTLT

%op BINOP CARET '^'
%op BINOP AND ' &'
%left CARET AND
%op UNOP BANG '!' %op UNOP UMAX '¯'
%op UNOP UMIN '@'
%op UNOP USUM '¬'
%op UNOP UPROD '-'
%op UNOP TILDE '~'
%left BANG UMAX UMIN USUM UPROD TILDE

%left BINOP %left UNOP

%eterm string 'string'
%etermpat Value* string '{IDENT}+ ' {:
if (kname[0] == '\'') {
    return new KStringValue(
        kname.Last(kname.GetLength()-1).First(kname.GetLength()-2));
} try {
    return (IdentifierNotFoundException e) {
        if (isdigit(kname[0]) ||
            (strlen((char *) kname) > 1 && isdigit(kname[1])))
            return value;
        KString error = "";
        error << kname;
        error << "'" was not found in the environment: ";
        error << e.asString();
        j_throw((char *)error, __FILE__, __LINE__);
    };
%private {:
    Value *value; /***/
    KString kname; /**"string"*/
:}
%asstring {:
    return (char *) kname;
:}
%setval {:
    if (isdigit(kname[0]) ||
        (strlen((char *) kname) > 1 && isdigit(kname[1])))
    {
        if (val) {
            //cerr << "Digit, OK" "endl;
            char *bp;
            double d = strtod(val, &bp);
            kname = strdup(val);
            if (strcmp(bp, "") {  
                value = new intValue((int) d);
            } else if (floor(d) == d) {
                value = new intValue((int) d);
            } else {
                value = new doubleValue(d);
            }
        } else {
            kname = "";
        }
} else {
    if (val) { /* cerr << "String OK" << endl; */
        kname = strdup(val);
    } else { /* cerr << "No String" << endl; */
        kname = "";
    }
}

program:
    statements
    {
        cout << "\n\n------------------------------" << endl;
        env = new Env();
        Dispatcher::populate(env);
        @1;
        cout << "------------------------------" << endl;
    }

function:
    SUB_DEC string LPAREN params RPAREN LBRACE function_body RBRACE
    {
        Closure *c = new Closure($7);
        in_env->insert($2->asString(), c);
        env = new Env(in_env);
        env->insert($2->asString(), c);
        c->setParams($4);
        c->setEnv(env);
        Param *trav = c->getParams();
        while (!(trav->isEnd())) {
            c->getEnv()->writeOverType(trav->getName(),
                                        trav->getValue());
            trav = trav->getLink();
        }
    }

Value*
function_body:
    statements RETURN expression SEMI {
    @1; return @3; :
    }

statement:
    expression {: @1; :}
    | conditional {: @1; :}
    | repetition {: @1; :}
    ;

Value*
args:
    expression COMMA args
    {:
        Value *v = @1;
        v->setLink(@3);
        return v;
    }
| expression  
{   
Value *v = @1;  
v->setLink(new EndValue());  
return v;  
:}  
;  

Value*  
array:  
expression COMMA array  
{:   
Value *v = @1;  
v->setArrayLink(@3);  
return v;  
:}  
| expression  
{:   
Value *v = @1;  
v->setArrayLink(new EndValue());  
return v;  
:}  
;  

Param*  
params:  
string COMMA params  
{:   
Param *p = new Param($1->asString(), new intValue(0));  
p->setLink(@3);  
return p;  
:}  
| string  
{:   
Param *p = new Param($1->asString(), new intValue(0));  
p->setLink(new EndParam());  
return p;  
:}  
;  

statements:  
statement SEMI statements    {: @1; @3;    :}  
| function  SEMI statements    {: @1; @3;    :}  
|                     {: /* empty */    :}  
;  

block:  
LBRACE statements RBRACE    {: @2;    :}  
;  

Value*  
repetition:  
WHILE LPAREN expression RPAREN block  
{:   
    int i = 0;  
env = new Env(in_env);  
while (@3->getbool()) { @5; i++; }  
}
return new intValue(i);
:
| FOR LPAREN expression COMMA expression COMMA expression RPAREN block |
| {:
|   int i = 0;
|   env = new Env(in_env);
|   @3;
|   while (@5->getbool()) { @9; @7; i++; } |
| return new intValue(i);
| :}

Value*
conditional:
| IF LPAREN expression RPAREN block |
| {:
|   bool b = @3->getbool(); |
|   if (b) @5;
|   return new boolValue(b);
| :}

| IF LPAREN expression RPAREN block ELSE block |
| {:
|   bool b = @3->getbool(); |
|   if (b) @5; else @7;
|   return new boolValue(b);
| :}

Value*
assignment:
| string EQ expression |
| {:
|   Value *v = @3;
|   try {
|     env->insert($1->asString(), v);
|   } catch (DuplicateIdentifierException e) {
|     env->assign($1->asString(), v);
|   }
| return v;
| :}

Value*
call:
| string LPAREN args RPAREN |
| {:
|   KString name = $1->asString();
|   Value *v = @1;
|   if (!v->isClosure()) { |
|     KString error = "";
|     error << name;
|     error << "' is already a variable, not a function";
|     j_throw((char *) error, __FILE__, __LINE__);
|   }
|   Closure *c = (Closure *) v;
|   return c->evaluate(@3);
| :}
Value
expression:

%hidden expression _LT expression
{: return Dispatcher::lt (01, 03); :}

%hidden expression _LE expression
{: return Dispatcher::le (01, 03); :}

%hidden expression _GT expression
{: return Dispatcher::gt (01, 03); :}

%hidden expression _GE expression
{: return Dispatcher::ge (01, 03); :}

%hidden expression _NE expression
{: return Dispatcher::ne (01, 03); :}

%hidden expression CHI_EQ expression
{: return Dispatcher::chi_eq (01, 03); :}

%hidden expression CHI_NE expression
{: return Dispatcher::chi_ne (01, 03); :}

%hidden expression CHI_GT expression
{: return Dispatcher::chi_gt (01, 03); :}

%hidden expression CHI_LT expression
{: return Dispatcher::chi_lt (01, 03); :}

%hidden expression CHI_GE expression
{: return Dispatcher::chi_ge (01, 03); :}

%hidden expression CHI_LE expression
{: return Dispatcher::chi_le (01, 03); :}

%hidden expression ANDAND expression
{: return Dispatcher::andand (01, 03); :}

%hidden expression PIPEPIPE expression
{: return Dispatcher::andand (01, 03); :}

%hidden expression MAX expression
{: return Dispatcher::max (01, 03); :}

%hidden expression MIN expression
{: return Dispatcher::min (01, 03); :}

%hidden expression PLUS expression
{: return Dispatcher::plus (01, 03); :}

%hidden expression SUB expression
{: return Dispatcher::sub (01, 03); :}

%hidden expression DIV expression
{: return Dispatcher::div (01, 03); :}

%hidden expression MULT expression
{: return Dispatcher::mult (01, 03); :}

%hidden expression LPROD expression
{: return Dispatcher::linear_product (01, 03); :}

%hidden expression ADDMAX_PROD expression
{: return Dispatcher::addmax_product (01, 03); :}

%hidden expression ADDMIN_PROD expression
{: return Dispatcher::addmin_product (01, 03); :}

%hidden expression MULTMAX_PROD expression
{: return Dispatcher::multmax_product (01, 03); :}

%hidden expression MULTMIN_PROD expression
{: return Dispatcher::multmin_product (01, 03); :}

%hidden expression GTGT expression
{: return Dispatcher::gtgt (01, 03); :}

%hidden expression LTLT expression
{: return Dispatcher::ltlt (01, 03); :}

%hidden expression CARET expression

%default_dim 22

{ return Dispatcher::caret (@1, @3); }
| %hidden expression AND expression
  { return Dispatcher::and (@1, @3); }
| expression BINOP expression
  {:
    KString error = "You have to fill the '??' in!";
    j_throw((char *) error, __FILE__, __LINE__);
  :}
| %hidden BANG expression { return Dispatcher::ubang (@2); }
| %hidden UMAX expression { return Dispatcher::umax (@2); }
| %hidden UMIN expression { return Dispatcher::umin (@2); }
| %hidden USUM expression { return Dispatcher::usum (@2); }
| %hidden TILDE expression { return Dispatcher::utilde (@2); }
| UNOP expression
  {:
    KString error = "You have to fill the '?' in!";
    j_throw((char *) error, __FILE__, __LINE__);
  :}
| assignment { return @1; }
| call { return @1; }
| primitive { return @1; }
;

Value*
primitive:
  LPAREN expression RPAREN
  { return @2; }
| FLOORLEFT expression FLOORRIGHT
  { return Dispatcher::ufloor(@2); }
| CEILLEFT expression CEILRIGHT
  { return Dispatcher::uceil(@2); }
| PIPE expression PIPE
  { return Dispatcher::uabs(@2); }
| string
  { return @1; }
| boolean
  { return @1; }
| LT expression COMMA expression GT
  { return Dispatcher::make_point(@2, @4); }
| LPAREN expression COMMA expression RPAREN
  { return Dispatcher::make_pixel(@2, @4); }
| LBRACE array RBRACE
  { return @2; }
;

Value*
boolean:
  TRUE { return new boolValue(true); }
| FALSE { return new boolValue(false); }
;
Dense Binops

%binop '+' '-' '*' '/' '×'
int     int     id      id
int     bool    id      !int    *
int     float   !float id    *
int     double  !double id   *
int     IntPoint id      id    *
int     IntDi  id      id    *
int     UcharDI !Uchar id   *
int     FloatDi !float id    *
bool     bool   id      id
bool     Uchar  !Uchar id   *
bool     float  !float id    *
bool     double !double id   *
bool     IntPoint  !int id    *
bool     IntDi  !int id    *
bool     UcharDI !Uchar id   *
bool     FloatDi !float id    *
Uchar    Uchar  id      id
Uchar    float  !float id    *
Uchar    double !double id   *
Uchar    IntPoint !int id    *
Uchar    IntDi  !int id    *
Uchar    UcharDI id      id
Uchar    FloatDi !float id    *
float    float  id      id
float    double !double id   *
float    IntPoint  !int id    *
float    IntDi  !int id    *
float    UcharDI !Uchar id   *
float    FloatDi !float id    *
double   double id      id
double   IntPoint  !int id    *
double   IntDi  !int id    *
double   UcharDI !Uchar id   *
double   FloatDi !float id    *
IntPoint  IntPoint id      id
IntDi    IntDi id      id
IntDi    BoolDi id      !IntDi    *
IntDi    UcharDi id      !IntDi    *
IntDi    FloatDi !FloatDi id    *
BoolDi   UcharDi !UcharDi id *
BoolDi   FloatDi !FloatDi id *
UcharDI  UcharDi id      id
UcharDI  FloatDi !FloatDi id *
FloatDi  FloatDi id      id
%end

%binop '>>'
int     int     id      id
int     bool    id      !int    *
int     Uchar  id      !int    *
int     float  id      !int    *
int     double  !int    *
bool     bool   !int    *
bool     Uchar  !int    *
bool     float  !int    *
bool     double  !int    *
Uchar    Uchar  !Uchar    *
Uchar    double  !Uchar    *
float    float  !int    *
float    double !int    *
double   double !int    *
IntDi    int     id      id
IntDi    bool    id      !int
IntDi    Uchar    id      !int
IntDi    float    id      !int
IntDi    double  !int    

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntDI</td>
<td>IntDI</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>IntDI</td>
<td>BoolDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>IntDI</td>
<td>UcharDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>IntDI</td>
<td>FloatDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>int</td>
<td>!IntDI</td>
<td>id</td>
</tr>
<tr>
<td>BoolDI</td>
<td>bool</td>
<td>!IntDI</td>
<td>int</td>
</tr>
<tr>
<td>BoolDI</td>
<td>Uchar</td>
<td>!UcharDI</td>
<td>id</td>
</tr>
<tr>
<td>BoolDI</td>
<td>float</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>BoolDI</td>
<td>double</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>BoolDI</td>
<td>BoolDI</td>
<td>!IntDI</td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>UcharDI</td>
<td>!UcharDI</td>
<td>!UcharDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>FloatDI</td>
<td>!IntDI</td>
<td>!IntDI</td>
</tr>
<tr>
<td>UcharDI</td>
<td>int</td>
<td>!Uchar</td>
<td>id</td>
</tr>
<tr>
<td>UcharDI</td>
<td>bool</td>
<td>!Uchar</td>
<td>id</td>
</tr>
<tr>
<td>UcharDI</td>
<td>double</td>
<td>!Uchar</td>
<td>!int</td>
</tr>
<tr>
<td>UcharDI</td>
<td>Uchar</td>
<td>!UcharDI</td>
<td>id</td>
</tr>
<tr>
<td>UcharDI</td>
<td>float</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>UcharDI</td>
<td>double</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>UcharDI</td>
<td>FloatDI</td>
<td>!IntDI</td>
<td>!IntDI</td>
</tr>
<tr>
<td>FloatDI</td>
<td>int</td>
<td>!IntDI</td>
<td>id</td>
</tr>
<tr>
<td>FloatDI</td>
<td>bool</td>
<td>!IntDI</td>
<td>int</td>
</tr>
<tr>
<td>FloatDI</td>
<td>Uchar</td>
<td>!UcharDI</td>
<td>id</td>
</tr>
<tr>
<td>FloatDI</td>
<td>float</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>FloatDI</td>
<td>double</td>
<td>!IntDI</td>
<td>!int</td>
</tr>
<tr>
<td>FloatDI</td>
<td>FloatDI</td>
<td>!IntDI</td>
<td>!IntDI</td>
</tr>
</tbody>
</table>

%end

%binop '<<'

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>int</td>
<td>bool</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>int</td>
<td>Uchar</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>bool</td>
<td>Uchar</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>bool</td>
<td>float</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>bool</td>
<td>double</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>Uchar</td>
<td>Uchar</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>Uchar</td>
<td>float</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>Uchar</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>Uchar</td>
<td>float</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>Uchar</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>Double</td>
<td>double</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>int</td>
<td>IntDI</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>int</td>
<td>bool</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>int</td>
<td>Uchar</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>IntDI</td>
<td>int</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>IntDI</td>
<td>bool</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>IntDI</td>
<td>Uchar</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>IntDI</td>
<td>float</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>IntDI</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>IntDI</td>
<td>double</td>
<td></td>
<td>!int</td>
</tr>
<tr>
<td>IntDI</td>
<td>IntDI</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>IntDI</td>
<td>BoolDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>IntDI</td>
<td>UcharDI</td>
<td></td>
<td>!UcharDI</td>
</tr>
<tr>
<td>IntDI</td>
<td>FloatDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>int</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>bool</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>Uchar</td>
<td></td>
<td>!UcharDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>float</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>double</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>BoolDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>UcharDI</td>
<td></td>
<td>!UcharDI</td>
</tr>
<tr>
<td>BoolDI</td>
<td>FloatDI</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>UcharDI</td>
<td>int</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>UcharDI</td>
<td>bool</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>UcharDI</td>
<td>Uchar</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>UcharDI</td>
<td>float</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>UcharDI</td>
<td>double</td>
<td></td>
<td>!Uchar</td>
</tr>
<tr>
<td>UcharDI</td>
<td>UcharDI</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>UcharDI</td>
<td>FloatDI</td>
<td></td>
<td>!UcharDI</td>
</tr>
<tr>
<td>FloatDI</td>
<td>int</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>FloatDI</td>
<td>bool</td>
<td></td>
<td>!IntDI</td>
</tr>
<tr>
<td>FloatDI</td>
<td>Uchar</td>
<td></td>
<td>!UcharDI</td>
</tr>
<tr>
<td>FloatDI</td>
<td>float</td>
<td></td>
<td>!IntDI</td>
</tr>
</tbody>
</table>

%end
<table>
<thead>
<tr>
<th>Type</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>UcharDI</td>
<td>!Uchar</td>
<td>id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>FloatDI</td>
<td>!float</td>
<td>id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uchar</td>
<td>Uchar</td>
<td>id</td>
<td>id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>id</td>
<td></td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%binop <- | <= | >  | >= | == | != | ° | ± | ² | ³ | » |
---

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>double</td>
<td>!double</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>bool</td>
<td>IntDI</td>
<td>!int</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>bool</td>
<td>BoolDI</td>
<td>!int</td>
<td>!IntDI</td>
<td>*</td>
</tr>
<tr>
<td>bool</td>
<td>UcharDI</td>
<td>!Uchar</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>bool</td>
<td>FloatDI</td>
<td>!float</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>Uchar</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>float</td>
<td>!float</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>double</td>
<td>!double</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>IntDI</td>
<td>!int</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>BoolDI</td>
<td>id</td>
<td>!UcharDI</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>FloatDI</td>
<td>!float</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>!double</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>IntDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>BoolDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>UcharDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>FloatDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>IntDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>BoolDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>UcharDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>FloatDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>IntDI</td>
<td>IntDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>IntDI</td>
<td>BoolDI</td>
<td>id</td>
<td>!IntDI</td>
<td>*</td>
</tr>
<tr>
<td>IntDI</td>
<td>UcharDI</td>
<td>id</td>
<td>!IntDI</td>
<td>*</td>
</tr>
<tr>
<td>IntDI</td>
<td>FloatDI</td>
<td>!float</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>BoolDI</td>
<td>BoolDI</td>
<td>!IntDI</td>
<td>!IntDI</td>
<td>*</td>
</tr>
<tr>
<td>BoolDI</td>
<td>UcharDI</td>
<td>!Uchar</td>
<td>!Uchar</td>
<td>*</td>
</tr>
<tr>
<td>BoolDI</td>
<td>FloatDI</td>
<td>!float</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>BoolDI</td>
<td>id</td>
<td>!UcharDI</td>
<td>*</td>
</tr>
<tr>
<td>Uchar</td>
<td>Uchar</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>!double</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>IntDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>BoolDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>UcharDI</td>
<td>id</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>FloatDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>IntDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>BoolDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>UcharDI</td>
<td>!float</td>
<td>!FloatDI</td>
<td>*</td>
</tr>
<tr>
<td>double</td>
<td>FloatDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
<tr>
<td>DMintDI</td>
<td>DMintDI</td>
<td>id</td>
<td>id</td>
<td>*</td>
</tr>
</tbody>
</table>
| %end %binop make_point

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>id</td>
<td>id</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>bool</td>
<td>id</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>id</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>double</td>
<td>id</td>
<td>id</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>Uchar</td>
<td>!int</td>
<td>id</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>float</td>
<td>!int</td>
<td>id</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>double</td>
<td>!int</td>
<td>id</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>Uchar</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>Uchar</td>
<td>double</td>
<td>!int</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>!int</td>
<td>!int</td>
<td>*pointCons_int_int</td>
</tr>
</tbody>
</table>
| %end %binop make_set

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>id</td>
<td>id</td>
<td>setCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>bool</td>
<td>id</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>Uchar</td>
<td>id</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>id</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td>id</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>!int</td>
<td>id</td>
<td>setCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>Uchar</td>
<td>!Uchar</td>
<td>!Uchar</td>
<td>*setCons_Uchar_Uchar</td>
</tr>
<tr>
<td>bool</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>bool</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>Uchar</td>
<td>Uchar</td>
<td>id</td>
<td>id</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>Uchar</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>!int</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>!int</td>
<td>!int</td>
<td>*setCons_int_int</td>
</tr>
</tbody>
</table>
| %end
---
%end

%unop sub
int id
bool id
Uchar id
float id
double id
IntPoint id
IntDI id
UcharDI id
BoolDI !IntDI
FloatDI id
%end

%unop cos sin tan exp log ceil floor sqrt sqr fabs
int !double
bool !double
Uchar !double
float !double
double id
IntDI !FloatDI
BoolDI !FloatDI
UcharDI !FloatDI
FloatDI id
%end

%unop max min '−' '®'
IntPoint id
IntSet id
UcharSet id
IntPS id
IntDI id
BoolDI !IntDI
UcharDI id
FloatDI id
%end

%unop abs
int !double
bool !double
Uchar !double
float !double
double id
IntDI id
BoolDI !IntDI
UcharDI id
FloatDI !IntDI
%end

########################### Sparse Binops ###########################

%sparse_binop dot
IntPoint IntPoint id id
%end

%sparse_binop contains
IntSet int id id
UcharSet Uchar id id
%end

%sparse_binop linear_product '©'
IntDI DDIntDI id id *
IntDI DDUcharDI !UcharDI id *
IntDI DDFloatDI !FloatDI id *
UcharDI DDIIntDI !IntDI id *
UcharDI DDUcharDI id id *
UcharDI DDFloatDI !FloatDI id *
FloatDI DDIIntDI !IntDI id *
FloatDI DDUcharDI !UcharDI id *
FloatDI DDFloatDI id id *
%sparse_binop addmax_product addmin_product multmax_product multmin_product
IntDI   DDIntDI         id              id      *
IntDI   DDUcharDI       !UcharDI        id      *
IntDI   DDFloatDI       !FloatDI        id      *
UcharDI DDIntDI         id              id      *
UcharDI DDFloatDI       !FloatDI        id      *
FloatDI DDIntDI         id              id      *
FloatDI DDUcharDI       !UcharDI        id      *
FloatDI DDFloatDI       id              id      *
%end

%sparse_binop '¢' '¡' '§' '¨'
IntDI   DDIntDI         id              id      *
IntDI   DDUcharDI       !UcharDI        id      *
IntDI   DDFloatDI       !FloatDI        id      *
UcharDI DDIntDI         id              id      *
UcharDI DDFloatDI       !FloatDI        id      *
FloatDI DDIntDI         id              id      *
FloatDI DDUcharDI       !UcharDI        id      *
FloatDI DDFloatDI       id              id      *
%end

%sparse_binop product '-'
IntDI   IntIntNbh       id      id      *
%end

%sparse_binop IA_boxy_pset
IntPoint        IntPoint        id      id      *
%end

%sparse_binop make_template
IntPS   IntDI   id      id      templateCons_IntPS_IntDI
IntPS   BoolDI  id      id      templateCons_IntPS_BoolDI
IntPS   UcharDI id      id      templateCons_IntPS_UcharDI
IntPS   FloatDI id      id      templateCons_IntPS_FloatDI
%end

%sparse_binop make_image
IntPS   int      id      id      imageCons_IntPS_int
IntPS   bool     id      id      imageCons_IntPS_bool
IntPS   Uchar    id      id      imageCons_IntPS_Uchar
IntPS   float    id      id      imageCons_IntPS_float
IntPS   double   id      !float  imageCons_IntPS_float
%end

%sparse_binop write_image
IntDI   charStar        id      id      *
BoolDI  charStar        id      id      *
UcharDI charStar        id      id      *
FloatDI charStar        id      !IntDI  id      *
IntDI   KString id      id      *
BoolDI  KString id      id      *
UcharDI KString id      id      *
FloatDI KString !IntDI  id      *
%end

%sparse_binop print_image
IntDI   charStar        id      id      *
BoolDI  charStar        id      id      *
UcharDI charStar        id      id      *
FloatDI charStar        id      !IntDI  id      *
IntDI   KString id      id      *
BoolDI  KString id      id      *
UcharDI KString id      id      *
FloatDI KString !IntDI  id      *
%end
%sparse_binop make_pixel
IntPoint int id id pixelCons_IntPoint_int
%end

Sparse Unops

%sparse_unop sum product '¬' '¬'
IntPoint id
IntDI id
BoolDI !IntDI
UcharDI id
FloatDI id
%end

%sparse_unop '¬'
int id
bool id
Uchar id
IntDI id
BoolDI !IntDI
UcharDI id
FloatDI !IntDI
%end

%sparse_unop '!' int id
bool id
Uchar id
IntDI id
BoolDI !IntDI
UcharDI id
FloatDI !IntDI
%end

%sparse_unop inorm mnorm enorm min max
IntPoint id
%end

%sparse_unop card choice empty
IntSet id
UcharSet id
IntPS id
%end

%sparse_unop inf sup boxy extensive
IntPS id
%end

%sparse_unop origin
IntPoint id
%end

%sparse_unop read_image
charStar id
KString id
%end

%sparse_unop IA_empty_ipset IA_universal_ipset IA_WhiteHole IA_BlackHole
int id
%end

%sparse_unop domain
IntDI id
BoolDI id
UcharDI id
FloatDI id
%end

%sparse_unop range
IntDI id
BoolDI !IntDI

int:int
bool:bool
Uchar:u_char
float:float
double:double
chars:char*
IntPoint:IA_Point<int>*
IntSet:IA_Set<int>*
UcharSet:IA_Set<u_char>*
FloatSet:IA_Set<float>*
IntPS:IA_Set<IA_Point<int>>*
IntDI:IA_Image<IA_Point<int>,int>*
BoolDI:IA_Image<IA_Point<int>,bool>*
UcharDI:IA_Image<IA_Point<int>,u_char>*
FloatDI:IA_Image<IA_Point<int>,float>*
DDIntDI:IA_DDTemplate<IA_Image<IA_Point<int>>,int>*
DBBoolDI:IA_DDTemplate<IA_Image<IA_Point<int>>,bool>*
DDUcharDI:IA_DDTemplate<IA_Image<IA_Point<int>,u_char>>*
DDFloatDI:IA_DDTemplate<IA_Image<IA_Point<int>,float>>*
IntIntNbh:IA_Neighborhood<IA_Point<int>,IA_Point<int>>*
KString:KString

### int ###
%binary '+ ' '- ' '* ' '/ ' '<' '<<' ' ^' '& ' '| ' '&& ' '|| ' '%: int int
%binary '< ' '<=' ' '> ' '>=' ' '== ' '!=': bool int int
%unary '- ' '! ' '~': int int

### bool ###
%binary '+ ' '- ' '* ' '/ ' '<' '^' '& ' '| ' '&& ' '||': bool bool bool
%binary '< ' '<=' ' '> ' '>=' ' '== ' '!=': bool bool bool
%unary '- ' '! ' '~': bool bool

### u_char ###
%binary '+ ' '- ' '* ' '/ ' '<' '<<' ' ^' '& ' '| ' '&& ' '||': Uchar Uchar
%binary '< ' '<=' ' '> ' '>=' ' '== ' '!=': bool Uchar Uchar
%unary '- ' '! ' '~': Uchar Uchar

### float ###
%binary '+ ' '- ' '* ' '/: float float float
%binary '<' '==' '!=': bool float float
%unary '-': float float

### double ###
%binary '+' '-' '*' '/': double double double
%binary '<' '<=' '>' '>=' '==' '!=': bool double double
%unary '-': abs exp log cos sin tan ceil floor fabs sqrt: double double

### IntPoint ###
%binary '+' '-' '*' '/': *IntPoint *IntPoint *IntPoint *IntPoint
%binary '+' '-' '*' '/': *IntPoint int *IntPoint
%binary '+' '-' '*' '/': *IntPoint *IntPoint int
%binary '+' '-' '*' '/': *IntPoint *IntPoint *IntPoint
%binary chi_eq chi_ne chi_gt chi_lt chi_ge chi_le: *IntPoint int
%binary chi_eq chi_ne chi_gt chi_lt chi_ge chi_le: *IntPoint *IntPoint
%binary chi_eq chi_ne chi_gt chi_lt chi_ge chi_le: *IntPoint *IntPoint int
%binary '<' '<=' '>' '>=' '==' '!=': bool *IntPoint *IntPoint
%binary '<' '<=' '>' '>=' '==' '!=': bool *IntPoint int
%binary '<' '<=' '>' '>=' '==' '!=': bool int *IntPoint
%binary pointcmp dot: int *IntPoint *IntPoint
%unary '-': *IntPoint *IntPoint
%unary min max sum product inorm mnorm: int *IntPoint
%unary enorm: double *IntPoint
%member dim: int *IntPoint

### IntSet ###
%binary '|' '&' '^' '/' '<' '<=' '>': *IntSet *IntSet *IntSet
%binary '|' '&' '^' '/' '<' '<=' '>': *IntSet int
%binary '|' '&' '^' '/' '<' '<=' '>': *IntSet *IntSet int
%member min max card choice empty: int *IntSet
%member contains: int *IntSet int

### UcharSet ###
%binary '|' '&' '^' '/' '<' '<=' '>': *UcharSet *UcharSet
%binary '|' '&' '^' '/' '<' '<=' '>': *UcharSet Uchar
%binary '|' '&' '^' '/' '<' '<=' '>': *UcharSet *UcharSet
%member min max card choice empty: Uchar *UcharSet
%member contains: Uchar *UcharSet Uchar

### IntPS ###
%binary '|' '&' '^' '/' '<' '<=' '>': *IntPS *IntPS *IntPS
%binary IA_boxy_pset: *IntPS *IntPoint *IntPoint
%member min max inf sup card choice empty boxy extensive: *IntPoint *IntPS
%member contains: *IntPoint *IntPS *IntPoint
%unary IA_empty_ipset IA_universal_ipset IA_WhiteHole IA_BlackHole:
  *IntPS int
%unary origin: *IntPS *IntPoint
### IntDI ###
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *IntDI
*IntDI *IntDI
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *IntDI
*IntDI int
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *IntDI int
*IntDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *IntDI *IntDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *IntDI int
%binary '<' '<=' '>' '>=' '==' '!=': bool int *IntDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *IntDI
*IntDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *IntDI int
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI int
*IntDI
%binary max min: *IntDI *IntDI *IntDI
%binary max min: *IntDI *IntDI int
%binary max min: *IntDI int *IntDI
%unary max min sum product: int *IntDI
%unary '!' '-' '~' abs: *IntDI *IntDI
%unary isqrt sqr sgn: *IntDI *IntDI
%member card:int *IntDI
%member domain: *IntPS *IntDI
%member range: *IntSet *IntDI

### BoolDI ###
%binary '&' '|' '^': *BoolDI *BoolDI *BoolDI
%binary '&' '|' '^': *BoolDI *BoolDI bool
%binary '&' '|' '^': *BoolDI bool *BoolDI
%binary '==' '!' strict_ne: bool *BoolDI *BoolDI
%binary '==' '!' strict_ne: bool *BoolDI bool
%binary '==' '!' strict_ne: bool bool *BoolDI
%member card: int *BoolDI
%member domain: *IntPS *BoolDI

### UcharDI ###
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *UcharDI
*UcharDI *UcharDI
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *UcharDI
*UcharDI Uchar
%binary '+' '-' '*' '/' '%' '&' '|' '^' '<<' '>>' '&&' '||': *UcharDI
Uchar *UcharDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *UcharDI *UcharDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *UcharDI Uchar
%binary '<' '<=' '>' '>=' '==' '!=': bool Uchar *UcharDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *UcharDI
*UcharDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *UcharDI Uchar
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI Uchar
*UcharDI
%binary max min: *UcharDI *UcharDI *UcharDI
%binary max min: *UcharDI *UcharDI Uchar
%binary max min: *UcharDI Uchar *UcharDI
%unary max min sum product: Uchar *UcharDI
%unary '!' '-' '~': *UcharDI *UcharDI
%member card: int *UcharDI
%member domain: *IntPS *UcharDI

### FloatDI ###
%binary '+' '-' '*' '/' '&&' '||': *FloatDI *FloatDI *FloatDI
%binary '+' '-' '*' '/' '&&' '||': *FloatDI *FloatDI float
%binary '+' '-' '*' '/' '&&' '||': *FloatDI float *FloatDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *FloatDI *FloatDI
%binary '<' '<=' '>' '>=' '==' '!=': bool *FloatDI float
%binary '<' '<=' '>' '>=' '==' '!=': bool float *FloatDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *FloatDI *FloatDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI float *FloatDI
%binary chi_lt chi_le chi_eq chi_ne chi_ge chi_gt: *BoolDI *FloatDI float
%binary max min: *FloatDI *FloatDI *FloatDI
%binary max min: *FloatDI *FloatDI float
%binary max min: *FloatDI float *FloatDI
%member card: int *FloatDI
%unary max min sum product: float *FloatDI
%unary '! ' '-' abs: *FloatDI *FloatDI
%unary sgn cos sin tan exp log ceil floor sqrt sqr fabs: *FloatDI *FloatDI
%member domain: *IntPS *FloatDI

### DDIntDI ###
%binary linear_product min max: *DDIntDI *DDIntDI *DDIntDI
%binary linear_product addmax_product addmin_product: *IntDI *IntDI *DDIntDI
%binary linear_product addmax_product addmin_product: *IntDI *DDIntDI *IntDI
%binary multmax_product multmin_product: *IntDI *IntDI *DDIntDI
%binary multmax_product multmin_product: *IntDI *DDIntDI *IntDI

### DDBoolDI ###
%binary hit_miss: *BoolDI *BoolDI *DDBoolDI
%binary hit_miss: *BoolDI *DDBoolDI *BoolDI

### DDUcharDI ###
%binary linear_product addmax_product addmin_product: *UcharDI *UcharDI *DDUcharDI
%binary linear_product addmax_product addmin_product: *UcharDI *DDUcharDI *UcharDI
%binary multmax_product multmin_product: *UcharDI *UcharDI *DDUcharDI
%binary multmax_product multmin_product: *UcharDI *DDUcharDI *UcharDI

### DDFloatDI ###
%binary linear_product addmax_product addmin_product: *FloatDI *FloatDI *DDFloatDI
%binary linear_product addmax_product addmin_product: *FloatDI *DDFloatDI *FloatDI
%binary multmax_product multmin_product: *FloatDI *FloatDI *DDDFloatDI
%binary multmax_product multmin_product: *FloatDI *DDDFloatDI *FloatDI

### IntIntNbh ###
%binary sum product max min: *IntDI *IntDI *IntIntNbh
%binary sum product max min: *IntDI *IntDI *IntIntNbh
### KString ###
%binary '<<': KString KString KString
%binary '<' '>' '==' '!=': bool KString KString
%member GetLength: int KString
A19. LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Put your text here. The text uses the *Body Text* style, which specifies double spacing. Notice the blank line before this paragraph.