Object-Oriented Programming without Messages

by

Fang Lin
B.Sc., Nanjing University of Aeronautics and Astronautics, 1994

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

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. M.R. Levy, Supervisor (Department of Computer Science)

[redacted]

Dr. M. van Emden, Department Member (Department of Computer Science)

[redacted]

Dr. Peter F. Driessen, External Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Andy Schloss, External Examiner(School of Music)

University of Victoria

Supervisor: Dr. Michael R. Levy

Abstract

Woozle is a programming language that was originally designed to facilitate Web shell programming. That is, Woozle programs simplify object-object transfer of Internet accessible objects, as well as extraction of content from objects. In this thesis, we study the implementation issues that arise from Woozle's unique view of object-object interaction.

Examiners:

[redacted]

Dr. M.R. Levy, Supervisor (Department of Computer Science)

[redacted]

Dr. M. van Emden, Department Member (Department of Computer Science)

[redacted]

Dr. Peter F. Driessen, External Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Andy Schloss, External Examiner (School of Music)

Table of Contents

List of Figures

Acknowledgements

I would like to express my sincere thanks to my thesis advisor, Dr. Michael Levy, University of Victoria. Without his direction and encouragement this thesis would have been impossible.

I am grateful for the help of all other people who contributed to the completion of this thesis by proofreading the thesis and making constructive suggestions. These include Frank G, Lin, David Chen, Robert Hatch and Dr. Xiangjun Qiu.

Chapter 1
Introduction

It has been nearly three decades since the first introduction of the notion *object* as a programming construct in *Simula* [14], a programming language that was designed for developing computer simulations. However, it is *Smalltalk* [10], the first uniformly object-oriented language (every data item is an object), that promoted the use of object-oriented methodology for software prototyping and application development in domains other than computer simulation. Since the advent of Smalltalk, there has been an explosion of interest in object-oriented programming [15], which has led to many varieties of object-oriented programming languages.

The object-oriented methodology was considered revolutionary because of its underlying conceptual simplicity. Rather than using the existing concepts of type, procedure, parameter passing and so on, object-oriented languages employ some fundamental concepts, namely object, class and message. The key idea in the object-oriented paradigm is to turn everything into an object. In other words, the main idea is to unify all the data items in the language. However, on closer inspection, it turns out that Smalltalk cannot unify all the data items because messages are not objects (although their arguments are). In conventional object-oriented languages, message dispatch is

determined by keyword selection. In situations where the main activity is object-to-object transfer, the use of keywords adds an unnecessary syntactic overload to the language. A related limitation of Smalltalk and similar languages is that they are not higher order: functions like "+", "*" and so on are not objects, they are considered messages. In the case of Web, most activity is object-to-object transfer. The use of messages is not necessary in these cases, as long as the receiver is able to determine the type of the sender. Note that we do not claim that the use of messages diminishes the power of languages that use them. Rather, we wish to explore the idea of achieving even greater unification of the object-oriented paradigm and thereby create a flexible and concise programming language. To investigate this approach, we recently developed a **W**eb **O**bject-**O**riented **S**cripting **L**anguage named *Woozle*. Like *Perl* [19] and *JavaScript* [12], Woozle is a high-level scripting language for Web applications. However, unlike Perl and JavaScript, Woozle is designed as a messageless object-oriented language. The more unified paradigm in Woozle seems to be sufficient for the development of concise web programs.

Woozle achieves its goal by adopting as the underlying design the idea that object-to-object transfer is naturally expressed as:

receiver sender

where the term *receiver* denotes the object that receives other objects and the term *sender* denotes the object that is sent to the receiver. We call this primitive operation *object composition*. In Smalltalk it is necessary to include a message keyword to achieve the same effect, as in, for example,

receiver get: sender

In Woozle, the meaning of composition is determined by the identity of the receiver and the type of the sender. In other words, message dispatch is determined by type matching rather than by keyword selecting.

Woozle was motivated by the following problem. Suppose that you are the designer of a fairly complex Web site. You may wish to allow readers to download all or part of the site for off-line reading. For example, your site contained a home page and two subdirectories, one for images and one for documents. You wished to download the home page, some of the images and two of the documents. How do you provide such downloads? You may write a program in Java [6] to specify the copy operation because Java has built-in URL support. The program that uses the built-in "URL" object to download the contents of a URL and uses the built-in "File" object to create a file in the local directory may contain more than thirty lines of code. However, the equivalent of this thirty-line Java code can be implemented in Woozle in just a couple of lines shown below. In this example, Woozle has made the Web download easier and conceptually clearer.

```
localDir = (dir "C:\Wired\April\");
remoteDir = (http "//www.wired.com/");
remoteGifs = (remoteDir "images/" ["pic1.gif","pic2.gif","pic3.gif"]);
remoteDocs = (remoteDir "docs/" ["doc1.html","doc2.html"]);
localDir (remoteDir "index.html");
[(localDir "localDocs\"),(localDir "localGifs\")][remoteDocs, remoteGifs];
```

To execute Woozle, we need some method for implementing messageless object-oriented scripts. The main contribution of this thesis is an algorithm for implementing such programming languages. This algorithm is based on Graph Reduction. But, the algorithm is novel in a number of respects that will be explained in chapter 4. Because Woozle is a novel language, we will give an overview of it in chapter 3. In chapter 2, the brief introduction of Graph Reduction will be given. Conclusions and future work will be drawn in chapter 5.

Chapter 2

A Brief Overview of Graph Reduction

Object-Oriented programming is now widely used, and is supported by many programming languages, including Smalltalk, C++ and Java. This thesis uses an implementation technique called *one-level-delay tree reduction* that is based on Graph Reduction [16]. The one-level-delay tree reduction algorithm uses infix order rather than prefix order. Graph Reduction is often used for functional programming language implementation, but not, before now, used for object-oriented programming language implementation. Graph Reduction was an important development because it led to very efficient implementations of functional programming languages. To aid the reader of this thesis, I will present a brief introduction to Graph Reduction in this chapter.

2.1 A Brief Introduction to Graph Reduction

The major task of programming in functional languages is to build definitions; that is, to construct functions to solve a given problem. In functional programming, expressions

that contain occurrences of the names of functions are evaluated by using the definition of functions as reduction rules for converting expressions to a reduced form.

Graph reduction is an implementation technique for functional programming languages. In graph reduction, both the definition of functions and the expressions to be evaluated are represented by graphs. The graph evaluation procedure consists of a sequence of simple steps, called reductions. A reduction replaces a reducible expression in the graph with its reduced form. A redex is a node in a graph that can be reduced. Evaluation is complete when there are no more redexes. We say the expression is then in *normal form*. If there exists more than one redex in the expression, the outermost redex will be selected for evaluation first. The process of evaluating an expression involves the following three steps:

- selecting the outermost redex in a graph which represents the expression
- reducing the selected redex
- updating the root of the redex in the graph with the reduction result

The process of evaluation repeats the above three steps until there are no more redexes in the expression.

To illustrate, suppose that the function "double" is defined as:

double x = x+x

This definition specifies that the "double" is the function of a single argument "x", which computes "x+x". The body of the function "double" is represent by the following graph.

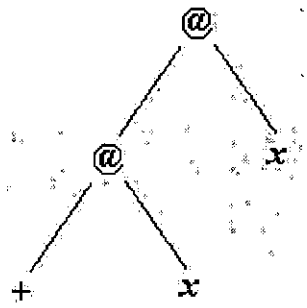


Figure 1. The body of the function "double".

where the sign "@" in Figure 1 represents the function application.

To evaluate an expression:

double 3

the following graph is created to represent the expression:

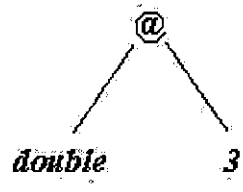


Figure 2. Apply the function "double" to 3.

The outermost redex in the expression is the application of the function "double". To apply the function "double" to 3, the graph reduction algorithm replaces the redex with an instance of the body of the function "double":

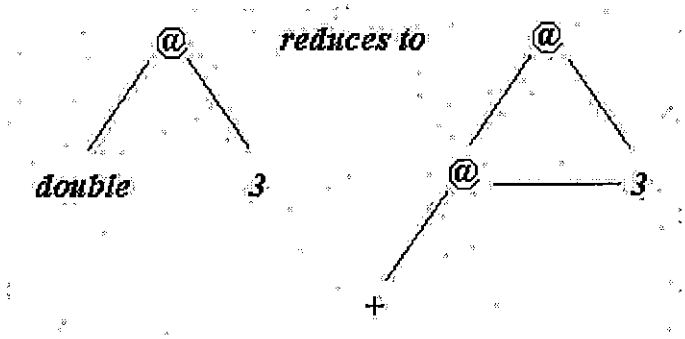


Figure 3. Replace the redex with an instance of the body of the function "double".

Now the outermost redex is the application of the function "+". It is the root of the tree whose leftmost child is "+". The "+" is one of the built-in operators. Thus, the built-in rules are used and "6", the result of the addition, replaces the root of the redex (shown in Figure 4).

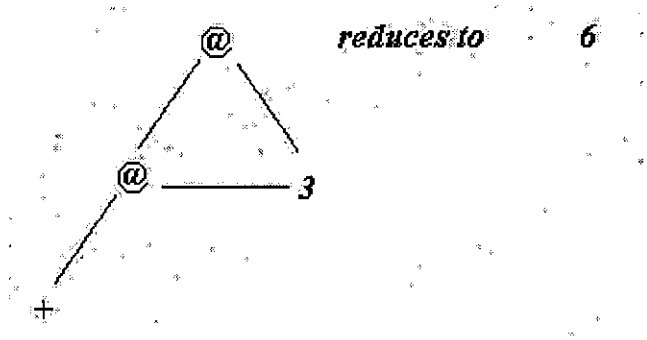


Figure 4. Replace the redex with 6.

Since there are no more redexes in the expression "6", the process of evaluation is complete.

Note that the parameter "3" is shared after the first reduction has taken place. If the actual parameter was not simply a leaf node, it would only have to be reduced once

provided that its value replaced its original root node. Using graphs (rather than trees) allow the interpreter to exploit this sharing.

From the above examples, we can see that the redex of the function "double" and the redex of the function "+" are treated differently. They are in fact different kinds of redex. In graph reduction, there exist two kinds of redex. They are the *supercombinators redex* and the *built-in primitive redex.*

Supercombinator and Built-in Primitives

A *supercombinator* is a function defined by a user such as the function "double". A *built-in primitive* is a system built-in function or operator such as "+" in the above example. If a function application is a supercombinator application and if there are sufficient arguments, then it is also a redex. It is reduced by replacing the root with an instance of its body, that is, by substituting the actual arguments for the formal parameters within its body. Supercombinators without arguments are called *constant applicative forms* (CAFs). They can always be reduced. If a function application is a built-in primitive application, then the function application is a redex only if the arguments are evaluated. Otherwise, the application is not a redex and its arguments must be evaluated by using the three-step process outlined at the beginning of this section.

As we pointed out previously, graph reduction algorithm evaluates an expression by repeating the three steps until a normal form of the expression is reached. To explain the process of evaluation in detail, we elaborate the following three steps: 1) find the next redex; 2) reduce the redex; 3) update the redex with the reduction result.

1) **Find the next redex**

The first step of the evaluation process is to find the next redex. If there exists more than one redex in the expression, the outermost redex will be selected to evaluate.

Suppose an expression to be evaluated is:

$f\,E_1\,E_2$

whose graph representation is:

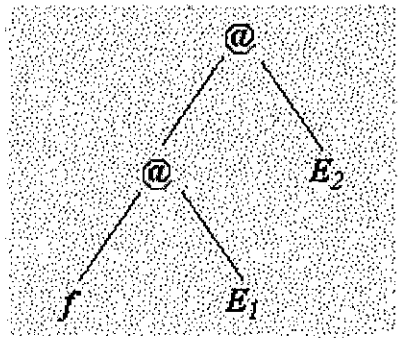


Figure 5. The graph representation of the expression "f E_1E_2".

where "f" is a function which takes one argument.

To find the function "f", we will go down the left branch of the application node from the root until we find a supercombinator or built-in primitive. The sequence of left nodes from the root to the leaf is called the *spine* of the expression. The act of traversing the spine is called *unwinding* the spine. The addresses of the nodes on the spine are pushed onto a stack. For example, to find the function "f", the spine stack would look like this:

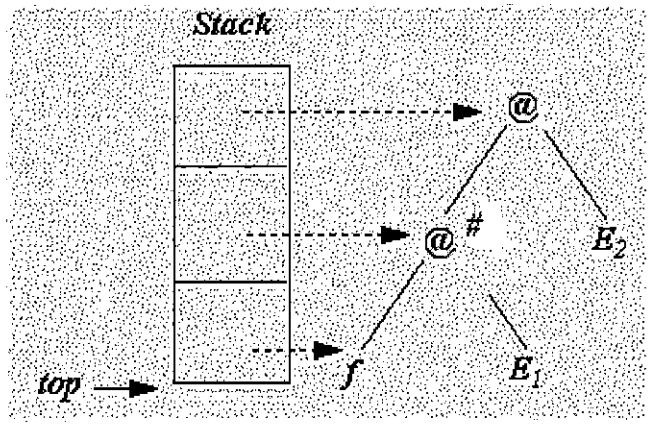


Figure 6. The spine stack.

Based on how many arguments there are of the found function, we will go back up the spine that number of application nodes to find the root of the outermost function application. Following this procedure, we find the root of the application of the function "f"(recall that "f" takes one argument), as marked with a '#'(shown in Figure 6).

From the above example, we can see that two steps are required to find the next redex. 1) Starting from the root, we unwind the spine until we reach a supercombinator or built-in primitive. 2) Based on the number of the arguments the supercombinator or primitive

takes, we go back up the spine that number of application nodes to find the root of the redex.

In step 2, if there are not enough application nodes (arguments) in the spine to match the number of arguments in the function, then the expression has reached *weak head normal form* (WHNF). Most evaluators will stop to reduce the arguments when the expression reaches WHNF. The evaluation result will be a partial application.

If the function we find is a supercombinator, then its application is certainly a redex. If the function is a strict primitive, such as "+", its arguments must be evaluated before reduction takes place. We evaluate its arguments by repeating the same evaluation process.

2) **Reduce the redex**

As noted before, a supercombinator redex is reduced by replacing the root with an instance of its body, that is, by substituting the actual arguments for the formal parameters within its body. The actual arguments are shared by the formal parameters rather than copied. Recall the example given at the beginning of this section. To construct the body of the function "double", the occurrences of the formal argument "x" in its body are substituted by the pointers of the actual argument "3". In the case of CAFs (supercombinators without arguments), the supercombinators themselves are redexes. For example, "double4" is a CAF:

double4 = double 4

We do not want to instantiate a new copy of "double 4" whenever "double4" is called, because that would needlessly repeat the computation of "double 4". Rather, the supercombinator "double4" is the root of the redex, and would be updated with the result of instantiating its body.

A built-in primitive redex is reduced by using the built-in rules. For example, to reduce an application of built-in operator "+", the built-in rule for "+" is used to calculate the sum of two arguments.

3) **Update the redex with the reduction result**

After the selected redex is reduced, the result will update the root of the redex. In the case where the redex is shared, the reduction is done only once.

For example, suppose we wish to evaluate an expression:

double (double 4)

where the definition of the function "double" is the same as it was in the example "(double 3)".

The complete evaluation steps of the expression are given in Figure 7. Notice that in the second step of evaluation, the redex "double 4" is shared by both of the arguments of the function "+". The redex is done only once. The result 8 updates the root of the redex as shown in the fourth step.

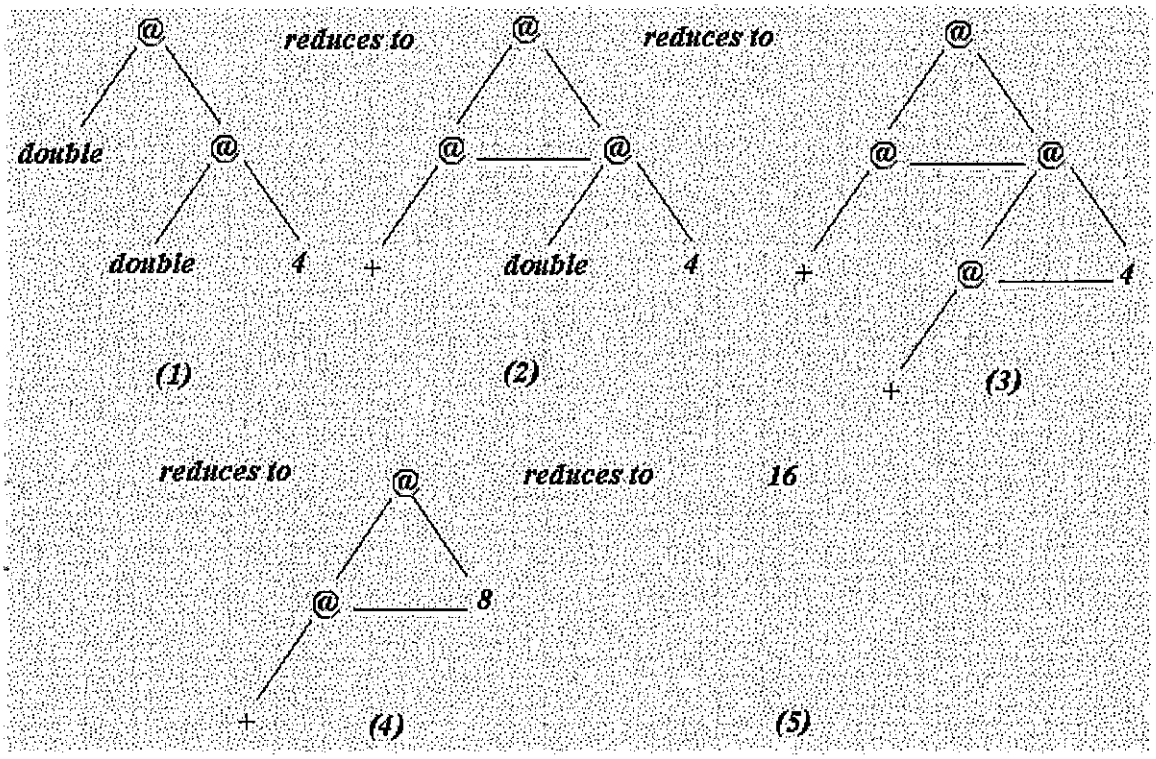


Figure 7. The complete evaluation steps of the expression "(double(double 4))".

Graph reduction, the three steps of which we have outlined in this section, is one way to implement functional programming languages. This technique has been adapted for the implementation of Woozle due to the similarities between funcitonal languages and Woozle.

In functional programming languages, a function will not be evaluated until the function gets enough arguments. If it does not get enough arguments, the result is a partial function. This indicates that the application of the function can not be completed until it gets its remaining arguments. Recall from chapter 1, Woozle's basic object –object transter model is :

receiver sender

During each object composition, the receiver takes only one sender and some time one step of composition will lead to an incomplete result. To illustrate, suppose we evaluate the expression "1+2" in Woozle, the object "+" will first be sent to object "1". The result of the composition, object "(1+)" is an incomplete result that is still waiting for a sender. This is very similar to the partial function in functional languages. In addition, functions in functional programming languages are considered to be values that can be the arguments of other functions. Similarly in Woozle, the results of object compositions can again be senders or receivers in other object compositions.

However, in Woozle, the meaning of an object composition is determined by the identity of the receiver and the type of the sender. If a sender is a Woozle expression, then the type of the sender can not be decided before its evaluation. In this manner, Woozle is quite different than functional programming languages.

The *One-step-delay tree reduction* algorithm derived from graph reduction is developed to implement Woozle. The detailed description of the algorithm is given in chapter 4.

Chapter 3

Woozle Language Overview

Woozle was motivated by the following problem. Suppose that you are the designer of a fairly complex Web site. Your documents may spread over several directories, and even, possibly, over several machines. Perhaps you designed your links so that the site is, in principle, easily moved from one machine to another. You may even wish to allow readers to download all or part of the site for off-line reading. How do you provide Web-site downloads?

A generic approach is to use a program that copies the Web site, given a root document. Although this is a simple solution, it has difficulties. Essentially, the problem is that a Web site is typically a sub-graph of a possible huge graph, and it is difficult for a generic program to know how to stop traversing paths when making the copy.

A second approach is to allow the Web designer to specify the copy operation. They could, for example, provide a link like this: To download this month's articles on widgets, click here. The link would download a script. The script would be run (using a helper program), and it would (a) retrieve the correct documents, and (b) recreate a copy of the remote directory structure on the local machine.

Woozle was designed specifically to allow this kind of scripting. The basic operation is object transfer (or, depending on your outlook, function composition). Thus, if L denotes a local object (for example, a file), and R a remote object (an html document, for example), then a "download" is written

L R;

and an "upload" is written

R L;

The implementation naturally takes care of details of the underlying protocol, issuing the appropriate GET or PUT requests and so on. In the present implementation, the operations are synchronous (so the program blocks until the transfer is complete), but we expect to develop an asynchronous version of the interpreter soon.

The Woozle interpreter supports integers, strings, lists and a few built-in objects, namely: http, dir, file and = (assignment). Expressions behave mostly as expected. For example,

x = (x+1);

will increment x by 1. The rule for a list receiver is this:

$[a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_m] = [a_1\ b_1, a_2\ b_2, \ldots, a_k b_k]$ *where* $k = min(n,m)$

$[a_1, a_2, \ldots, a_n]\ x = [a_1\ x, a_2\ x, \ldots, a_n\ x]$ *for all other objects x*

Most objects iterate if the sender is a list:

$x\ [a_1, a_2, \ldots, a_n] = [x\ a_1, x\ a_2, \ldots, x\ a_n]$

These rules were designed to help specify concise downloads. For example, suppose you wish to download some files to a local directory:

dir "C:\mydir\" (http "//www.csr.uvic.ca/~mlevy/" (["f1","f2","f3"] ".html"));

is a convenient short-hand for

dir "C:\mydir\" (http "//www.csr.uvic.ca/~mlevy/f1.html");

dir "C:\mydir\" (http "//www.csr.uvic.ca/~mlevy/f2.html");

dir "C:\mydir\" (http "//www.csr.uvic.ca/~mlevy/f3.html");

Object-Oriented programming is supported by many programming languages, including Smalltalk, C++ and Java., but Smalltalk is the only programming language, which attempts to present a completely unified view of objects. In Smalltalk, objects communicate by messages. Message keywords are use to determine method dispatch. In Woozle, the unifying notion is object-object transfer. In Woozle, there are two parts to a transaction: a receiver and the sent object, which we will call the sender. This is a slight abuse of grammar, since the word sender implies the sending agent rather than the sent object, but we found the term "sent object" to be cumbersome. The syntax and semantics of Woozle will be described, informally, in this chapter. We will also describe the behavior of the built-in types, and give the syntax for user-defined objects.

3.1 The Object-to-Object Transfer Paradigm for Web Shell Languages

Woozle is designed to be a Web shell language suitable for accessing web objects in a manner similar to the way conventional shell languages access resources on single machines. A typical application would perform the following kinds of tasks:

- Retrieve a remote document and save it locally

    - Receiver: local file descriptor
    - Sender: remote file
- Display a remote document in a local browser
    - Receiver: local browser program
    - Sender: remote file
- Send email
    - Receiver: SMTP server
    - Sender: local email object (made up of the appropriate email fields)
- Filter a remote file
    - Receiver: a filter object (based, for example, on regular expressions)
    - Sender: remote file

This object transfer model led to the conception of object-oriented programming that differs from existing languages in this respect: messages play no special role – they are simply objects. Although in conventional object-oriented languages such as Smalltalk and Java an object is capable of receiving and responding to messages, messages themselves are not objects. However, in the sampling of typical Web shell scripts examined above, messages do not appear to be needed. We certainly do not claim that the use of messages eliminates the power of languages, but it does slightly limit their expressivity. In a Web shell language, if "L" denotes a local object and "R" a remote object, then the action "download" can be written as

L R;

And "upload" as

R L;

Passing higher-order objects around yields more concise programs because we can avoid the wrapper machinery that is necessary when doing the same thing in message-based languages.

3.2 Language Description

A Woozle environment is made up of collection of objects. The syntax of built-in objects, such as integers, strings, lists, and a number of literal objects including +, -, *, /, as well as user-defined objects, are described below.

3.2.1 Woozle Abstract Syntax

Abstract syntax is required to express how objects interact with one another. A Woozle program is a sequence of Woozle terms. A Woozle term is a Woozle expression that is terminated by a semi-colon. Four types of expression are permitted:

1. An identifier
2. An operator
3. A literal
4. A composition

An *identifier* is any sequence of alphanumeric characters starting with an alpha. An *operator* is any one of the following: +, -, *, /, ==, <, <=, >, >=, !=, =. A Woozle *variable* is any identifier that is not a Woozle identifier, operator or literal.

There are four types of *Woozle literals*: *integers*, *strings*, *booleans*, and *lists*. An integer is represented as a sequence of digits. A string is a sequence of characters surrounded by double quotation marks. There are two booleans, namely true and false. A list is a comma-separated sequence of Woozle expressions surround by square brackets. A composition is an expression followed by another expression

Woozle's concrete syntax is similar to its abstract syntax, except that the association is left to right. This is unlike many other languages, where precedence depends on function. There is no algebraic hierarchy. However, as with many other languages, parentheses can be used to alter the order of precedence.

3.2.2 Types

Woozle takes a very simple view of types: the *type* of a Woozle object is denoted by a literal string. For example:

Type 12 = "integer"

Type "woozle" = "string"

Type false = "boolean"

The literal objects +, *, and so on, have types "+", "*", and so on. All types are ordered on a lattice whose greatest element is always the string "any". The Woozle type system, used for expression reduction, does not try to capture type information in the usual mathematical sense.

3.2.3 Reduction Machine

The Woozle system contains an abstract machine that can perform reductions. This *reduction machine* reduces expressions until they are in normal form (can not be further reduced) and prints their results. The reduction machine uses reduction rules to determine when and how to reduce expressions. The algorithm for reduction will be explained in chapter 4. The next section illustrates Woozle's informal semantics by describing Woozle reduction rules.

3.2.4 Informal Woozle Semantics

A Woozle expression is represented as a sequence of objects. The normal association is left to right. However, parentheses can be used to alter the order of association. The first object in a sequence of Woozle objects called a *receiver*. A receiver expects to receive one or more objects that are called *senders*. The receiver's reaction will depend on the type of the receiver, the type of the senders, and the internal state of the receiver. The expression reduction algorithm is based on the identity and types of the objects in an expression. If there is no rule that reduces the Woozle expression, then the expression is

called a *compound object.* The concise definition of compound objects is given in chapter 4. We will explain the semantics of built-in objects by giving their reduction rules. The sign '=>' designates a single reduction step.

Integers

The following ten rules apply to the reduction of integers (both j and k shown below represent integer objects):

1. j + k => the sum of j and k
2. j - k => the difference between j and k
3. j * k => the product of j and k
4. j / k => the quotient of j and k
5. j == k => true if j is equal to k, false otherwise
6. j != k =>true if j is not equal to k, false otherwise
7. j < k =>true if j is less than k, false otherwise
8. j <= k => true if j is less than or equal to k, false otherwise
9. j > k => true if j is greater than k, false other wise
10. j >= k => true if j is greater than or equal to k, false otherwise

Regarding the above ten rules, the compound objects could be "+k", "j-", and so on. These compound objects may receive other objects, or be sent to other objects.

Strings

The following seven rules apply to string reduction (both j and k shown below represent the string objects):

1. j + k => concatenates k to the end of j
2. j k => copy k to j
3. j == k => true if j is lexicographically equal to k;, false otherwise
4. j < k => true if j is lexicographically less than k, false otherwise
5. j <= k => true if j is lexicographically less than or equal to k, false otherwise
6. j > k => true if j is lexicographically greater than k, false otherwise
7. j >= k => true if j is lexicographically greater than or equal to k, false otherwise

Lists

There are three rules for list reduction ($a_{1...}a_n$ and $b_{1...}b_n$ shown below represent any objects):

1. $[a_1,a_2,\ldots,a_n][b_1,b_2,\ldots,b_m] \Rightarrow [a_1\ b_1,a_2\ b_2,\ldots a_k b_k]$, where k = min(m,n)
2. $[a_1,a_2,\ldots,a_n]\ x \Rightarrow [a_1\ x,\ a_2\ x,\ldots,a_n\ x]$,where x is any object whose type is not "list"
3. $x\ [a_1,a_2,\ldots,a_n] \Rightarrow [x\ a_1,x\ a_2,\ldots x\ a_n]$, where x is any object whose type is not "list"

Lists can be used as vectors. For example:

$[a_1,a_2] + [b_1,b_2]$ $\Rightarrow [a_1+,a_2+][b_1,b_2]$ *by list rule 2*

$\Rightarrow [a_1+b_1,a_2+b_2]$ *by list rule 1*

Booleans

There are two rules for boolean reduction (e_1 e_2 ...e_n and f_1 f_2 ...f_n are Woozle expressions):

1. true $\{e_1;e_2;\ldots;e_n;\}$ $\{f_1;f_2;\ldots;f_m;\}$ => $e_1;e_2;\ldots;e_n;$
2. false $\{e_1;e_2;\ldots;e_n;\}$ $\{f_1;f_2;\ldots;f_m;\}$ => $f_1;f_2;\ldots;f_n;$

Assignment

Assignment is denoted by "=". Any object can be assigned to a variable as in

x = y;

where "y" is an object and "x" is a variable.

Like most conventional programming languages, the right-hand side may need to be parenthesized if the programmer wishes the right-hand side to be evaluated prior to assignment.

By using the rules of lists together, an expression with single assignment can be used to swap the value of more than two variables. For example, the following expression swaps the value of the variables "x" and "y".

[x, y] = [y, x] =>[x =, y=] [y, x]

=>[x = y; y = x]

In conventional programming languages, swapping the value of two variables normally introduce a third temporary variable. However in Woozle, this sort of multiple assignment is expressible and simultaneous. Here is another example. The following expression initializes several variables to zero.

```
[x,y,z] = 0
```

The next section describes how users can add new objects to a Woozle session.

3.2.5 User-defined Objects

Woozle comes with a set of predefined objects designed for Web scripting, such as integer objects, list objects, the *http* object (used to retrieve remote resources), file objects and so on. Woozle also allows users to create their own objects. The process of creating a user-defined object is done by defining a set of associated rules using a special Woozle built-in object called *define*.

To illustrate, suppose we wish to define a box that can hold two other objects. This is done in Woozle with the following script using the "define" object:

```
define "box" (this: "box", x: "any", y:"any"){
        this.1 = x;
        this.2 = y;
};
```

where the "box" is the name of a user-defined object, and the pairs of round brackets and curly brackets specify the reduction rule of the box. The content between the round pair

is called the head of the rule and the content between the curly pair is called the body of the rule. The head of a rule specifies which objects can interact with the user-defined object. The body of a rule specifies how these objects interact with the user-defined object. The rule of the box object indicates that if a box receivers two other objects whose types are both "any", the box will hold these two objects by saving them one by one inside its storage rooms. The rule for the "define" object itself will be discussed later in this section. It must be noted that each "define" expression as above can only specify one rule for a user-defined object. If objects need more than one rule, more "define" expressions are needed. For example, if a user would like to retrieve the contents of a box object defined, as in the above example, she needs to add more rules. For example:

```
define "first" { };                                    (1)
define "second" { };                                   (2)
define "box" (this:"box",x:"first"){                   (3)
        return (this.1);
};
define "box" (this:"box",x:"second"){                  (4)
        return (this.2);
};
```

Expression (3) and (4) add two more rules for the box object. These are used to retrieve the content of the box. Expressions (1) and (2) define the objects "first" and "second" for the purpose of assisting defining the rules in expressions (3) and (4). Note that the objects "first" and "second" are created without defining the head of a rule. Such objects

that can be evaluated without interacting with other objects are called self-evaluated objects. These objects are normally used to assist the definition of other objects.

As mentioned above, the "define" object itself is a Woozle built-in object. In order to understand how the "define" object is employed to create a rule for a user-defined object, we need to explain two reduction rules associated with the "define" object as the following:

1) define N (S_1, ... ,S_m) {E_1; ... ;E_n} => a user-defined object named N;

2) define N {E_1; ... ;E_n} => a user-defined object named N;

where *N* is the name for an object, and (S_1, ..., S_m) and {E_1; ...; E_n} specify a rule for the object. Both of those two reduction rules can be reduced to a user-defined object named N. To explain these two rules in detail, we elaborate as the following each of the three elements of the rules: object name, object rules and reduction results.

Object name

An object name is represented by a Woozle string object that denotes the name of a user-defined object. For example, the Woozle expression

define "box"...;

defines an object named "box".

Object rules

Recall that a reduction rule of Woozle contains two parts: its head and its body.

- **head**

The head of a rule containing a sequence of parameters id denoted by

$(S_1, ..., S_m)$

where each S_i is a parameter defined by a pair of its name and its type joined by ":". For example,

x:"integer"

specifies a parameter named "x" whose type is integer.

Each parameter refers to a Woozle object. The parameter determines the name, the type and the position of the object when it is interacting with a user-defined object. In the head of a rule, a Woozle built-in object, "this" is used as a self-reference to the user-defined object. The type of the user-defined object is also specified in the head of the rule. For example, to specify a user-defined object named "factorial" whose type is also "factorial", we have

define "factorial" (...,this: "factorial",...){...};

- **body**

The body of a rule containing a sequence of expressions is denoted by

$\{E_1; ...; E_n\}$

where each E_1 is a Woozle expression.

A user-defined object contains storage rooms that can hold other objects. Recall that in the beginning of this section, we give an example of a user-defined object named "box" which can hold two other objects. The reference to a storage room is indicated by a Woozle built-in object "this" and a number connected by ".", i.e.

this.1 = x;

this.2 = y;

where the object "this" is a reference to the box object and the number "1" indicates the first storage room of the box.

Result

After either of the two reduction rules of the "define" object is selected to perform the reduction, the new reduced object named N will be returned. A Woozle built-in object "return" is used to return the result of the evaluation of a body of a rule.

By default, the user-defined object itself will be returned as the result of the evaluation. For example, if the result of the evaluation is an integer object: 5, we can use the following expression:

Return 5;

The two reduction rules shown in page 28 of the "define" object are designed for two purposes. The first is used to "define" objects that are associated with at least one reduction rule. And, the second is used to define self-evaluated objects. By default, the type of a self-evaluated user-defined object is denoted by the name of the object. The following three examples show the use of these two rules.

Example I: a self-evaluated user-defined object named "plus"

Consider the following Woozle expressions.

```
define "plus"{
        Return +;
};
```

The above expressions define an object named "plus" which can replace the Woozle built-in object +. The following two expressions are both evaluated to 3.

```
1 plus 2 => 3
1 + 2    => 3
```

Example II: a user-defined object named "triple" with two reduction rules

Consider the following Woozle expressions.

```
define "triple" (this:"triple",x:"integer") {
        return (x+x+x);
```

```
};
define "triple" (x:"integer",this:"triple") {
        return (x+x+x);
};
```

The above expressions define an object named "triple" which triples the value of the integer x. The following two expressions are both evaluated to 9;

```
triple 3 => 9
3 triple => 9
```

The triple object can be considered as a unary function, which takes an argument using either one of the formats specified in the head of its rules and return the triple value of the argument "x".

Example III: a recursive user-defined object named "fac"

A user-defined object named "N" is recursive if the name "N" appears in a body of its rule. The following script defines a recursive object named "fac".

```
define "fac" (this:"fac",n:"integer") {
        (n ==1) {return 1} {
                (n<1) {return 1} {
                        return (n * (fac (n-1) ) );
                };
```

```
        };
};
```

The following expression is evaluated to 6.

```
fac 3 => 6
```

In the next chapter, we describe the algorithm used for executing Woozle programs.

Chapter 4

Woozle Language Implementation

Woozle is implemented in Java and tested on IBM PC compatibles. It is an interpreted language. The interpreter consists of three major components: a scanner, a parser and a reduction machine (shown in Figure 8)

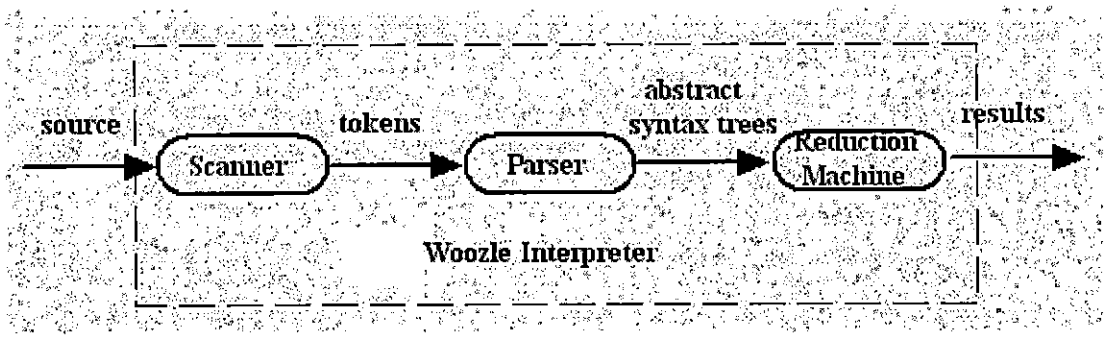


Figure 8. Three components of Woozle Interpreter

The scanner performs the lexical analysis by translating a Woozle source into a token sequence. The parser, which is a bottom-up LR(0) Parser [1], constructs an abstract syntax tree by processing the token sequence emitted by the scanner. The reduction machine then carries out successive reductions on the abstract syntax tree and generates the evaluation result. The interpreter employs a multithreaded implementation, which allows simultaneous lexical analyzing and abstract syntax tree constructing/reducing.

The syntax of Woozle can be briefly described as follows using the Backus-Naur Form (BNF) [20]

<Woozle> ::= { *<expression>* ; }$^{+}$

<expression> ::=*<expression><expression>*

<expression> ::=(*<expression>*)

<expression> ::= *<identifier>* |< *literal>* |*<operator>*

<identifier> ::= *alpha* | {*alpha* | *digit*}*

<literal> ::= *<integer>* |< *string>* |< *boolean>* | *<list>*

<operator> ::= +| - | * | / | == | <= | >= | != | =

<integer> ::= {*digit*}$^{+}$

<string> ::= " {*character*}* "

<boolean> ::= *true* | *false*

<list> ::=[ ] | [{*<expression>* , }* *<expression>*]

In the above syntax description:

- The symbol "::=" denotes Definition.
- Non-terminals are enclosed between the special symbols "<" and ">", such as "<list>" and "<expression>".
- Terminals consisting of symbols usually appear as is, such as "+" and "*".
- The star closure, denoted by "{...}*", says to repeat whatever is in the brackets zero or more times. The plus closure, denoted by "{...}$^{+}$", says the structure in the brackets must appear at least once.
- The vertical bar "|" is used when there are more than one rule for any non-terminal.

Based on the above syntax, tokens generated by the scanner could be single characters (such as "["), some special sequences of characters (such as "=="), quoted strings, integers and identifiers (which are names assigned to variables or special objects, i.e. "typeof"). The parser considers the context of each token and classifies groups of tokens into larger entities such as "list" and "expression". The products of the parser are abstract syntax trees. They are graph representations of the expressions that to be evaluated.

Woozle expressions can be evaluated through two ways: command-line input and file input. To illustrate, suppose we evaluate the following two expressions where "X" is a variable.

(1) X =([1,2] +);

(2) X 3;

The results of evaluation for above two expressions are:

(1) [1+, 2+] (saved in "X")	by list rule 2
(2) X 3 => [1+3, 2+3]	by list rule 2
=>[4,5]	by integer rule 1

where the rules are described in chapter 3.

Figure 9 and Figure 10 are two screen snapshots for evaluating the above two expressions. Figure 9 is the screen snapshot for evaluating expressions through command-line input. There are three fields in Figure 9, which are "Evaluating", "Input History" and "Output History". The "Evaluating" field is the command-line input area.

The "Input History" field shows the history of all the inputs, which are the above two expressions. The "Output History" field shows the history of the above evaluation results. Pressing the button "Reset" restarts the interpreter. The button "Do it" can be used to ask the interpreter to evaluate the expression in the command-line input area.

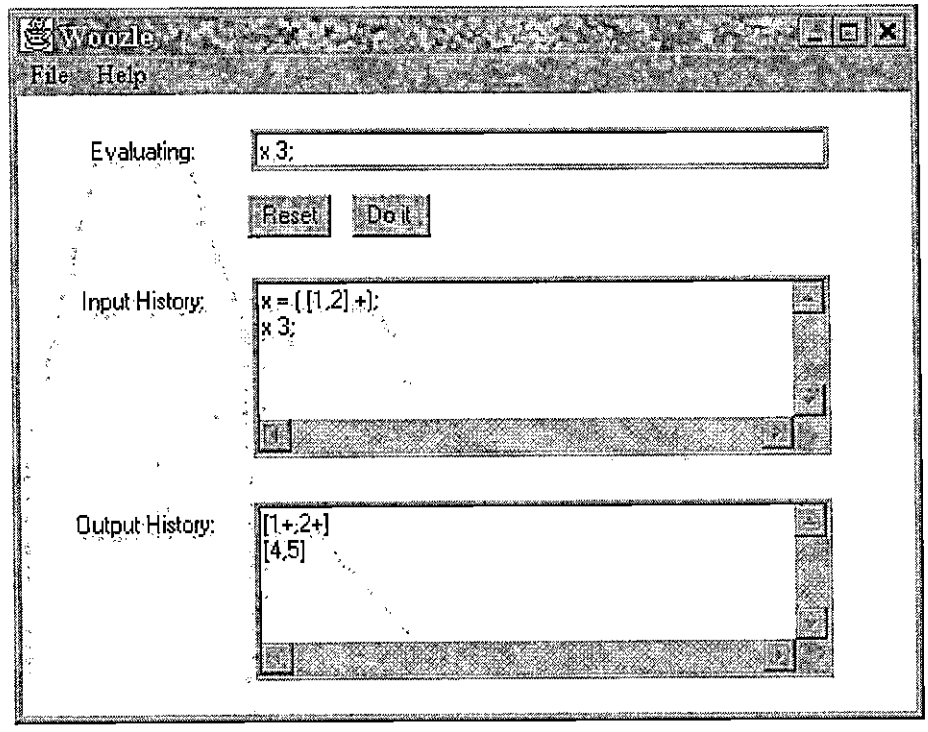


Figure 9. The screen snapshot for evaluating expressions through command-line input.

Figure 10 is the screen snapshot for evaluating expressions through file input. There are two windows in this snapshot, where the back windows is the same as the one in Figure 9, while the front window is used to display the content of the input file. Suppose the sample expressions are saved in a file named "C:\example.woo". Selecting item "Open" under menu "File" opens the input file in the front window. After the button "Run" is

pressed, the interpreter evaluates the expressions in the input file. The results are printed in the "Output History" field as shown in Figure 10.

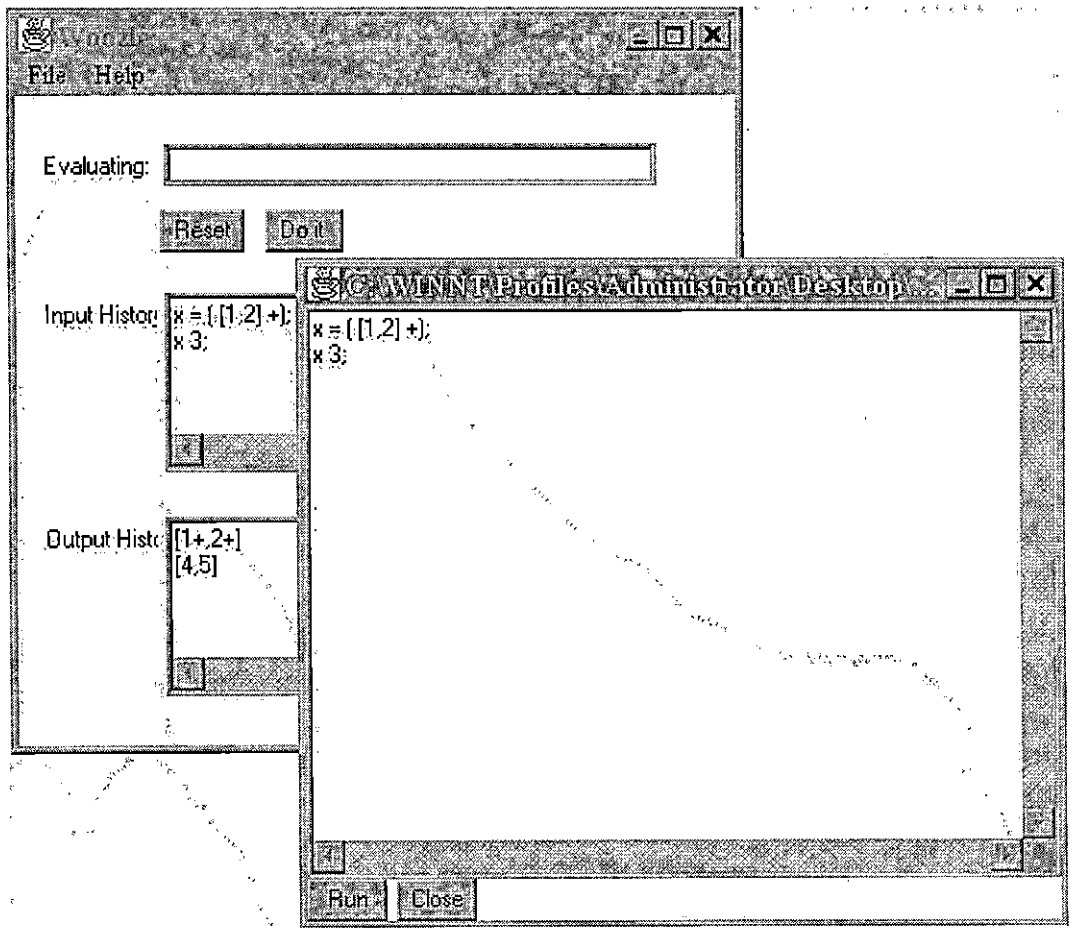


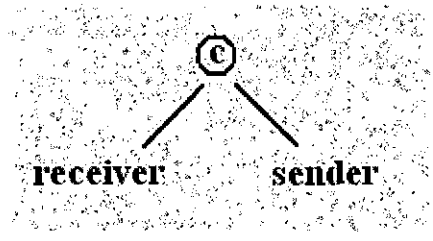
Figure 10. The screen snapshot for evaluating expressions through file input.

The most significant contribution of Woozle is the algorithm, called *one-level-delay* tree reduction, which implements the reduction machine. This algorithm is derived from graph reduction (which is introduced in chapter 2). However, unlike graph reduction, which is based on functions, one-level-delay tree reduction is based on rules. The rest of

the chapter describes this unique algorithm that is used to achieve the evaluation of expressions in Woozle. Related concepts are discussed as well.

4.1 Woozle Abstract Syntax Trees

In Woozle, the expression to be evaluated is represented in the form of its *abstract syntax tree*. The leaves (nodes without children) of the tree are all single objects. The intermediate nodes (nodes with children) are called *coordinators*, and these are used to represent object composition. The left child of a coordinator is the receiver. The right child of a coordinator is the sender. Any tree with a coordinator named "C" as its root is called the *Coordinator Tree* of "C". The expression "*receiver sender*" is represented as:



where the node with the sign"©"is the coordinator.

The trees (A) and (B) in Figure 11 denote the expressions "(1+)2" and "1(+2)". Both abstract syntax trees evaluate to 3. However, since the brackets in expressions can alter the sequence of composition, the structures of the tree (A) and (B) are different. Figure 12 shows a slightly more complicated example.

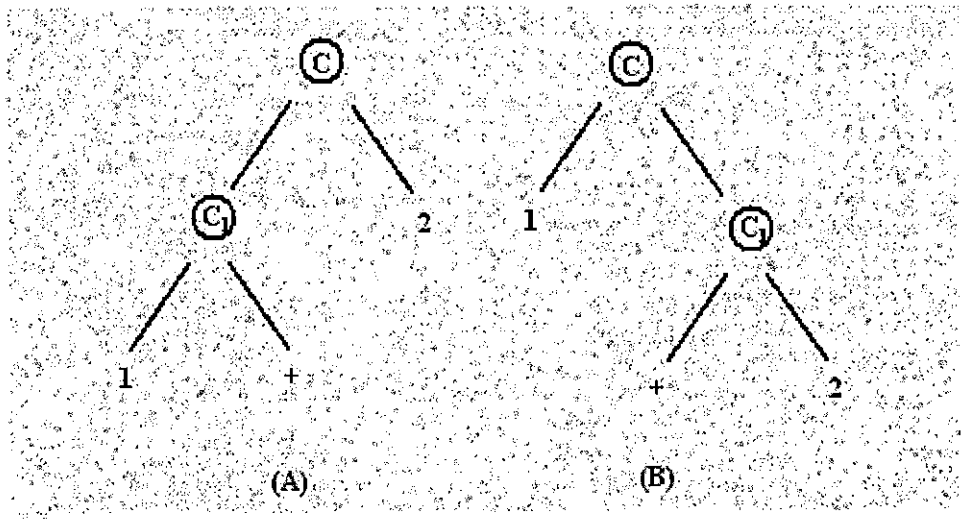


Figure 11. The abstract syntax trees of the expressions "(1+)2" and "1(+2)".

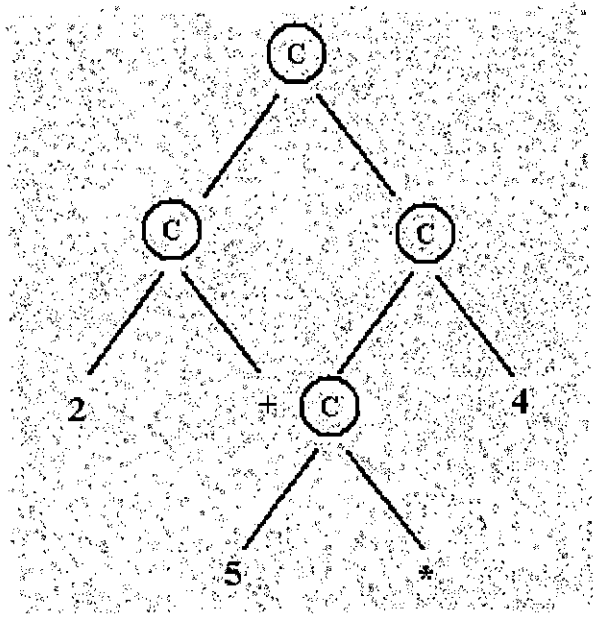


Figure 12. The abstract syntax tree of the expression "2+(5*4)".

4.2 The Universal Rule Table

Woozle expressions are reduced by applying a sequence of reduction rules. When a receiver object is combined with a sender object, the receiver object will choose the correct rule for the compound object. These reduction rules can be stored either separately inside each object or together inside the Woozle system accessible by all Woozle objects. After designing both algorithms, we found the latter superior because it is simpler to implement, it more easily handles user-defined objects, and it is less redundant. Therefore, we define the *Universal Rule Table* (URT) as the collection of all reduction rules, and have evaluations of expressions referred to the URT for their reduction rules.

Each rule in the URT is identified by an *index*. As stated before in page 29, a rule contains two parts: its *head* and its *body*. The head of a rule is a sequence of object types. Recall that the type of a Woozle object is denoted by a literal string. The body of a rule is the function that implements the rule. In the following discussion of the process of tree reduction, two terms are frequently used: *fully match* and *partially match*.

Fully Match

Expression evaluation is based on type matching. Consider a rule R_x with the head $<T_1, T_2, ..., T_n>$. The expression "$X_aX_{a+1}...X_b$" *fully matches* the head of the rule R_x if and only if

a=1, b=n, and the type of X_i is T_i, where i=1,2,…,n.

Partially Match

If the type sequence of the expression "$X_aX_{a+1}...X_b$" is a proper sub-sequence of <T_1, T_2, ..., T_n>, we say the expression *partially matches* the head of the rule R_x.

When the head of a rule is fully matched by an expression, the body of that rule (a function) is then invoked. The result of the execution of the body returns as the result of the evaluation of the expression. For example, there is a rule R_x whose head is a 3-Tuple <"integer", "+", "integer">. The body of the rule is a function that takes two integers and returns the sum of these two integers. The expression "1+2" can be evaluated using this rule as its reduction rule and the result 3 is the result of the evaluation.

When there are two or more rules whose heads are fully matched by an expression, the rule with the smaller index value will be selected as the reduction rule for this expression.

4.3 Compound Object

Each object composition takes only two objects: the receiver and the sender. Therefore, for any rule with N elements in its head, we need N-1 steps of composition before the corresponding evaluation is completed. During the first N-2 steps, the head of the rule may only be partially matched. Results of such partial matchings are also Woozle

objects, which are called *compound objects*. Recall from Section 4.1 that Woozle single objects are the leaves of a Woozle abstract syntax tree. Woozle compound objects are the sub-trees of a Woozle abstract syntax tree.

Compound object

The Woozle expression "$X_iX_{i+1}...X_j$" is a compound object if there exist at least one rule in the URT, which has the head:

$<T_1, T_2, ..., T_n>$,

where

- $i<j$ and $j-i+1<= n-1$
- $T_i', T_{i+1}', ..., T_j'$ represents the type sequence of $X_iX_{i+1}...X_j$
- $T_i', T_{i+1}', ..., T_j'$ is the proper sub-sequence of $T_1, T_2, ... T_n$

Matched Rule Set

A *Matched Rule Set* (MRS) is created for each compound object. The MRS contains all the indexes of the rules that satisfy the above mentioned compound object conditions. When we compose the compound object with another object, only the rules whose indexes are contained in its MRS are searched. Because this prevents needless searching of the rest of the URT, using a MRS increases the efficiency of rule searching.

Let us return to the example shown in Figure 11, and consider the composition of the coordinator "C_1" with object "1" and object "+". If there is no rule which has been fully matched by the expression "1+" with the type sequence <"integer", "+">, then object "1" and object " +" will be composed to form the compound object "1+". Its MRS contains all the indexes of the rules that are partially matched by the sub-expression "1+" whose type sequence is <"integer", "+">.

4.4 Sender, Receiver and Sender-Receiver

For a particular rule with the head <T_1, T_2, …, T_n>, let us consider the Woozle object "$X_iX_{i+1}…X_j$" where i<=j.

The object "$X_iX_{i+1}…X_j$" can act as a receiver if the following conditions are satisfied:

a) $j-i+1 < n$

b) the type of X_I is T_1, the type of X_{i+1} is T_2, …, and the type of X_j is T_{j-i+1},

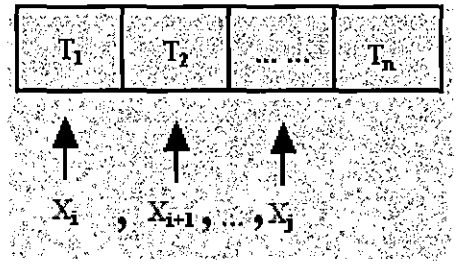


Figure 13. Receiver

This indicates that the object "$X_iX_{i+1}...X_j$" expects to receive the object sequence "X_{j+1}, X_{j+2}, ..., $X_{j+(n-j+i+1)}$", where the type of X_{j+1} is T_{j-i+2}, the type of X_{j+2} is T_{j-i+3}, ..., and the type of $X_{j+(n-j+i-1)}$ is T_n.

The object "$X_iX_{i+1}...X_j$" can act as a sender if the following conditions are satisfied:

a) $j-i+1<n$

b) the type of X_j is T_n, the type of X_{j-1} is T_{n-1}, ..., and the type of X_i is T_{n-j+i}

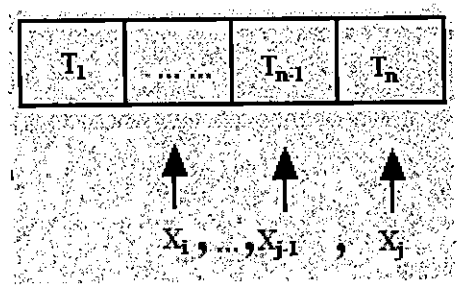


Figure 14. Sender

This indicates that the object "$X_iX_{i+1}...X_j$" expects to be sent to the object sequence "X_{i-1}, X_{i-2}, ..., $X_{i-(n-j+i-1)}$", where the type of X_{i-1} is $T_{n-j+i-1}$, the type of X_{i-2} is $T_{n-j+i-2}$, ..., and the type of $X_{i-(n-j+i-1)}$ is T_1.

The object "$X_iX_{i+1}...X_j$" can act as a sender-receiver if the following conditions are satisfied:

a) $j-i+1 < n-1$

b) the type of X_i is $T_k(1<k<n)$, the type of X_{i+1} is T_{k+1}, ..., and the type of X_j is T_{k+j-i},

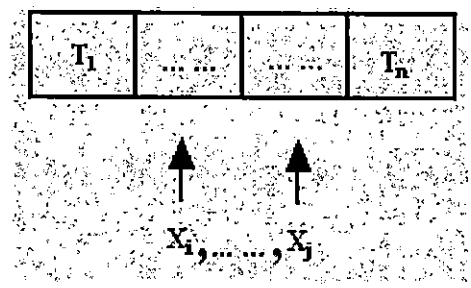


Figure 15. Sender-Receiver

This indicates that the object "$X_iX_{i+1}...X_j$": **a)** expects to be sent to the object sequence "$X_{i-1}, X_{i-2}, ..., X_{i-(k-1)}$", where the type of X_{i-1} is T_{k-1}, the type of X_{i-2} is T_{k-2}, ..., and the type of $X_{i-(k-1)}$ is T_1; **b)** expects to receive the object sequence "$X_{j+1}, X_{j+2}, ..., X_n$", where the type of X_{j+1} is $T_{k+j-i+1}$, the type of X_{j+2} is $T_{k+j-i+2}$, ..., and the type of $X_{k+j-i+(n-j)}$ is T_n.

As the consequence of imposing those conditions for Woozle objects, it can be seen that the left child object of a coordinator inside an abstract syntax tree can only act as a receiver or a sender-receiver. Similarly, the right child object can only act as a sender or a sender-receiver. Such concepts are used to avoid unnecessary rule searching during object compositions.

4.5 Constrained Associative Property

In Woozle, rule matching is used in object compositions. The heads of rules distinguishes them from each other. However, it is possible that the head of a rule is the sub-sequence of the head of another rule. Consider the following example. There are two rules in the URT applying to integer objects j and k:

1) *j ^ => double j*

2) *j ^ k => j to the power of k*

where the head of the rule 1) is a proper subsequence of that of the rule 2).

When evaluating the expression

2 ^ 3

according to composition sequence, the object 2 and the object ^ will be chosen to evaluate first using the rule 1):

2 ^ 3 => 4 3

In this case, the rule 2) will never be chosen as the reduction rule, since the head of the rule 1) is a proper subsequence of the head of the rule 2). We consider the rule 2) is covered by the rule 1). However, if the rule 1) doesn't exist and there is no other rule in the URL which covers the rule 2). Then the following two expressions both evaluate to 8:

(2 ^) 3 => 8

2 (^ 3) => 8

This indicates that when the rule 2) is not covered by another other rules in the URT, the composition sequence is irrelevant to the evaluation result of the expression "2 ^ 3".

Hence, for a particular rule R_y with the head "$<T_1,T_2,\ldots T_n>$", consider the evaluation of the expression "$X_i,X_{i+1},\ldots X_j$", where

- $j-i+1 = n$ and $n>2$
- the type of X_i is T_1, the type of X_{i+1} is T_2, …, and the type of X_j is T_n

We say the rule R_y satisfies the *associative property* if the following condition is met:

> *During the n-1 steps of composition, whatever composition sequence we choose the rule R_y will always be selected as the reduction rule.*

That is, if the rule R_y is chosen to be the reduction rule, the result of the evaluation will be independent of the sequence of composition.

Not every rule in the URT satisfies this property. For example, suppose we have a rule R_x with the head "$<T_1,T_2,\ldots T_{n-1}>$". The head of the rule R_x is a proper sub-sequence of the rule R_y. If the rule R_x is in the URT, R_x will be selected to be the reduction rule when composing the sub-expression "$X_i,X_{i+1},\ldots X_{j-1}$". Therefore, the result of evaluating the expression "$X_i,X_{i+1},\ldots X_j$" changes due to the existence of the rule R_x. In this case, the rule R_y does not satisfies the associative property.

From the above example, we can see that the rule R_y satisfies the associative property if the URT does not contain a rule (except the rule R_y itself) whose head is a proper sub-sequence of the head of the rule R_y. We call such an associative property a *Constrained Associative Property*.

To illustrate, consider the evaluation of the expression "a b c" with the type sequence $<T_a, T_b, T_c>$, when the rule R_y with the head $<T_a, T_b, T_c>$ exists in the URT. The following two expressions will evaluate to the same result.

1) *(a b) c*

2) *a (b c)*

if neither of the following two rules exist in the URT.

1) *the rule R_x with the head "$<T_a,T_b>$"*

2) *the rule R_z with the head "$<T_b,T_c>$"*

Because of this property, the Woozle program can be written in a flexible way and behaves in a manner similar to a functional programming language. For example, in Woozle, there exists a rule with the head <"integer", "+","integer"> in the URT. This rule satisfies the constrained associative property. Because of this property, the expression "1+" can not be reduced, and the result of the evaluation, the compound object "1+", can exist individually in the system. This compound object can act as a receiver that receives an integer object "y" and returns the evaluation result of the expression "1+y". The behavior of this compound object is similar to the behavior of a function in a functional programming language.

4.6 Tree Reduction Algorithm

Now let us consider the composition of a coordinator's children on a syntax tree. If the expression represented by the Coordinator Tree only partially matches the head of a given rule, then the expression is not reducible. On the other hand, if the expression fully

matches the head of a given rule, the expression is reducible. The result returned by the rule body evaluation replaces the root of the Coordinator Tree. This is called a step of tree reduction. Successive reductions are then carried out on the abstract syntax tree until the tree is in its normal form (that is, the tree cannot be further reduced). Thus, the expression evaluation involves the following two distinct tasks:

- selecting the next Coordinator Tree to be composed
- composing (if not reducible) the selected Coordinator Tree, **or** reducing (if reducible) the selected Coordinator Tree and updating the root of the tree with the reduction result.

4.6.1 Coordinator Tree Selection Order

The algorithm for selecting the next Coordinator Tree to be reduced is determined by post-order binary tree-traversal. For example, in Figure 16, the order in which the compositions are done is:

B->C->A

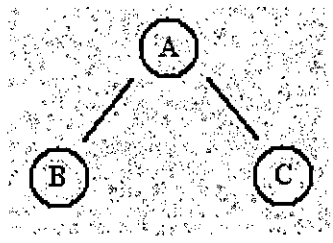


Figure 16. A tree with three coordinators.

Consider the following more complicated abstract syntax tree, which has nine coordinators:

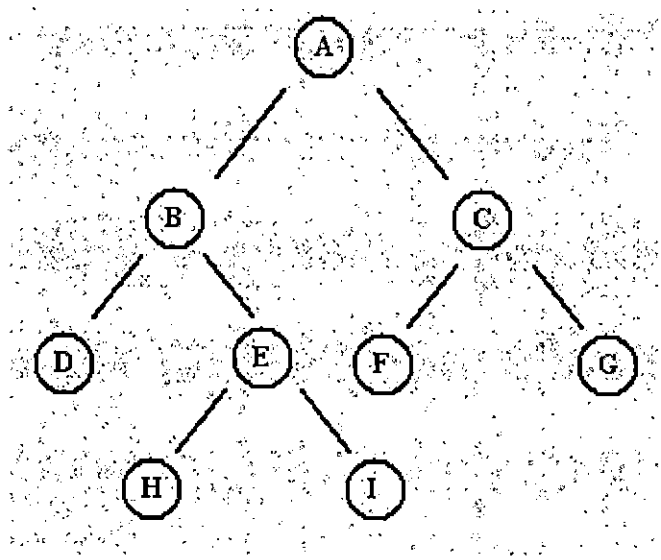


Figure 17. A tree with nine coordinators.

According the algorithm, the search order for the next Coordinator Tree to be reduced should be:

D -> H -> I -> E -> B -> F -> G -> C -> A

4.6.2 One-Level-Delay Tree Reduction

In Woozle, object compositions are based on rule matching. Rules are different from each other and are distinguished by their heads. However, this does not guarantee that the head of a rule cannot be the sub-sequence of the head of another rule, that is, that not every rule in the URT satisfies the constrained associative property.

For example, consider the expression "a b c" with type sequence "<T_a, T_b, T_c>", and suppose that the following two rules are in the URT:

1) *the rule R_x with head <T_a, T_b>*

2) *the rule R_y with head <T_a, T_b, T_c>*

where the head of the rule "R_x" is the proper sub-sequence of the head of the rule "R_y". Thus, R_y does not satisfy the constrained associative property. When evaluating the expression "a b c", the Coordinator Tree with children objects "a" and "b" is chosen to be composed first. The head of the rule R_x will then be fully matched, whereas the head of the rule R_y will never be fully matched. In a situation like this, where one rule is covered by another rule, we provide a unique algorithm called *one-level-delay tree reduction.* This algorithm can give all the rules in the URT, especially those without satisfying the constrained associative property, opportunities to be fully matched.

The idea behind the *one-level-delay tree reduction* algorithm is to use brackets to force reduction. The root of a Coordinator Tree, a coordinator, is *marked* if its representing expression is within brackets. Assume an expression fully matches the head of the rule R_x in the URT. If the root of the expression's corresponding Coordinator Tree, a coordinator, is marked, then the marked expression will be reduced first. Otherwise, if the coordinator is not marked, the decision of whether or not to reduce the expression will be made by the coordinator's parent. The variable named *selectedRuleIndex* associated with each coordinator is used to save the index of the rule R_x. This indicates that the reduction of a Coordinator Tree (an expression) may be delayed until the composition of

its parent level. Thus, we call such a reduction algorithm a one-level-delay tree reduction algorithm.

The one-level-delay algorithm is divided into four methods, the selection of which depends on the children of the coordinator to be composed. The methods and the conditions for selecting them are listed below:

Method A

Method A is chosen when the children of a coordinator are both single objects. In Figure 18, the coordinator "C" is shown with two children X_j and X_{j+1}, that are both single objects.

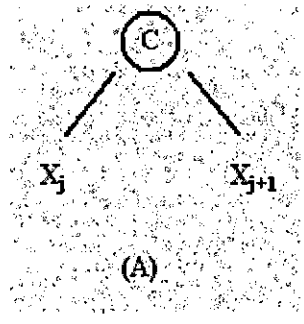


Figure 18. X_j and X_{j+1} are both single objects.

The method used to compose the children of the coordinator "C" consists of the following six steps.

Assume that R^c is the MRS of the coordinator "C" and S^c is the *selectedRuleIndex* of the coordinator "C".

1) Set R^c to empty; Set S^c to minus 1.

2) Select the next unselected rule R_i in the URT:

- If the type sequence of X_jX_{j+1} fully matches the head of rule R_i, then:
  - If the coordinator "C" is marked, the Coordinator Tree of "C" will be reduced by evaluating the body of the rule R_i. Update "C" with the result of the evaluation and go to step 6.
  - If the coordinator "C" is not marked, and if it does not have a parent on the tree, then the Coordinator Tree of "C" will be reduced by evaluating the body of the rule. Update "C" with the result of the evaluation, then go to step 6.
  - If the coordinator "C" is not marked, if "C" has a parent on the tree, and if S^c is minus 1, then set S^c to i, and proceed to step 3.
- If the expression X_jX_{j+1} partially matches the head of rule R_i, add index i into R^c.

3) If there exist a rule in the URT that has not been previously selected, go back to step 2.

4) If R^c is empty and S^c is minus 1, report error and stop the evaluation.

5) If R^c is not empty, then the Coordinator Tree of "C" is a compound object with the MRS R^c and *selectedRuleIndex* S^c.

6) Proceed to the next coordinator to be composed.

Method B

Method B is chosen when the left child of a coordinator is a single object and the right child is a compound object. In Figure 19, the coordinator "C" has two children, the single object X_j and the compound object $X_{j+1}X_{j+2}...X_k$ represented by the Coordinator Tree of "C_1".

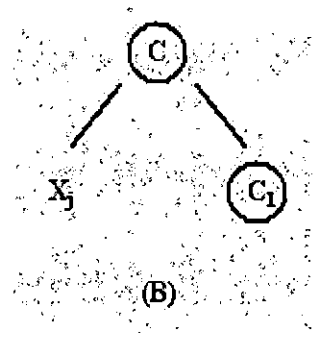


Figure 19. X_j is a single object and "C_1" is a compound object

The method used to compose the children of the coordinator "C" consists of the following six steps.

Assume that R^c is the MRS of the coordinator "C", that S^c is the *selectedRuleIndex* of "C", that R^R is the MRS of the coordinator "C_1", and that S^R is the *selectedRuleIndex* of "C_1".

1) Set R^c to empty; Set S^c to minus 1

2) Select the next unselected rule R_i in R^R:

- If the compound object $X_{j+1}X_{j+2}X_k$ can act as a sender, then:

- If the expression $X_jX_{j+1}...X_k$ fully matches the head of the rule R_i and the coordinator "C" is marked, the Coordinator Tree of "C" will be reduced by evaluating the body of the rule R_i. Update "C" with the result of the evaluation and go to step 6.
- If the expression $X_jX_{j+1}...X_k$ fully matches the head of the rule R_i, if the coordinator "C" is not marked, and if it does not have a parent on the tree, then the Coordinator Tree of "C" will be reduced by evaluating the body of the rule. Update "C" with the result of the evaluation and go to step 6.
- If the expression $X_jX_{j+1}...X_k$ fully matches the head of the rule R_i, if the coordinator "C" is not marked, and if it has a parent on the tree, then set S^c to i and go to step 6.
- If the expression $X_jX_{j+1}...X_k$ partially matches the head of the rule R_i, add i into R^C and proceed to step 3.

□ If the compound object $X_{j+1}X_{j+2}...X_k$ can act as a receiver, then:

- Proceed to step 3.

□ If the compound object $X_{j+1}X_{j+2}...X_k$ can act as a sender- receiver, then:

- If the expression $X_jX_{j+1}...X_k$ partially matches the head of the rule R_i, add i into R^C and proceed to step 3.

3) If there exists a rule in R^R that has not been previously selected, go back to step 2.

4) If R^c is empty and S^c is minus 1, report error and stop the evaluation.

5) If R^c is not empty, the Coordinator Tree of "C" is a compound object with the MRS R^c and the *selectedRuleIndex* S^c.

6) Proceed to the next coordinator to be composed.

In the above method, S^R is not used to do the one-level-delay. The reason is that "C_1" is the right child of "C" and is marked for sure. Therefore, S^R will always be minus 1.

Method C

Method C is chosen when the right child of a coordinator is a single object and the left child is a compound object. In Figure 20, the coordinator "C" has two children, the compound object $X_{j+1}X_{j+2}...X_k$ represented by "C_1" and the single object X_{j+1}.

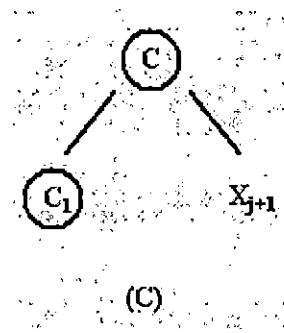


Figure 20. X_{j+1} is a single object and "C_1" is a compound object

The method used to compose the children of the coordinator "C" consists of the following six steps:

Assume that R^c is the MRS of the coordinator "C", that S^c is the *selectedRuleIndex* of "C", that R^L is the MRS of the coordinator "C_1", and that S^L is the *selectedruleIndex* of "C_1":

1) Set R^c to empty; Set S^c to minus 1.

2) Select the unselected Rule R_i in R^L:

   - If the compound object $X_iX_{i+1}...X_j$ can act as a sender:
     - Proceed to step 3.
   - If the compound object $X_iX_{i+1}...X_j$ can act as a receiver, then:
     - If the expression $X_iX_{i+1}...X_jX_{j+1}$ fully matches the head of the rule R_i and "C" is marked, then the Coordinator Tree of "C" will be reduced by evaluating the body of the rule R_i. Update "C" with the result of the evaluation and go to step 6.
     - If the expression $X_iX_{i+1}...X_jX_{j+1}$ fully matches the head of the rule R_i, if the coordinator C is not marked, and if it does not have a parent on the tree, then the Coordinator Tree of "C" will be reduced by evaluating the body of the rule. Update "C" with the result of the evaluation and go to step 6.
     - If the expression $X_iX_{i+1}...X_jX_{j+1}$ fully matches the head of the rule R_i, if the coordinator C is not marked, and if it has a parent on the tree, then set S^c to i and go to step 6;
     - If the expression $X_iX_{i+1}...X_jX_{j+1}$ partially matches the head of the rule R_I, add i into R^C and proceed to step 3.
   - If the compound object $X_iX_{i+1}...X_j$ can act as a sender- receiver, then:
     - If the expression $X_iX_{i+1}...X_jX_{j+1}$ partially matches the head of the rule R_i, add i into R^C and proceed to step 3.

3) If there exists a rule in R^L that has not been previously selected, go back to step 2.

4) If R^c is empty and S^c is not minus 1, this means that there is no fully matched rule for "C_1" and that the expression $X_iX_{i+1}...X_jX_{j+1}$ does not match the head of the rule R_i. Reduce "C_1" by evaluating the body of the rule whose index is S^L. Update "C_1" with its evaluation result, recompose the children of the coordinator "C" by using the proper method, and go to step 6.
5) If R^c is empty and S^c is minus 1, report error and stop the evaluation.
6) If R^c is not empty, then the Coordinator Tree of "C" is a compound object with the MRS R^c and the *selectedRuleIndex* S^c.

Method D

Method D is chosen when both of the children of the coordinator are compound objects. In Figure 21, the coordinator "C" has two children, "C_1" and "C_2". Both of the children are compound objects.

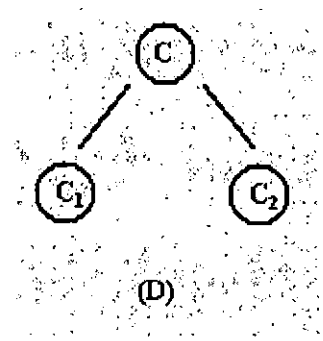


Figure 21. "C_1" and "C_2" are both compound objects

In the above figure, "C_2" is the right child of "C" and is marked for sure, so that the selectedRuleIndex of C_2 is minus 1. Recall from Method C that the selectedRuleIndex of the right child of C is minus 1. Thus, the method D is similar to Method C. The difference is that, in step 2 of Method C, we select those rules that are in the intersection of the MRSs of "C_1" and "C_2" instead of the MRS of "C_1".

When the last coordinator on the abstract syntax tree has been composed, the composition procedure is completed, and the abstract syntax tree is in the normal form. The result of the composition process is the output after composing the last coordinator.

Corresponding to each rule saved in the MRS, there is an entry with the following three fields: *reference number*, *start position,* and *matched length.* Consider a rule R_i with the head $<T_1, T_2, \ldots, T_n>$. Its index is in the MRS of the compound object $X_jX_{j+1}\ldots X_k$. The entry of rule R_i in the MRS is shown in the following table:

Reference Number	**Start Position**	**Matched Length**
i	**j**	**k-j+1**

The conditions that make a compound object a receiver, a sender, or a sender-receiver are as follows:

1) If the Start Position j equals 0, the compound object can act as a receiver for the rule R_i.

2) If the sum of the Start Position j and the Matched Length "k+j+1"equals the length of the head of the rule R_i head, then the compound object can act as a sender for the rule R_i.
3) Otherwise, the compound object can act as a sender-receiver for the rule R_i.

Furthermore, during the object composition procedure, if conditions 1) and 2) are both satisfied at the same time, then the rule R_i is selected to be the reduction rule.

4.7 An Example of Woozle Tree Reduction

We show here how to evaluate an expression using the one-level-delay tree reduction algorithm.

Assume there are four entries in the Woozle system's URT shown as follows:

Index	Head	Body
0	**<"integer", "+", "integer">**	Function *"IntPlusInt"*
1	**<"integer", "+", "float">**	Function *"IntPlusFlo"*
2	**<"integer", "*", "integer">**	Function *"IntmultipleInt"*
3	**<"integer", "*", "float">**	Function *"IntMultipleFlo"*

Consider the expression "1+2*". Its abstract syntax tree is shown in Figure 22. The composing order of the coordinators is

$C_1 \rightarrow C_2 \rightarrow C_3$

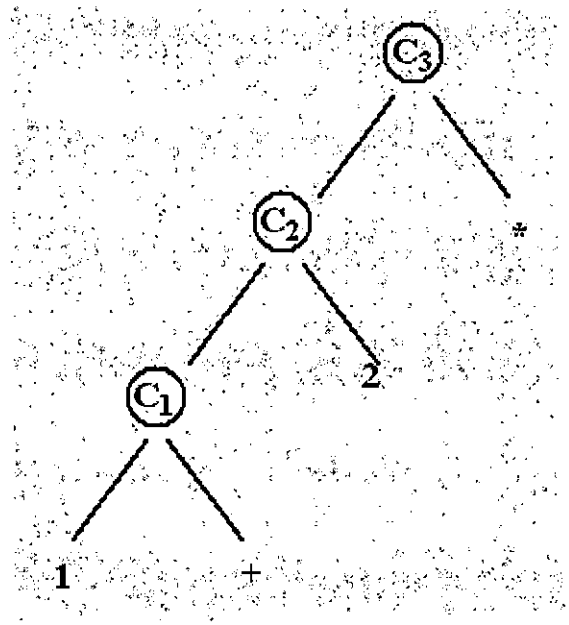


Figure 22. The abstract syntax tree of the expression "1+2*".

When composing the Coordinator Tree of "C_1", Method A is applied because both children of "C_1" are single objects. As there is no rule in the URT whose head has been fully matched by the sub-expression "1+" with the type sequence <"integer", "+">, the Coordinator Tree of "C_1" cannot be reduced. Therefore, the Coordinator Tree of "C_1"are composed to form the compound object "1+" with the MRS R^{C1} as shown in the following Table:

Reference Number	Start Position	Matched Length
0	1	2
1	1	2

Furthermore, the selectedRuleIndex of "C_1" is minus 1.

When composing the Coordinator Tree of "C_2", Method C is applied because the left child of "C_2" is a compound object and the right child of "C_2" is a single object. Find the rules whose indexes are contained in R^{C1} and the heads of those rules is matched by the sub-expression "1+2" with the type sequence <"integer", "+", "integer">. Set the *selectedRuleIndex* of "C_2" to 0 since the rule R_0 has been fully matched. The generated R^{C2} is empty as shown in the following Table:

Reference Number	Start Position	Matched Length

Because "C_2" is not marked, the reduction of the Coordinator Tree of "C_2" is delayed to its parent level, which is "C_3". The shape of the tree remains the same.

When composing the Coordinator Tree of "C_3", we still use Method C. Since R^{C2} is empty, there is no need to do rule matching. According to step 4 of Method C, the *selectedRuleIndex* of C_2 is 0 instead of minus 1. Hence, the body of the rule R_0 is evaluated. "C2" is updated with the resulting object 3.(see Figure 23). The Coordinator Tree of "C_3" is then recomposed using Method A. The re-composition result is the compound object "3*". Its MRS is shown in the following table:

Reference Number	Start Position	Matched Length
2	1	2
3	1	2

Furthermore, the *selectedRuleIndex* of "C_3" is minus 1.

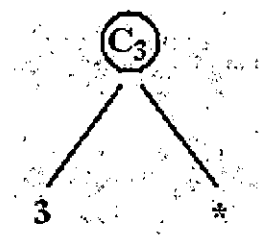


Figure 23. The tree of the expression "3*"

The updated abstract syntax tree (shown in Figure 23) has reached its Normal Form. The evaluation procedure for the expression "1+2*" is completed. The composition result of the Coordinator Tree of "C_3" is the result of the evaluation, which is the compound object "3*".

4.8 A Comparison between One-Level-Delay Tree Reduction and Graph Reduction

One-level-delay tree reduction, developed to implement Woozle, is derived from graph reduction, an algorithm developed to implement functional programming languages.

Thus, these two algorithms are analogous in the following procedure: building a tree/graph for an expression, unwinding the tree/graph to find the reducible expression, reducing the expression and updating the root of the reducible expression with the reduction result. These two algorithms have the same process of evaluation; that is, they both carry out successive reductions on a graph until the graph reaches its normal form.

However, a functional programming language emphasizes such concepts as functions and function applications, whereas Woozle emphasizes objects and object compositions. The distinction between these two languages results in differences of implementation. More specifically, the major differences lie in the methodologies they respectively use to build abstract syntax trees, find reducible expressions, and reduce reducible expressions.

4.8.1 Building Abstract Syntax Trees

In both algorithms, expressions are represented as trees/graphs. In Graph reduction, the leaves of the tree are constant values, built-in functions, or variable names, and the nodes of the tree are applications. Functions always appear as the left children of nodes and the arguments of the functions appear as the right children of nodes.

In one-level-delay tree reduction, the leaves of the tree are all Woozle single objects. Objects in Woozle can act either as senders or as receivers. This depends not on what kind of objects they are, but on which positions they occupy in object compositions. Therefore, the abstract syntax tree of an expression is built by placing the receivers as the

left children of the tree and the senders as right children of the tree. The nodes of the tree are all coordinators that denote object compositions.

4.8.2 Finding Reducible Expressions

In graph reduction, if a function has been found with enough arguments, then the function application can be carried out; that is, its corresponding expression may be reducible, depending on whether the function is a supercombinator or built-in primitive. Recall that functions always appear as the left children on a tree. Therefore, we can find the function by unwinding the spine of nodes until we reach the left-most child, which is a function. We can find the reducible expression by going back through the spine the number of application nodes equal to the number of arguments the function takes.

In one-level-delay reduction, rules are used to determine the meaning of object compositions. If the head of a rule has been fully matched by an expression, the expression can then be reduced by using the rule as its reduction rule. Therefore, the major task during evaluation is to match the expression against the rules. Each object composition takes only two objects a time and matches them against the rules in the URT. If a rule is found to be the reduction rule, the expression is reducible. Nodes (coordinators) are selected for composition according to the post-order binary tree-traversal algorithm

4.8.3 Reducing Reducible Expressions

In graph reduction, supercombinator applications and built-in primitive applications are reduced in different ways. All supercombinator applications are reducible. They are reduced by instantiating the body of the function. Strict built-in primitive applications are reducible only if all of their arguments have been evaluated. They are reduced by applying one of the system built-in rules.

In one-level-delay tree reduction, the reducible expression will be reduced either if the head of the reduction rule is not covered by any other heads of the rules, or its root has been marked. Otherwise, reduction will be delayed until the reduction of its parent level. The way to reduce the reducible expression is to execute the function, the body of the reduction rule.

Chapter 5

Conclusions and Future Work

The main contribution of this thesis is the presentation of an algorithm, called one-level-delay tree reduction. It is used for evaluating expressions in messageless object-oriented programming languages such as Woozle. In the algorithm, expressions are represented in the form of their abstract syntax trees and are reduced by applying a sequence of reduction rules contained in the URT. The algorithm contains four methods. The coordinator is composed using one of the four methods. The evaluation process will stop when the expression cannot be further reduced.

Woozle was originally conceived as a concise, Object-Oriented Web shell language. It soon became clear that the messageless object-object paradigm was worth exploring in its own right. Our major conclusions are the following.

Smalltalk was very successful in supporting a powerful programming paradigm using a small set of fundamental concepts, namely, class, object and message sends. We have shown that it is possible to reduce this set even further to simply object-to-object transfer. Messages are often helpful for readability, but are subsumed by Woozle's simpler model.

The messageless paradigm provides a natural way to use higher-order objects in an Object-Oriented language. The use of higher-order objects, although not necessary, often leads to more concise programs. In many cases, we can avoid the need for auxiliary classes such as "Factory" or "Strategy" classes [7] by using higher-order objects directly.

It is possible to replace method selection by message name with rule-selection based on the type of the sender. This suggests more complex rule-selection schemes that could include, for example, using the state of the receiver as part of the rule-selection mechanism. These alternatives to using messages seem worth exploring. Furthermore, object-oriented concepts, such as inheritance, are worth investigating under Woozle's messageless object-oriented paradigm.

Woozle has been implemented in Java. The one-level delay tree reduction algorithm was used. Several built-in types are supported, which are integers, strings, lists and a number of literal objects including +, -, *, /. The build-in objects http, dir, and file were supported in the first implementation, but have not yet been implemented in the recent version. User-defined objects are supported.

Although we have not provided a formal proof of correctness of the one-level delay tree reduction algorithm, we have at least verified that it yields the appropriate values using several sample Woozle programs. A correctness proof would be a useful future contribution to the idea of messageless object-oriented paradigm.

The Woozle program is interpreted rather than compiled. Unlike a compiler, which translates a source program once and for all, an interpreter examines the program repeatedly. Therefore, compilation can be more efficient than interpretation for Woozle. However, the repeated examination of the source program by an interpreter allows interpretation to be more flexible than compilation. An interpreter can allow programs to be changed "on the fly". With a compiler, on the other hand, all translation is completed before the target program is run, which prevents the target program from being readily adapted as it runs. In practice, Woozle could be implemented using a combination of the two techniques, which is to design a set of instructions for a Woozle reduction machine. In that case, the Woozle program could first be compiled into the Woozle reduction machine code and then the reduction machine code could be compiled to the target machine code, or interpreted directly. This would be faster than the direct interpretation of Woozle source code. Recall that we introduced graph reduction in chapter 2. The G-machine [16] is a fast implementation of graph reduction. While, the G-code is the intermediate code for the G-machine. By adopting such concept, Woozle programs could be run on any machine that supports the Woozle reduction machine.

At present, the Woozle interpreter is synchronized. For example, when transferring a file from a remote site, the program execution is blocked until the transfer is complete. The performance of the execution will be low if the program is heavily involved in network-related activities. Woozle is designed mainly for Web applications. Since most Web activities (such as file downloading) are slow processes, providing Woozle with the

feature of concurrency (i.e. multithreaded) would greatly improve its network performance.

In summary, Woozle remains in the design and experiment stage. Its messageless object-oriented paradigm worth further exploring. We need to present the correctness proofs of the reduction algorithm used in evaluating Woozle expressions. Future improvements to Woozle could include supporting the concurrency feature, completing built-in objects, and reconstructing the reduction machine.

References

[1] Barrett, W., Bates, R., Gustafson, D., Couch, J., *Compiler Construction: Theory and Practice*, 2nd Edition, Science Research Associates, Inc., 1986.

[2] Bird, R., Wadler, P., *Introduction to Functional Programming*, Prentice Hall International (UK) Ltd, 1988.

[3] Chambers, C., Ungar, D., *Making Pure Object-Oriented Language Practical*, OOPSLA '91 Conference Proceeding, pp. 1-15, 1991.

[4] Cornell, G., Horstmann, C., *Core Java*, SunSoft Press, 1996.

[5] Dugan, B., *Simula and Smalltalk: A Social and Political History*, 1994. Available at http://www.cs.washington.edu/homes/brd/history.htm.

[6] Flanagan, D., *Java in a Nutshell*, O'Reilly & Associates, Inc., 1996.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns. Elements of Object-Oriented Software*, Addison-Wesley Publishing Company, 1995.

[8] Geary, D., Mcclellan, A., *Graphic Java: Mastering the AWT*, SunSoft Press, 1997.

[9] Holmes, J, *Object-Oriented Compiler Construction*, Prentice-Hall, Inc., 1995.

[10] Hopkins, T., Horan, B., *Smalltalk: an introduction to application development using visualworks*, Prentice Hall International (UK) Ltd., 1995.

[11] Ingall, D., *A Simple Technique for Handling Multiple Polymorphism*, OOPSLA '86 Proceedings, pp. 347-349, 1986.

[12] *JavaScript Authoring Guide*, Netscape Communications Corporation, 1996. Available at http://www.jsworld.com/help/tutorials/ns/index.html.

[13] Nierstrasz, O., *A Survey of Object-Oriented Concepts*, Object-Oriented concepts, Databases and Applications, ed. W. Kim and F. Lochovsky, pp. 3-21, ACM Press and Addison-Wesley, 1989.

[14] Nygaard, K., Dahl, O., *The Development of the SIMULA Language*, ACM SIGPLAN Notices, Vol. 13, No. 8, pp. 245-272, 1978.

[15] Pascoe, G., *Elements of Object-Oriented Programming*, IEEE Tutorial: Object-Oriented Computing, Vol. 1, pp. 15-20, 1986.

[16] Peyton-Jones, S., *The Implementation of Functional Programming Languages*, Prentice Hall International (UK) Ltd, 1987.

[17] Peyton-Jones, S., Lester, D., *Implementing Functional Languages: A Tutorial*, Prentice Hall International (UK) Ltd, 1992.

[18] Rentsch, T, *Object Oriented Programming*, IEEE Tutorial: Object-Oriented Computing, Vol. 1, pp. 21-27, 1986.

[19] Schwartz, R., *Learning Perl: an introduction*, O'Reilly & Associates, Inc., 1993.

[20] Sethi, R., *Programming Languages: Concepts & Constructs*, 2nd Edition, Addison-Wesley Publishing Company, 1996.

[21] Stroustrup, B., *The C++ Programming Language*, 2nd Edition, Addison-Wesley Publishing Company, 1991.

[22] *Web Programming Languages*, Object Services and Consulting, Inc., 1996. Available at http://www.objs.com/survery/lang.htm.

Glossary

binary function A function which takes two arguments.

compound object An expression that is not a variable primitive.

nonterminal Stands for a (possibly infinite or empty) string in the terminals. It is used in a language's structural rules.

object A basic unit. Everything except qualifiers and punctuation such as '(' and ',' are objects.

primitive object A literal, like integer 1,2,3 and string "abc". Also called single object

redex A node in a graph that can be reduced.

single object See primitive object.

terminal Any member of .Σ (an alphabet).

unary function A function with takes one argument.

Woozle expression An object, or a Woozle expression composed with another Woozle expression. Note that function application is just another form of Woozle expression.

VITA

Surname: Lin Given Name: Fang

Place of Birth: Shanghai, P. R. China

Educational Institutions Attended:

University of Victoria	1995 to 1997
Nanjing University of Aeronautics and Astronautics	1990-1995

Degrees Awarded:

B.Sc. Nanjing University of Aeronautics and Astronautics	1994

Honours and Awards:

The Victoria Canada-China Friendship Association Bursary	1995
Nanjing University of Aeronautics and Astronautics Scholarship	1990 to1994

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Object-Oriented Programming without Messages

Author: 

FANG LIN
(Name in Block Letters)

March 18, 1998
(Date)