DINO

Language Reference Manual

***Author*: Manu Jain**

Table of Contents

1. Introduction

This manual describes the DINO language, developed by the author as a project for the PLT class of spring 2010 at Columbia University taught by Prof. Stephen Edwards.

This manual is modeled after the C language reference manual, which forms Appendix A of the "The C Programming Language" book by Kernighan and Ritchie.

2. Lexical Conventions

Converting a program written in DINO to executable code is a multi-step process. The first step involves running the scanner over the program, which outputs a sequence of tokens. This is known as lexical transformation.

2.1. Tokens

There are six types of tokens – identifiers, keywords, constants and operators. Tokens are separated by white spaces (blanks, tabs, new-lines). Comments are ignored.

2.2. Comments

A comment starts with the characters ** and ends with the characters **. Comments do not nest.

2.3. Identifiers

An identifier is a sequence of letters and digits, starting with a letter. Case distinctions are ignored.

2.4. Keywords

The following identifiers are reserved as keywords, and may not be used otherwise:

dinosaur	named	do	return	eat	hide
thing	listof	times	function	sleep	attack
define	int	if	property	wakeup	position
create	string	then	tostring	drink	height
zap	while	else	toint	run	length
	empty				

2.5. Constants

Integer and string constants (string literals) are supported.

2.6. Operators

Supported operators are additive operators, multiplicative operators, relational operators, equality operators and assignment operator.

3. Meaning of Identifiers

Identifiers can refer to many different things – tags of types (basic types or derived types), properties of types, functions, and objects or variables of types.

3.1. Basic Types

The fundamental types supported are *empty*, *int*, *string*, *dinosaur* and *thing*.
The type *empty* represents an empty value. It is the type returned by functions that don't return any value.
The type *int* represents signed integer values.
The type *string* represents a sequence of characters. Strings are surrounded by double quotes.
The type *dinosaur* represents a dinosaur. The *dinosaur* type supports properties that can be of any basic type or list of basic type.
The type *thing* is used to represent anything that is not a *dinosaur*. The *thing* type supports properties that can be of any basic type or list of basic type.

3.2. Derived Types

There may be an infinite class of derived types created from the basic types, by creating lists of basic types, and by creating functions that operate on basic types or list of basic types and return either a basic type or a list of basic type.

4. Conversions

Conversion from one type to another is generally not supported, except for conversions between *int* and *string*.
Conversion from type *int* to type *string* is supported through the *tostring* keyword.
Conversion from type *string* to type *int* is supported through the *toint* keyword. When the value of the string doesn't evaluate to an integer, *toint* return value is indeterminate.

5. Expressions

5.1. Primary Expressions

Primary expressions are identifiers and constants.

5.2. Dinosaur and Thing Properties

A *dinosaur* or *thing* property is an expression that contains an object of *dinosaur* or *thing* type, followed by a dot, followed by a property member of that type.

The *dinosaur* or *thing* property expression evaluates to one of the basic types (3.1) excluding the *empty* type, or a list of a basic type.

5.3. Functions

A function is an expression that contains an identifier (that represents an object of *dinosaur* type), followed by an identifier that represents a function name, followed by zero or more arguments.

The *function* expression evaluates to one of the basic types (3.1), or a list of one of the basic type. In other words, a *function* returns a basic type or a list of a basic type.

Each argument of a *function* may be of any basic type, or a list of a basic type, except the *empty* type. A function automatically gets the *dinosaur* object whose identifier precedes the function-name as the first argument.

5.4. Multiplicative Operators

Supported multiplicative operators are multiplication (*) and division (/). The operands of these operators must be of type *int*, and the result is also an *int* type. For division, the result is rounded off to the nearest integer.

5.5. Additive Operators

Supported additive operators are plus (+) and minus (-). The operands of these operators must be of type *int*, and the result is also an *int* type.

5.6. Relational Operators

Supported relational operators are < (less), > (greater), <= (less than or equal) and >= (greater than or equal). The operands of these operators must be of type *int*, and the result is either 0 if condition is false or 1 if condition is true.

Relational operators are only allowed within the *if* conditional statement.

5.7. Equality Operators

The = (equal to) and != (not equal) operators are similar to relational operators, except that they also support comparison of *string* types in addition to *int* types.

Equality operators are only allowed within the *if* conditional statement.

5.8. Assignment Expression

The assignment operator (=) requires a variable or object of a basic type (except the *empty* type) as the left operand, with the right-hand side operand being an expression that evaluates to the same type. As a result of the assignment, the left hand side variable or object takes the value of the evaluated expression on the right-hand side.

6. Definitions

6.1. Function Definition

Function definitions are used to define new functions.

An function definition consists of the keyword *define,* followed by the keyword *function,* followed by the function return-type, followed by an identifier (function name), followed by a list of arguments enclosed in brackets, followed by the function-body.

The function return-type may be any basic type or a list of a basic type.

The identifier in the function definition becomes the function-name of the newly defined function. Function-names must be unique.

Each item in the list of arguments contains a type, which can be any basic type or a list of a basic type, followed by the name of that argument.

The function automatically receives an argument named “dino” that is of type *dinosaur*.

The function body contains zero or more expressions. If the function has a return-type which is not *empty,* the last statement in the function body is a *return* keyword followed by an expression that evaluates to the same type as the function return-type.

6.2. Property Definition

Property definitions are used to define new properties on *dinosaur* or *thing* types.

A property definition consists of the keyword *define,* followed by the keyword *property*, followed by the property type, followed by the keyword *dinosaur* or *thing*, followed by a dot, followed by an identifier (property name), followed by an expression that evaluates to the same type as the property type. The expression in the end is used to provide a default value to the property.

The property type may be any basic type or a list of a basic type.

7. Creation

New objects are instantiated by using the *create* keyword.

7.1. Creating Dinosaurs

New dinosaur objects are created by using the *create* keyword, followed by the *dinosaur* keyword, followed by the *named* keyword, followed by an identifier. The identifier in the end denotes the object name and has to be unique.

7.2. Creating Things

New thing objects are created by using the *create* keyword, followed by the *thing* keyword, followed by the *named* keyword, followed by an identifier. The identifier in the end denotes the object name and has to be unique.

7.3. Creating Lists

Lists are created by using the *create* keyword, followed by the *listof* keyword, followed by one of the basic types.

8. Statements

Statements are executed in sequence. They are of several types.

8.1. Selection Statement

Selection statements may be of two different forms –

if (expression) statement
if (expression) statement *else* statement

In both forms of the *if* statement, if the expression evaluates to non-zero, the first sub-statement is executed. In the second form, the second sub-statement is executed if the expression evaluates to 0.

Nesting *if*s are not permitted.

8.2. Iteration Statement

Iteration statements may be of two different forms –

do expression *times* statement
while (expression) statement

In the *do* statement, the sub-statement is executed as many times as the expression specifies. The expression evaluates to an integer.

In the *while* statement, the sub-statement is executed repeatedly until the value of the expression evaluates to 0.

Nesting *do* or *while* statements are not permitted.

9. External Declarations

External declarations are not supported. All the source code for a DINO program must reside in a single unit of input.