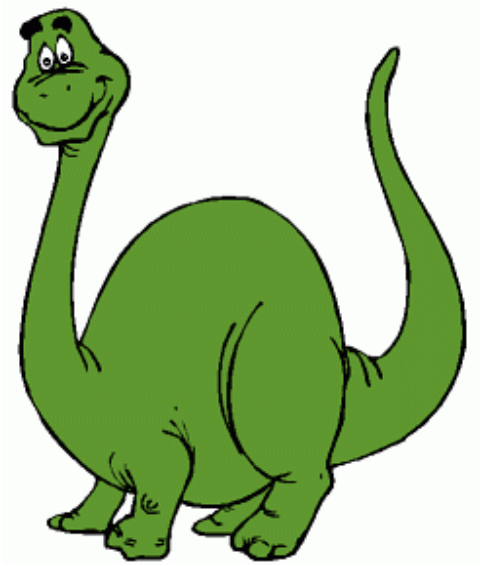
DINO

A Friendly 'Dinosaur' Language for Kids



Final Project Report

Author: **Manu Jain**

Table of Contents

1. Introduction

1.1. What is DINO

DINO is intended to be a fun, easy-to-learn language for kids.

It is created by the author as a project for the PLT class of fall 2010 at Columbia University taught by Prof. Stephen Edwards.

1.2. Purpose

The purpose behind creating DINO is to get kids interested in programming. The language is aimed at getting young kids (ages 8+) interested in programming, teaching them the basics of programming, and making programming fun for them.

1.3. Basic Idea

The basic idea of the language is to give the young programmers simple types like integers and strings to play with, along with just one other type called “dino” whose properties and behavior they can “build” as per their own liking and as they go along.

1.4. Evolution of Concepts

The concepts around how the language should be structured, what its syntax should be and what features it should provide have undergone significant changes since when the project proposal was first given, on to when the language reference manual was created, and through the development phase. The language started out looking more VB-like, and ended up looking more C-like!

2. Language Tutorial

2.1. Basics

DINO programs look like simplified C programs, albeit with a few differences.

DINO supports only four types – bool, integer, string and dino. It has support for built-in expressions that operate on one or more of those types.

DINO provides the ability to create C-like functions. It also provides the ability to define *methods*, which add behavior to the dino type and *properties*, which add data to the dino type.

The main entry point of a DINO program is the *main* function.

2.2. A Simple Program

A simple DINO program can consist of just the main function. An example is the “hello world” program below –

```
main( )
```

```
{
        print( “hello world!”);
}
```

Another way the hello world program may be written is by using a string variable –

```
main( )
{
        string hello;
        hello = “hello world!”
        print( hello);
}
```

2.3. *A More Complex Program*

A typical DINO program will have methods and properties for the dino type defined preceding the main function, followed by a main method.

```
/*
 Properties defined on the dino type.
These add data members to each dino object,
with default values assigned at object creation time.
*/
property int Height;
property string Name;

/*
Methods defined on the dino type.
These add behavior to each dino object.
*/
method bool IsSameHeight(dino saurus)
{
        if(me.Height == dino.Height)
        {
                return true;
        {
        else
        {
                return false;
        }
}

main( )
{
        dino trex;
        trex.Height = 50;
        trex.Name = “T-Rex”;
        dino sauropod;
        sauropod.H/eight = 100;
        saurpod.Name = “Sauropod”;
```

```
        print(tostring(trex.IsSameHeight(sauropod)));
    }
```

3. Language Manual

3.1. 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.

3.2. Syntax Notation

The syntax in this document is written in a variant of Extended Backus-Norm Form (EBNF), using regular expression repetition operators.

3.3. The Big Picture

DINO programs are written in a single source file. DINO source files have ".dino" file extension.

The entry point of a DINO program is the 'main' function. Functions, methods and properties are defined outside and preceding the 'main' function.

Statements in DINO may appear inside any function or method definition, and in *main*.

A DINO program has the following high-level structure:

```
[function definition]*
[method definition]*
[property definition]*

main program =
main, '(', ')', '{'
[expression, ';']* [white-space]*
[statement, ';']* [white-space]*
'}'
```

3.4. Language Rules

DINO uses static scoping.

It evaluates expressions and parameters from left-to-right, i.e. it is left-associative.

It follows applicative-order argument evaluation, and thus evaluates parameters before executing the body of the function.

It performs short-circuit evaluation, evaluating the body of statements and operands of operators if needed.

It follows normal operator precedence rules.

3.5. 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.

3.5.1. Tokens

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

```
token = comment | identifier | keyword | constant | operator
```

3.5.2. Comments

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

```
comment = /* [ascii character]*  */
```

3.5.3. Identifiers

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

```
identifier = (‘a’-‘z’  ‘A’- ‘Z’) [‘a’-‘z’  ‘A’-‘Z’ ‘0’-‘9’]*
```

3.5.4. Keywords

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

int	do	while	main
listof	times	method	me
dino	if	property	return
string	else	tostring	nothing

3.5.5. Constants

Integer and string constants (string literals) are supported.

```
constant =      [‘0’ – ‘9’]+                                    ** int constant **
                |    “, [‘a’-‘z’  ‘A’-‘Z’ ‘0’-‘9’]+ , “         ** string constant **
```

3.5.6. Operators

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

3.6. Meaning of Identifiers

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

```
identifier =    tags of basic type
                | tags of derived type
                | function-name
                | property-name
                | variable-name
```

3.6.1. Basic Types

The fundamental types supported are *nothing*, *bool*, *int*, *string*, *dinosaur*.
The type *nothing* represents an empty value. It is the type returned by functions and methods 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 *dino* represents a dinosaur.

```
basic type =    nothing | bool| int | string | dino
```

3.6.2. 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.

```
derived type = list<basic-type>
             |  function<basic-type>
             |  function<list<basic-type>>
             |  property <basic-type>
             |  property<list<basic-type>>
```

3.7. *Conversions*

Conversion from one type to another is generally not supported. Integers can be converted to a string literal through the *tostring* method.

3.8. *Expressions*

```
expression =   identifier | constant              ** primary expression **
             | function call
             | dino method call
             | property accessor
             | multiplicative operator
             | additive operator
             | increment and decrement operator
             | relational operator
             | equality operator
             | assignment expression
```

3.8.1. Primary Expressions

Primary expressions are identifiers and constants.

```
primary expression =  identifier | constant
```

3.8.2. Function Call

A function call is an expression that contains an identifier that represents a defined function, followed by zero or more arguments.

```
function call =          function-name,  '(',  [expression, ',']*,  ')'
```

A function call evaluates to one of the basic types, or a list of one of the basic type. In other words, a function call returns a basic type or a list of a basic type.

Each argument of a function call may be an expression that evaluates to any basic type, or a list of a basic type.

3.8.3. Dino Method Call

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

```
method call =        variable-name, '.',
                     method-name, '(', [expression, ',']*, ')'
```

A method call evaluates to one of the basic types, or a list of one of the basic type. In other words, a method call returns a basic type or a list of a basic type.

Each argument of a method call may be an expression that evaluates to any basic type, or a list of a basic type. When a function is called, it automatically gets the *dino* object on which it is called as the first argument. This object is called *me* within the function body.

3.8.4. Dino Property Accessor

A property accessor is an expression that contains an identifier (that represents an object of *dinosaur* type), followed by a dot, followed by one of the defined properties.

```
property accessor =  variable-name, '.',  property-name
```

A property accessor expression evaluates to one of the basic types excluding the *nothing* type, or a list of a basic type.

3.8.5. 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.

```
multiplicative =     operand, white-space*, ('*' | '/'), white-space*, operand
```

3.8.6. 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.

```
additive =     operand, white-space*, ('+' | '-'), white-space*, operand
```

3.8.7. Increment and Decrement Operators

Supported increment operator is ++ and decrement operator is --. The operands of these operators must be of type *int*, and the result is also an *int* type.

```
increment =     operand, “++”
decrement =     operand, “—“
```

3.8.8. 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.

```
less than =       operand,  white-space*,  ‘<’,  white-space*, operand
less than or equal =  operand,  white-space*,  “<=”,  white-space*, operand

greater than =  operand,  white-space*,  ‘>’,  white-space*, operand
greater than or equal =  operand,  white-space*,  “>=”,  white-space*, operand
```

Relational operators are only allowed within the *if* and *while* expressions.

3.8.9. 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.

```
equal to =       operand,  white-space*,  “==”,  white-space*, operand
not equal to =  operand,  white-space*,  “!=”,  white-space*, operand
```

Equality operators are only allowed within the *if* and *while* expressions.

3.8.10. Assignment Expression

The assignment operator (=) requires a variable or object of a basic type (except the *nothing* type) or a property as the left operand, with the right-hand side operand being an expression that evaluates to the same type.

```
assignment =
identifier, white-space*, ‘=’, white-space*, expression
```

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.

3.9. Definitions

3.9.1. Function Definition

function definition =
return-type, white-space+,
function-name, white-space+,
‘(‘, white-space+,[argument, ‘,’, white-space*]*,‘)’,
‘{‘,
[expression]*, ‘;’, white-space*
[statement]*, ‘;’, white-space*
return, white-space+, expression, ‘;’, white-space*
‘}’

Function definitions are used to define new functions.

An function definition consists of 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.

3.9.2. Dino Method Definition

method definition =
method, white-space+,
return-type, white-space+,
method-name, white-space+,
‘(‘, white-space+,[argument, ‘,’, white-space*]*,‘)’, white-space+,
‘{‘
[expression]*, ‘;’, white-space*
[statement]*, ‘;’, white-space*
return, white-space+, expression, ‘;’, white-space*
‘}’

Method definitions are used to define new methods on the dino type.

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

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

The identifier in the method definition becomes the function-name of the newly defined method. A method-name must be unique among all methods and properties.

3.9.3. Property Definition

property definition =
property, white-space+,
type, white-space+,
property-name, ';'

Property definitions are used to define new properties (data members) on the *dino* type.

A property definition consists of the keyword *property*, followed by the property type, followed by an identifier (property name), followed by a semicolon.

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

3.10. Variable Creation

New variables are created by writing the type followed by an identifier.

3.10.1. Creating Integers

New integers are created by using the *int* keyword, followed by an identifier, followed by a semicolon.

integer creation = *int*, white-space+, identifier, ';'

3.10.2. Creating Strings

New strings are created by using the *string* keyword, followed by an identifier, followed by a semicolon.

string creation = *string* white-space+, identifier, ';'

3.10.3. Creating Dinosaurs

New dinosaur objects are created by using the *dino* keyword, followed by the identifier.

dino creation = *dino*, white-space+, identifier, ';'

3.10.4. Creating Lists

Lists are created by using the *listof* keyword.

list creation =
listof, ‘(‘,
basic-type, ‘)', white-space+,
identifier, ‘;’

3.11. *Statements*

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

```
statement =   expression
            | if statement
            | if-else statement
            | do-times statement
            | while statement
```

3.11.1. Selection Statement

Selection statements may be of two different forms –

if statement =
if, white-space*, ‘(‘, expression, ‘)’, white-space*,
‘{‘
    sub-statement
‘}’

if-else statement =
if, white-space*, ‘(‘, expression, ‘)’, white-space*,
‘{‘
    sub-statement
‘}’
else, white-space+ ,
‘{‘
    sub-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.

3.11.2. Iteration Statement

Iteration statements may be of two different forms –

do-times statement =
do, white-space*, '(', expression, ')', white-space*, *times*
'{'
sub-statement
'}'

while statement =
while, white-space*, '(', expression, ')', white-space*,
'{'
sub-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.

3.12. External Declarations

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

4. Project Plan

4.1. Process

I worked alone on this project, developing the different pieces in an ad-hoc manner, as time permitted.

In terms of software development process, I initially followed the water-fall process, which turned out to be a mistake (as waterfall always is!). I backtracked and started following the iterative development process, where I had more success.

4.2. Programming Style

Since I used the MicroC project as the starting point, I used its coding style.

4.3. Show your project timeline

Concept Phase (Sept 2010):
I started out by thinking of a concept for the language that would be interesting for me. I came up with the DINO language because I have young kids and I want to get them interested in programming. The existing languages seem too complex, with many types and a lot of features, for a young kid. Therefore I thought it would be interesting if I could develop a language which I could use to teach my kids the basics of programming.

Initial Proposal Phase (Sept 2010)
Once I decided on the concept, the next step was deciding the structure of the language, the syntax and the grammar. This was interesting since I struggled to make the language easy to use, yet at the same time flexible and powerful enough to be useful and allow programmers to create "building blocks".

LRM Phase (Oct – Nov 2010)
Next step was creating the Language Reference Manual (LRM). This required me to think more realistically about the language and its syntax. I revised my initial proposal and made several significant changes. As an example, I took out the support for custom types derived from dinosaur.

Scanner – Parser (Nov 2010)
After the LRM, I started on the scanner and parser. As I developed these pieces, I realized that my project proposal needed even more revisions. I again made significant changes to my language proposal, for example taking out support for the 'thing' type, removing the 'end' keyword and replacing it with parenthesis, etc.

Translator (Dec 2010)
The last step was creating the complete translator. For this, I backtracked and started afresh, using the MircoC translator as the base, and making one change at a time. This was the most time-consuming part of the project and after some time, I had to drop supporting the compiler and byte-code generator since I simply ran out of time. I developed test cases in parallel with the development effort.

Final Project Report (Dec 2010)
The last step was creating the final project report, although the development went on in parallel, and is still continuing! I am continuing to face difficulties in developing the interpreter.

4.4. Roles and Responsibilities

Since I worked alone on this project, I took on all roles and responsibilities.

4.5. Software Development Environment

I used the Eclipse IDE, with the OcaIDE plugin for OCaml development (http://www.algo-prog.info/ocaide/).

The OCamlLex Lexical Analyzer was used to generate the lexical analysis module.

The OCamlYacc Syntactical Analyzer was used to generate the syntactical analysis module.

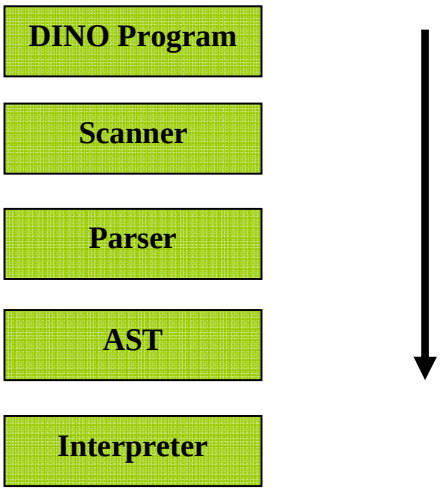
The OCaml compiler was used to generate modules for other source files.

The version of OCaml used for development was 3.12.

5. Architectural Design

5.1. Major Components

I could not continue development of the compiler and byte-code generator, due to lack of time. The major components of the final product are –



5.2. Developers

Since I worked alone on this project, I developed all the components.

6. Test Plan

I developed multiple test cases, each one to test different features of the language. The test cases were run manually.

6.1. Hello World Test with String Literal

This test case tested that the simple “Hello World” program was translated successfully and the output was as expected.

Test Program:

```
main( )
{
        print( “hello world!”);
}
```

Expected Output:

```
hello world!
```

6.2. Hello World Test with String Variable

This test case tested that the translator processed string variables successfully.

Test Program:

```
main( )
{
        string hello;
        hello = “hello world!”;
        print( hello);
}
```

Expected Output:

```
hello world!
```

6.3. Integer Operators Test

This test case tests that the integer binary and unary operators work as expected.

Test Program:

```
main( )
{
        int x;
        int y;

        x = 2;
        y = 3;

        /* test addition */
        y = x + y;
        print( tostring(y));

        /* test subtraction */
        y = y – x;
        print( tostring(y));

        /* test multiplication */
        y = y * x;
        print( tostring(y));

        /* test division */
        y = y/x;
        print( tostring(y));

        /* test post-increment */
        y = y++;
        print( tostring(y));

        /* test post-decrement */
        y = y--;
```

```
        print( tostring(y));

        /* test greater-than */
        if(x > y)
        {
                print( tostring(x));
        }
        else
        {
                print( tostring(y));
        }

        /* test less-than */
        if(x < y)
        {
                print( tostring(x));
        }
        else
        {
                print( tostring(y));
        }

        /* test equals */
        if(x == y)
                print( “x == y”);

        /* test not-equal-to */
        if(x != y)
                print( “x != y”);
}
```

Expected Output:

```
5
3
6
3
4
3
3
2
x != y
```

6.4. Simple Dino Test

This test case tests whether the translator processes dino types and dino properties successfully.

Test Program:

```
property string Name;

main( )
{
        dino trex;
        trex.Name = “T-Rex”;

        print(tostring(trex.Name));
}
```

Expected Output:

T-Rex

6.5. Complex Dino Test

This test case tests whether the translator processes dino types, dino methods and dino properties successfully.

Test Program:

```
property int Height;
property string Name;

method int IsSameHeight(dino saurus)
{
        if(me.Height == dino.Height)
        {
                return 1;
        {
        else
        {
                return 0;
        }
}

main( )
{
        dino trex;
        trex.Height = 50;
        trex.Name = “T-Rex”;
        dino sauropod;
        sauropod.H/eight = 100;
        saurpod.Name = “Sauropod”;

        print(tostring(trex.Name));
        print(tostring(sauropod.Name));
```

```
        print(tostring(trex.IsSameHeight(sauropod)));
}
```

Expected Output:

```
T-Rex
Sauropod
0
```

6.6. Do-Times Test

This test case tests whether the translator processes the do-times statement successfully.

Test Program:

```
main( )
{
        int i ;

        do 4 times
        {
                i = i + 1;
                print(tostring(i));
        }
}
```

Expected Output:

```
1
2
3
4
```

7. Lessons Learned

I learned a lot of lessons while doing this project. Some of them are given below.

7.1. Beneath the Covers

One thing the PLT class and this project taught me, and for which I am grateful, is what lies under the cover of a language. This is one aspect of software development about which I had never given much thought.

I have learned that beneath the covers, most languages are very similar. They have the same building blocks, and are even built using the same set of tools.

7.2. It's not Easy to Make it Simple

I tried to design a language that is simple and English-like to some extent, since it is targeted towards children.

I discovered the hard-way that making the grammar unambiguous for the translator takes some doing. Those ugly parentheses are there for a purpose! It takes a lot of time and effort to make a language that is English-like and easy to write and read, and yet unambiguous. In the end, I had to make changes to the syntax of DINO to make it more C-like.

7.3. OCaml Really is Different!

One thing about which I was very proud of before taking the class was my ability to pick up any language, learn it quickly, and become productive in it in no time. I have previously learned and used FORTRAN, PASCAL, COBOL, C, C++, BASIC, Visual Basic, C#, Java, and a few others.

OCaml threw me off since it was so different from any other language I had learned. It took me some time to get used to its syntax and organization. Even though I learned to understand it and write a few lines in it, I never grew comfortable with it.

I find that even today, I can't "think" in OCaml. When trying to add a feature to my language, I think about how I would do it in C# or Java, and it's trivial. But attempting to do the same in OCaml consumes a lot of time, and in many cases I find that my proficiency in OCaml is just not enough and I fail to achieve what I set out for.

In retrospect, I should have spent much more time with the language to make the project a success.

7.4. Be Less Ambitious

I should have understood my limitations of time (full-time job, two courses during fall 2010 semester at Columbia, and family commitments), resources (I am working alone) and unfamiliarity with OCaml and the development tools, and chosen a less ambitious project idea.

For a long time, I had the confidence that I would eventually master OCaml and would become as productive in it as I am in C# or Java, but that didn't happen. I found that I couldn't achieve even simple things without spending a lot of time and effort.

I should have been less ambitious.

7.5. Develop All Parts Together Iteratively

I started with the MicroC project as my base.

I made the mistake of developing only the scanner and parser first. I spent a lot of time on it, refining and changing my ideas as I went along.

By the time I came to the interpreter and compiler, I found that I had completely broken them. I got a ton of errors that I just couldn't resolve.

This meant that I had to start over afresh. I made one small change at a time, but in all the pieces, so that the complete project kept compiling all the time.

But I had lost a lot of time because I raced ahead with the scanner and parser without taking the other parts along.

7.6. Working Alone is Boring

One thing I sorely missed in this project was a partner.

I don't think I understood the reason why CVN students had to do the project alone. In contrast, in the other class I took this semester, four CVN students sitting in different parts of the country completed the group project successfully and without any major issues.

In this project, not having a partner made developing the language that much more boring and also increased the work load. I found that I was not as committed or motivated as when doing a project with partners.

At the least, choosing partners could be made optional.

8. Appendix – Code Listing

8.1. Scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| '.'      { DOT }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| "++"     { INCR }
| "--"     { DECR }
```

```
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "do"     { DO }
| "times"  { TIMES }
| "return" { RETURN }
| "int"    { INT }
| "string" { STRING }
| "dino"   { DINO }
| "method" { METHOD }
| "property" { PROPERTY }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| '"'['a'-'z' 'A'-'Z' '0'-'9' '_']*'"' as lxm { STR_LITERAL (lxm) } (*
| '"'[^'"']*'"' as lxm {STR_LITERAL (lxm) } *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.2. *Parser.mly*

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token INCR DECR
%token RETURN IF ELSE FOR WHILE DO TIMES
%token INT STRING DINO
%token METHOD PROPERTY
%token <int> LITERAL
%token <string> STR_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left INCR DECR
```

```
%start program
%type <Ast.program> program

%%
/* a program consists of lists of variables, function-declrations,
method-declarations, property-declarations, in that order */
program:
   /* nothing */ { [], [], [], [] }
 | program vdecl { ($2 :: (fun(a,b,c,d) -> a) $1), (fun(a,b,c,d) -> b)
$1, (fun(a,b,c,d) -> c) $1, (fun(a,b,c,d) -> d) $1 }
 | program fdecl { (fun(a,b,c,d) -> a) $1, ($2 :: (fun(a,b,c,d) -> b)
$1), (fun(a,b,c,d) -> c) $1, (fun(a,b,c,d) -> d) $1 }
 | program mdecl { (fun(a,b,c,d) -> a) $1,  (fun(a,b,c,d) -> b) $1,
($2 :: (fun(a,b,c,d) -> c) $1), (fun(a,b,c,d) -> d) $1 }
 | program pdecl { (fun(a,b,c,d) -> a) $1,  (fun(a,b,c,d) -> b) $1,
(fun(a,b,c,d) -> c) $1,  ($2 :: (fun(a,b,c,d) -> d) $1) }

fdecl:
   ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $1;
       formals = $3;
       locals = List.rev $6;
       body = List.rev $7 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    formal_decl                   { [$1] }
  | formal_list COMMA formal_decl { $3 :: $1 }

formal_decl:
                  INT ID { Formal_Int($2) }
            | DINO ID { Formal_Dino($2) }
            | STRING ID { Formal_String($2) }

/* dino method declaration */
mdecl:
   METHOD ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
RBRACE
     { { mname = $2;
       mformals = $4;
       mlocals = List.rev $7;
       mbody = List.rev $8 } }

      /* dino property declaration consists of a type and identifier*/
      /* TODO - replace first ID with type */
pdecl:
   PROPERTY ID ID SEMI
     { { ptype = $2;
       pname = $3;  } }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }
```

```
vdecl:
    INT ID SEMI { Var_Int($2) }
      | STRING ID SEMI { Var_String($2) }
      | DINO ID SEMI { Var_Dino($2) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
      | DO expr TIMES stmt { Do($2, $4) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
      | STR_LITERAL      { Str_Literal($1) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
      | expr INCR        { Unaryop($1, Incr) }
      | expr DECR        { Unaryop($1, Decr) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
      | ID DOT ID LPAREN actuals_opt RPAREN { MethodCall($1, $3, $5) }
/* dino method call */
      | ID DOT ID  { Get($1, $3) } /* dino property get */
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.3. *AST.ml*

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater |
Geq

type unary_op = Incr | Decr

type var =
      Var_Int of string
      | Var_String of string
      | Var_Dino of string

type expr =
    Literal of int
      | Str_Literal of string
  | Id of string
  | Binop of expr * op * expr
  | Unaryop of expr * unary_op
  | Assign of string * expr
  | Call of string * expr list
      | MethodCall of string * string * expr list
      | Get of string * string
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
      | Do of expr * stmt

type formal =
                  Formal_Int of string
            | Formal_String of string
            | Formal_Dino of string

type func_decl = {
    fname : string;
    formals : formal list;
    locals : var list;
    body : stmt list;
  }

type method_decl = {
    mname : string;
    mformals : formal list;
    mlocals : var list;
    mbody : stmt list;
  }

type property_decl = {
    ptype : string;
    pname : string;
  }
```

```
type program = var list * func_decl list * method_decl list *
property_decl list

let rec string_of_expr = function
    Literal(l) -> string_of_int l
      | Str_Literal(s) -> s
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " "
^
      string_of_expr e2
      | Unaryop(e1, o) ->
      string_of_expr e1 ^
      (match o with
      Incr -> "++" | Decr -> "--")
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
      | MethodCall(o, f, el) ->
              o ^ "." ^ f ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
      | Get(o, p) ->
            o ^ "." ^ p
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; "
^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt
s
  | Do(e, s) -> "do " ^ string_of_expr e ^ " times " ^ string_of_stmt s

let string_of_vdecl = function
      Var_Int(s) -> "int " ^ s ^ ";\n"
  | Var_String(s) -> "string " ^ s  ^ ";\n"
      | Var_Dino(s)  -> "dino " ^ s ^ ";\n"

let string_of_formaldecl = function
      Formal_Int(s) -> "int " ^ s
  | Formal_String(s) -> "string " ^ s
      | Formal_Dino(s)  -> "dino " ^ s

let string_of_formal = function
```

```
      Formal_Int(s) ->  s
  | Formal_String(s) -> "string " ^ s
      | Formal_Dino(s)  -> s

let string_of_fdecl fdecl =
      fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_formaldecl fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_mdecl mdecl =
      mdecl.mname ^ "(" ^ String.concat ", " (List.map
string_of_formaldecl mdecl.mformals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl mdecl.mlocals) ^
  String.concat "" (List.map string_of_stmt mdecl.mbody) ^
  "}\n"

let string_of_pdecl pdecl =
      pdecl.ptype ^ " " ^ pdecl.pname ^ ";\n"

let string_of_program (vars, funcs, methods, properties) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs) ^ "\n" ^
      String.concat "\n" (List.map string_of_mdecl methods) ^ "\n" ^
      String.concat "\n" (List.map string_of_pdecl properties) ^ "\n"
```

8.4. *Interpret.ml*

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of int * int NameMap.t

(* Main entry point: run a program *)

let run (vars, funcs, methods, properties) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
      in

  (* Put method declarations in a symbol table *)
  let method_decls = List.fold_left
      (fun methods mdecl -> NameMap.add mdecl.mname mdecl methods)
      NameMap.empty methods
  in

        (* Put property declarations in a symbol table *)
  let prop_decls = List.fold_left
      (fun properties pdecl -> NameMap.add pdecl.pname pdecl properties)
```

```
      NameMap.empty properties
  in

  (* Invoke a function and return an updated global symbol table *)
      (* TODO: change code below to treat 'globals' parameter as
Ast.var list instead of NameMap.t *)
  let rec call fdecl actuals globals methods properties =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
                        Literal(i) -> i, env
                  | Str_Literal(s) -> 1, env (* TODO: add support for
string type *)
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate
*)
      | Id(var) ->
        let locals, globals = env in
        if NameMap.mem var locals then
          (NameMap.find var locals), env
        else if NameMap.mem var globals then
          (NameMap.find var globals), env
        else raise (Failure ("undeclared identifier " ^ var))
      | Binop(e1, op, e2) ->
        let v1, env = eval env e1 in
          let v2, env = eval env e2 in
        let boolean i = if i then 1 else 0 in
        (match op with
          Add -> v1 + v2
        | Sub -> v1 - v2
        | Mult -> v1 * v2
        | Div -> v1 / v2
        | Equal -> boolean (v1 = v2)
        | Neq -> boolean (v1 != v2)
        | Less -> boolean (v1 < v2)
        | Leq -> boolean (v1 <= v2)
        | Greater -> boolean (v1 > v2)
        | Geq -> boolean (v1 >= v2)), env
                  | Unaryop(e, op) ->
        let v1, env = eval env e in
        (match op with
          Incr -> v1 + 1
        | Decr -> v1 - 1), env
      | Assign(var, e) ->
        let v, (locals, globals) = eval env e in
        if NameMap.mem var locals then
          v, (NameMap.add var v locals, globals)
        else if NameMap.mem var globals then
          v, (locals, NameMap.add var v globals)
        else raise (Failure ("undeclared identifier " ^ var))
      | Call("print", [e]) ->
        let v, env = eval env e in
        print_endline (string_of_int v); (* TODO: modify print to
support all types *)
        0, env
            | MethodCall(o, f, actuals) -> 0, env
            (* TODO: Implement case for MethodCall - a method should
get a diff. env. than a functional call. *)
```

```
                        (* For each dino object, need to maintain a
list of properties and their values. *)
                        (* A method should be given the list of
properties of the dino object on which it is called('o'). *)
                        (* A dino method should also be able to access
all globals *)
                    | Get(o, p) -> 0, env
                        (* TODO: Implement case for Property Get. *)
                        (* Maintain a list of properties and their
values for each dino object. *)
                        (* When a property 'Get' call is made, return
the value of that property for the given dino object 'o' *)
      | Call(f, actuals) ->
        let fdecl =
          try NameMap.find f func_decls
          with Not_found -> raise (Failure ("undefined function " ^ f))
        in
        let actuals, env = List.fold_left
            (fun (actuals, env) actual ->
            let v, env = eval env actual in v :: actuals, env)
            ([], env) (List.rev actuals)
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals methods properties
          in 0, (locals, globals)
        with ReturnException(v, globals) -> v, (locals, globals)
    in

    (* Execute a statement and return an updated environment *)
    let rec exec env = function
      Block(stmts) -> List.fold_left exec env stmts
      | Expr(e) -> let _, env = eval env e in env
      | If(e, s1, s2) ->
        let v, env = eval env e in
        exec env (if v != 0 then s1 else s2)
      | While(e, s) ->
        let rec loop env =
          let v, env = eval env e in
          if v != 0 then loop (exec env s) else env
        in loop env
      | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
          let v, env = eval env e2 in
          if v != 0 then
            let _, env = eval (exec env s) e3 in
            loop env
          else
            env
        in loop env
            (* implementation of do-times logic *)
            | Do(e, s) ->
            let i, env = eval env e in
                  let rec loop j env =
            if i > 0 then
                              loop (i-1) (exec env s)
```

```
                        else
                                env
                    in loop i env
      | Return(e) ->
        let v, (locals, globals) = eval env e in
        raise (ReturnException(v, globals))
    in

    (* Enter the function: bind actual values to formal arguments *)
    let locals =
                    (* iterate over 2 lists (formals, actuals), and bind
each formal argument to its value *)
      try List.fold_left2
        (fun locals formal actual -> NameMap.add (string_of_formal
formal) actual locals)
        NameMap.empty fdecl.formals actuals
      with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments passed to " ^
fdecl.fname))
    in
    (* Execute each statement in sequence, return updated global symbol
table *)
    snd (List.fold_left exec (locals, globals) fdecl.body)

  (* Run a program: find and run "main" *)
  in try
    call (NameMap.find "main" func_decls) [] vars methods properties
     with Not_found -> raise (Failure ("did not find the main()
   function"))
```

8.5. *Dino.ml*

```
type action = Ast | Interpret

let _ =
  let action =
      if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);

      ("-i", Interpret)]
  else Interpret in
      let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
     | Interpret -> ignore (Interpret.run program)
```