EasyPet
Language Reference Manual

Team 13:

Bo Liu	bl2400
Xiaolong Jiang	xj2127
Liang Wei	lw2425
Shen Wang	sw2613
Weikang Wan	ww2267

REVISION HISTORY				
Revision	**Version**	**Description of Change**	**Changed By**	**Effective Date**
1	0.1a		Team 13	03/22/11
2				
3				

Table of Content

Language Reference Manual

1. Introduction

This manual describes the EasyPet language designed by PLT Team 13 (Weikang Wan, Liang Wei, Shen Wang, Xiaolong Jiang and Bo Liu). This language is the course project of COMS W4115 (Programming Language & Translators).

EasyPet is a language intending to facilitate the implementation of electronic pet programs, which mainly involves GUI design, item description, items interaction and intensive event-driven programming. EasyPet is designed to make these steps convenient.

2. Lexical Conventions

2.1 Tokens

Tokens of EasyPet include keywords, operators, identifiers, constants, string literals and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds and comments as described below (collectively, ``white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

2.2 Comments

EasyPet provides two ways of making comments - single line comments and multiple lines comments. Single line comments are led by the characters //. All the code following this token is considered as comments. The characters /* introduce a multiple lines comment, which terminates with the characters */. Multiple lines comments do not nest and they do not occur within a string or character literals.

2.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length, and for internal identifiers, at least the first 31 characters are significant; some implementations may take more characters significant. Internal identifiers include preprocessor macro names and all other names that do not have external linkage (Par.A.11.2). Identifiers with external linkage are more restricted: implementations may make as few as the first six characters significant, and may ignore case distinctions.

2.4 Keywords

There are 26 identifiers reserved by EasyPet as keywords. These identifiers should not be used for other purposes.

auto	double	int	Panel
Status	Pet	name	arraylist
action	image	sound	false
def	String	char	return
final	continue	for	while
void	do	if	struct
true	def_timer	def_btn	extends
X	Y		

2.5 Constants

Constants are identifiers whose value cannot be changed after being defined for the first time. EasyPet uses keyword final to identify constants. This keyword can be combined with keywords double, int and String to define double constant, integer constant and string constant, respectively.

2.6 String Literals

A string literal is essentially a string constant. It is a sequence of characters enclosed in a pair of double quotes. When a string literal is defined, it is assigned an individual slot of memory, which means for two strings, even if they have the same contents, they are put on memory separately.

After being defined, a string literal cannot be modified further more. Any intended modification on a generated string is stored to a newly allocated memory space.
To contain special characters as well as characters like “ and \, an corresponding escape sequence should be used as shown in the following form.

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

3. Meaning of Identifiers

In EasyPet, identifiers are used to represent either a function or a variable. It is a name that refers to a certain location on either the memory or the dist. Defined with the name, an identifier also has type and storage class attributes. The type attributes describe what kind of data is stored in the allocated space and the storage class attributes rule the lifetime of the identifiers.

3.1 Basic Types

EasyPet provide three basic variable types: *int*, *double* and *String*. Besides, EasyPet has two specific intrinsic data types: I*tem* and *Panel*.

“Item” is a very general data type for item definition. For an item variable, it can contain one single *name* of the item and several related *status*.*name* is used to identify this item and the *status* is used to define the behavior of the item corresponding the environment and time. Notice, an *item* can be used to, but not restricted to describe a pet. For example, it is totally fine to define an *item* of air which every pet can breathe or grass that a rabbit can eat.

Panel is basically a main entry of the program. When an item is defined, it must be initialized in the Panel if the programmer wants the item to be display in the GUI.

3.2 Derived Types

Beside the basic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

Arraylist of objects of a given type;
Functions returning objects of a given type, *def, def_timer, def_btn*;
Structures containing a sequence of objects of various types, *Item, Panel*.

In general these methods of constructing objects can be applied recursively.

3.3 Type Qualifiers

If the programmer wants to keep a variable unchangeable after definition, *final* can be added at the front to achieve this. In addition, EasyPet can implement heritage by *extends* keyword Qualifiers do not influence other aspects of a variable.

4. Lvalue

Lvalue is an expression that refers to a region of storage. It is required by certain operators. Refer to the operator part to see which operators expect an lvalue.

5. Conversions

5.1 Integer and Double

An integer and double number can be casted to each other time. To convert a double number to integer, use *(int)* identifier. Similarly, *(double)* should be used to cast an integer to a number of types double.

When a *double* number is converted to an *int* number, the fractional part is discarded; if the resulting value cannot be represented in the integral type, the behavior is undefined.

When a *int* number is converted to floating, and the value is in the representable range but is not exactly representable, then the result may be either the next higher or next lower representable value. If the result is out of range, the behavior is undefined.

5.2 Arithmetic Conversions

Many operators can take operands of types of both *int* and *double*. In this case, the *int* operand is converted to *double* type implicitly.

6. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

6.1 Primary Expressions

Primary expressions are identifiers, constants, strings or expressions in parentheses.

primary-expression
identifier
constant
string
(expression)

An identifier is a primary expression, provided it has been suitably declared as discussed as below. Its type is specified by its declaration.

An identifier is an lvalue if its type is string, arithmetic, structure.
A constant is a primary expression. Its type depends on its form as described in Part 2.5
A string literal is a primary expression. Its type is originally “array of char”.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The precedence of parentheses does not affect whether the expression is an lvalue.

6.2 Postfix Expressions

The operations in postfix expressions group left to right.

postfix-expression:
primary-expression
postfix-expression.identifier
postfix-expression(argument-expression-list)
postfix-expression<K>
argument-expression-list:
assignment-expression
assignment-expression-list, assignment-expression

6.2.1 Array Reference

A postfix expression followed by an expression in angle brackets is a postfix expression denoting the generic type inside array, e.g. *arraylist<expression>*

6.2.2 Function Calls

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function.

The term argument is used for an expression passed by a function call; the term parameter is used for an input identifier received by function definition or described in a function declaration. In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value.

6.2.3 Structure Reference

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand expression must be a structure, and the identifier must name a member of the structure. The value is the named member of the structure, and its type is the type of the member.

6.3 Unary Operators

Unary expressions are formed by combining a unary operator with a single operand. The unary operators in EasyPet are +, -, and !.

6.3.1 Unary Plus Operator

The operand of the unary + operator must have arithmetic type, and the result is the value of the operand as follows.

+*expression*

6.3.2 Unary Minus Operator

The operand of the unary - operator must have arithmetic type, and the result is the negative of the operand as follows.

-expression

6.3.3 Unary Logical Negation Operator

The operand of the ! operator must have logical type True or False.

!expression

6.4 Binary Operator

The binary operators are categorized as follows:

Multiplicative operators: multiplication (*), division (/) and mod(%);
Additive operators: addition (+) and subtraction(-);
Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=);
Equality operators: equality (==) and inequality (!=);
Logical operators: AND (&&) and OR (||);
Assignment operator (=).

6.4.1 Multiplicative Operators

The multiplicative operators in EasyPet are *, /, and %. Operands must have arithmetic type.
The * operator performs multiplication. The / operator performs division. If two integers don't divide evenly, the result is truncated, not rounded. The % operator performs mod.

6.4.2 Additive Operators

The additive operators in Next are + and -. They perform addition and subtraction respectively.

6.4.3 Relational operators

The relational operators compare two operands and produce an integer literal result. The result is false if the relation is false, and true if it is true. The operators are: less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=). Both operands must have an arithmetic type. The relational operators associate from left to right.

6.4.4 Equality Operators

The equality operators in EasyPet, equal (==) and not-equal (!=), like relational operators, produce a result of an integer literal. The result is true if both operands have the same value, and false if they do not. Both operands must have an arithmetic type.

6.4.5 Logical Operators

The logical operators are &&(and) and ||(or). These operators have left-to-right evaluation. The resulting integer literal is either false or true. Both operators must have logical types. If the compiler can make an evaluation by examining only the left operand, the right operand is not evaluated.
In the following expression,

```
expression1 && expression2
```

the result is true if both operands are true, or false if one operand is false. In the same way, the following expression is true if either operand is true, and false otherwise. If expression1 is true, expression2 is not evaluated.

expression1 || expression2

6.4.6 Assignment Operators

There is only one assignment operator (=) in EasyPet. An assignment results in the value of the target variable after the assignment. It can be used as subexpressions in larger expressions. Outside of the declaration section, assignment operators can only operate on attributes in EasyPet; these are integer and string literals. Assignment expressions have two operands: a modiable value on the left and an expression on the right. In the following assignment:

expression1 = expression2;

The value of expression2 is assigned to expression1. The type is the type of expression1, and the result is the value of expression1 after completion of the operation.

7. Declarations

Declarations introduce identiers (variable names) in the program, and, in the case of the complex types (character, location, item), specify important information about them such as attributes. When an object is declared, space is immediately allocated for it and it is immediately assigned a value.

7.1 Primitive Types

The primitive types in EasyPet are integer, double and string. The primitive types are declared as follows:

int id = value;
float id = value;
string id = value;
Item id(value<, value, ...>);
The id is the name of variable and value stands for a primitive value or an expression.

7.2 Arraylist Type

The *arraylist* type enumerates current type(image/sound) is an array of statuses. The status enumeration type is declared as follows:

arraylist<K> id;

7.3 Complex Types

Each of the complex types has its own declaration syntax. The declaration are as follows:

```
Panel id {
   Item-complex-declaration-list
}

Item id {
   declaration-list:
      primitive-declaration
}
```

Where:
The id stands for the name of variable.

The primitive-declaration-list stands for a list of primitive declaration in the form of primitive-declaration-1, primitive-declaration-2, primitive-declaration-3, etc. These represent the items with some attributes contained in the panel or represent the statuses with attributes, some of which may be possessed by some items.

8. Statements

In Easypet, except as described, statements are executed in sequence.
Statements are executed for their effect, and do not have values. They fall into several groups.

Statement:
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

8.1 Expression Statement

Most statements in EasyPet are expression statements, which have the form

expression-statement:
expression-opt;

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

8.2 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called ``block'') is provided. The body of a function definition is a compound statement.

compound-statement:
{ *declaration-list-opt* statement-list-opt }
declaration-list:
declaration
declaration-list declaration
statement-list:
statement
statement-list statement

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block, after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space; identifiers in different name spaces are treated as distinct.

Initialization of automatic objects is performed each time the block is entered at the top and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initialization of static objects is performed only once, before the program begins execution.

8.3 Selection Statements

Selection statements choose one of several flows of control.

selection-statement:
if (*expression*) *statement*

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

In both forms of the "*if*" statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0.
The else ambiguity is resolved by connecting an else with the last encountered else-less if at the same block nesting level.

8.4 Iteration Statements

Iteration statements specify looping.

iteration-statement:
while (*expression*) *statement*
for (*expression-opt*; *expression-opt*; *expression-opt*) *statement*

In the while statements, the substatement is executed repeatedly as long as the value of the expression remains unequal to 0; the expression must have arithmetic or pointer type. With while, the test, including all side effects from the expression, occurs before each execution of the statement;

In the "for" statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the "for" is terminated. The third expression is evaluated after each iteration, and thus specifies a reinitialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. If the substatement does not contain continue, the statement

for (*expression1*; *expression2*; *expression3*) *statement*
is equivalent to
expression1;
while (*expression2*) {
statement
expression3;
}

Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to testing a non-zero element.

8.5 Jump Statements

A *continue* statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

```
while (...) {
    for (...) {
        ... ... ...
        continue: ;
    }
     continue;
}
```

A *break* statement may appear only in an iteration statement or, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the *return* statement. When *return* is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

9. Scope

A program need not all be compiled at one time: the source text may be kept in several files containing translation units. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

In our language the only one scope to consider is the *lexical scope* of an identifier which is the region of the program text within which the identifier's characteristics are understood;

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces.

The scope of a parameter of a function definition begins at the start of the block defining the function and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

10. Grammar

In Easypet, it defined the typewriter style words and terminals in a literal way, and leave terminal symbols int-constant, char-constant, float-constant, string-constant undefined. Grammar can be transformed into input acceptable for an automatic parser-generator mechanically. To indicate alternatives in productions, one can add whatever syntactic marking. It is necessary to expand "one of" the construction to duplicate each production with am opt symbol. For expression with the opt symbol, it can be optional.

Function-definition:
declaration-list-opt
compound-statement

Primary-expression
identifier
constant
string
(expression)

Postfix-expression:
primary-expression
postfix-expression.identifier
postfix-expression(argument-expression-list)
postfix-expression<K>
argument-expression-list:
assignment-expression

assignment-expression-list, assignment-expression

Unary expression
+expression
-expression
!expression

Statement:
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

Jump-statement:
goto identifier;
continue;
break;
return expression-opt;

Expression-statement:
expression-opt;

Compound-statement:
{ *declaration-list-opt* statement-list-opt }
declaration-list:
declaration
declaration-list declaration
statement-list:
statement
statement-list statement

Selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* else *statement*

Iteration-statement:
while (*expression*) *statement*
for (*expression-opt*; *expression-opt*; *expression-opt*) *statement*

Type specifier: one of

char, int, double, string, item

Expression:
assignment-expression
expression , assignment-expression

Assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

Assignment-operator: one of
= *= /= %= += -= <<= >>= &= ^= |=

11. Library Reference

EasyPet language provides convenient internal functions as follows:

arraylist.add(int i, K)

provides user facility to add object K which is consistant to the generic type into arraylist;

arraylist.remove(int i)

provides user facility to remove the ith object from arraylist;

arraylist.get(int i)

provides user facility to retrieve the ith object from arraylist;

arraylist.attach()

is used to indicate this arraylist is used to represent the arraylist of status (image/sound) will be used in the Panel;

random();

is used to return a random float value between 0 and 1.

12. Graphic User Interface

Electronic pet program usually has a graphic user interface. The graphic user interface shows the images corresponding to each item (pets, plants etc.). The image content changes according to the items' status and the image position changes according to the position of the item. Also, the pop-up window shows the status of the item as well as the button that the user can use to interact with the items. One important convenience that EasyPet provides to the user is the automatic generation of graphic user interface.

12.1 Item Image

The item image reflects the status of the item. Each item can has multiple statuses at the same time and the user can assign each status a corresponding image. To present an image, the user can build an image arraylist and add images into and remove images from it. Then, the user can use *attach* method to attach the arraylist to the item icon on the GUI. For all the images in the arraylist, EasyPet generates codes to present them in the icon one by one in turn.

For each item, EasyPet also reserved two keywords: *X* and *Y*. By modifying the values of these two identifiers, the coder can change the position of the item on the screen to make the item move.

12.2 Item Status

For every item instance, EasyPet automatically collect all its attribute information and present them in a pop-up window.

12.3 Interactive Button

The player interacts with the item by clicking corresponding functional buttons. When programming, the coder defines the expected interaction function *f* with the item. Then EasyPet will generate a button with the name *f*_button for this function and arrange it in the item-specific pop-up window. The coder can refer to the button by using its name.