

AGIL - The U7 Use Code Language

Zachary Simpson

March 28, 1991

TABLE OF CONTENTS

TABLE OF CONTENTS	1
Purpose	2
Basics	2
Comments.....	2
Compiling And Running	2
Identifiers.....	2
Variables	3
Scope	3
Lists.....	4
Dereferencing Lists.....	5
Operators.....	5
Operators and Lists	6
Comparisons	6
Basic Constructs	7
Routines, Functions, and Usables	7
Declaration	7
Returning a Value	7
Examples of returning one value	7
Examples of returning a list	7
Statements.....	8
Calling Routines and Functions.....	8
If Constructs	9
Print Constructs.....	9
Dereferencing Variables	9
Conversation Formatting.....	10
Accessing Special Characters	10
Loop constructs.....	10
While	10
Foreach	11
Converse, Key	11
Break.....	12
Goto	12
Examples of legal goto statements.....	13
Examples of an illegal goto	13
APPENDIX	14
Operator Tables	14
Comparison Tables	15

CONCEPTS

Purpose

The Ultima 7 use-code and conversation language (referred to as AGIL - A Goofy Interpreted Language) is intended to describe the function of all usable items in the world. NPCs are considered just another case of a usable item - where conversation is the result. The language tries to remain as generic as possible. Much functionality will be implemented in the form of C or AGIL routines.

AGIL is based on concepts from several high-level languages including C, Pascal, and LISP. The language is highly structured, but loosely typed. For those unfamiliar with the structured programming techniques of languages such as C and Pascal, I have written a separate document entitled *An Introduction to Structured Programming in AGIL*.

Basics

AGIL programs can be written with any standard text editor. As a convention, the extension .USE is applied to all AGIL source files.

The language is **not** case-sensitive. That is, the variable names Loop, loop, and LOOP are all the same.

Comments

Comments are specified with either `/* */` or with a `//`, and extending to the end of line. Comments may not be nested.

```
Routine XYZ {  
    /* This is a comment*/ Var = 1;  
    Var = 2;    // This is another comment  
}
```

Compiling And Running

To compile an AGIL program, run *UCC* with the filename. This produces several files with the extensions .d and .o.

In order to run, the program *UCCLINK* must be run to convert the .o files into a single file ready for interpretation.

An external interpreter exist currently called *INTER*. Running this program will allow for conversations and some other aspects of AGIL programs to be tested before including them into the game.

Identifiers

Identifiers in AGIL (variables, routine names, and labels) have the following significant character lengths:

Variables	Unlimited
Labels	Unlimited
RoutineNames	8

This means that the following two routine names are the same: BigName11 and BigName12. Because one is not allowed to specify more than one routine name with the same name, an error would be generated if these two names appeared during compile.

Variables

Variables in AGIL are very flexible; they are not required to be of any particular type, but are instead determined at run-time to be either integers or strings, based on context.

For example, the statement `Var = 1` assigns the integer value 1 to the variable `Var` and hence, `Var` becomes of type integer. A similar statement of `Var = "abcdef"` will assign the string to `Var`.

All operations are made intelligently based on type. In most cases, the programmer will be specifying a logical operation of integer/integer or string/string. However, in the case that a mis-matched operation is made, the interpreter does its best to resolve the expression.

The basic rule of mis-matched operations is that the left-hand side of an expression determines the end type.

For example, the expression `12 + "13"` will always resolve to an integer because the left-hand side is an integer type. In this case, the interpreter will try to convert the string "13" to the integer 13, and then add 12+13 to get 25. If however, the string can not be converted to a number then an expression such as `12 + "abc"` would resolve to a best-fit depending on the operation type. In this case the result would be 12. (The best-fit definitions are defined in the *Operator Table* in the appendix.)

The opposite example would be a string base and an integer operand. For example `"abc" + 123` would convert the number to a string and concatenate the result to create the string "abc123".

Scope

All variables that the programmer can specify in AGIL are local in scope. Global variables and routines are defined, but will be handled specially to force proper integration.

Variables do not need to be declared formally, however, the compiler does enforce that variables be initialized before a reference in the source file.

Illegal reference of Variable

```
Routine DoItToIt {  
    while Variable < 10 {  
        [Hello];  
    }  
}
```

Legal initialization of Variable

```
Routine DoItToIt {  
    Variable = 0  
    while Variable < 10 {  
        [Hello];  
    }  
}
```

Note, the enforcement of pre-initialization can not always prevent the programmer from referencing an uninitialized variable. For example, one could initialize a variable inside of an if statement, and then reference that variable outside of the if. Because the compiler only cares about position of the variable reference relative to the top of the source file, this would be a legal initialization even though it is impossible to know if the variable will be initialized.

Example of a legal program that will result in a reference to an uninitialized variable

```
Routine NebulusInit {  
    if False {      // can never be executed!  
        Var = 1;  
    }  
    if Var = 1 {    // Var was never initialized!  
        [Hello];  
    }  
}
```

Lists

A list is a collection of data elements in a specific order. In AGIL, any variable can be a list. To construct a list, the delimiters << >> are used.

The following are examples of legal lists:

```
Members = << "Avatar", "Dupre", "Iolo", "Shamino" >>;  
Digits  = << 1,2,3,4,5,6,7,8,9,0 >>;  
Junk    = << "ABC", 45, Digits, "A String!">>;
```

As demonstrated, the elements of a list do not all have to be of the same type, one may freely mix integer and string types.

In addition, a list may reference another list, such as demonstrated in the previous example with *Digits* in the list *Junk*. However, caution should be used when referencing one list within another because each element of the referenced list will be added to the new list.

For example, given the following two lists:

```
a = << 1, 2, 3 >>;
b = <<a, 4, 5, 6>>;
```

Then the second element of *b* is 2, and the 4th element is 4! Every single element of *a* was added to *b*, thereby changing the relative position of all following elements in *b*!

Dereferencing Lists

A particular element may be retrieved from a list with the construct **List::ElementNumber**. For example, given the list:

```
MyList = << "A", "B", "C" >>;
```

then the expression

```
Var1 = MyList::2
```

would set the variable *Var1* to equal "B".

Operators

The following table lists the operators in the AGIL language.

=	Assignment
+	Add
-	Subtract
*	Multiply
/	Divide
modulo	Modulo Arithmetic
[]	Print (conversations)
<< >>	Construct List
" "	String constant
::	List Dereference

Operations are defined differently depending on the types of data involved, for example:

```
2 + 1 = 3
"abc" + "def" = "abcdef"
2 - 1 = 0
"abc" - "def" = "abc"    // subtraction is undefined for strings
1 + "abc" = 1
1 + "2" = 3              // strings are converted to numbers
                          // if possible
"abc" + 1 = "abc1"       // numbers are converted to strings
```

For a complete explanation of the function of all operators on every data type, refer to the *Operator Table* in the appendix.

Operators and Lists

When operators are used on a list, they always operate on the first element of the list.

For example,

```
List1 = << 1, 2, 3 >>;  
List2 = << 4, 5, 6 >>;  
List1 + List2 = 5      // 1 + 4 = 5
```

There are no operators that work solely on lists. However, there will be some routines implemented which will perform tasks such as concatenate two lists and add elements to lists.

Comparisons

The following table lists the comparison operators in the AGIL language.

=, is	Equals
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal
!=, <>	Not Equal
and	Logical And
or	Logical Or
not	Logical Negation
true, yes	Logical True
false, no	Logical False
in	Is member of list

When making comparisons between different data-types, the interpreter will attempt to convert the data to the type of the left-hand side of the comparison. For example:

```
1 = 1           // True  
"abc" = "abc"  // True  
1 = "abc"      // False  
1 = "1"        // True  
"1" = 1        // True  
"abc" = 1      // False
```

For a complete explanation of all comparisons on every data-type, see the *Comparison Table* in the appendix.

Basic Constructs

Routines, Functions, and Usables

There are three types of sub-routines in AGIL: Routines, Functions, and Usables. The distinction is made as follows:

Functions may accept parameters and always return a value.

Routines may accept parameters, but never return a value.

Usables are declared for each usable object type in the world. They never accept parameters nor return values.

Declaration

Routines and functions are declared in the form:

```
Routine RName accepts Param1, ... { thingss_to_do; }
```

```
Function FName accepts Param1, ... { things_to_do; return value; }
```

```
Usable UsableItemName { things_to_do; }
```

Returning a Value

Functions must return a single value with the statement *return* as described above. The return can return only a single value, however, the value may be a list. For example:

Examples of returning one value

```
Function GetPlayerName {  
    return "Avatar";  
}
```

```
Function Square accepts X {  
    S = X*X;  
    return S;  
}
```

Examples of returning a list

```
Function GetParty {  
    return << "Avatar", "Dupre", "Iolo" >>;  
}
```

```
Function GetDays {  
    WeekEnd = << "Sat", "Sun" >>;  
    return WeekEnd;  
}
```


Statements

The instructions that define a *function*, *routine*, or *usable* are called **statements** and are encapsulated in the curly braces {}.

Every statement must end with a semi-colon (;).

However, the start and end of the routines, as well as other compound expressions such as *if*, *while*, *foreach*, etc. do not need semi-colons as they are not statements. However, if you do use them it will be OK.

Calling Routines and Functions

Routines and functions may call each other using the following convention:

```
RoutineName (parameter_list);  
  
RoutineName ();
```

The following illustrates a routine call to the routine *ARoutine* with two parameters.

```
Routine ARoutine accepts A, B {  
    ...  
}  
  
Routine TryToCall {  
    ARoutine (1,2);    // calls ARoutine with the parameters 1 and 2  
}
```

Because *functions* return values, the call to a function has an implicit value. The following example demonstrates a call to the square function, and how the return value can be used.

```
Function Square accepts X {  
    return X*X;  
}  
  
Routine Test {  
    A = Square (2);    // A will be assigned 4  
    // the next line constructs a list of powers  
    B = << Square (1), Square (2), Square (3) >>;  
}
```

If Constructs

If constructs are formed in either of the following ways:

```
if boolean_expression { if_true_code; }

if boolean_expression { if_true_code; } else { else_code; }
```

For example:

```
Function IsOdd accepts x {
  if x modulo 2 = 0 {
    return Even;
  }
  else {
    return Odd;
  }
}
```

In an *if* statement, the *boolean_expression* is any expression which evaluates to **true** or **false**. See the *Comparison* section under *Variables* for a list of all boolean operators.

Print Constructs

Conversation text may be printed by encapsulating the text in square brackets []. All text inside of these brackets will be printed, with the exception of certain special characters that are used for formatting. The following table lists these characters:

~	Carrige Return
*	Force a key to be pressed
@	Highlight word
<>	Dereference varaible

Dereferencing Variables

Inside of a print statement, a variable's value may be reference by surrounding the varaible name with <>. For example, the following program:

```
Name = "Avatar";
[Hello <Name>!!!];
```

Would produce the output:

```
Hello Avatar!!!
```

There may not be any expressions within the <>. For example the following is **illegal**:

```
A = 1;  
[a times 3 = <a*3>]; // This is illegal!!!
```

Conversation Formatting

When displaying a conversation text, the interpreter will try to intelligently display the text. The following priorities are used:

Stars (*) will always cause a pause.

If the screen is filled without a star, then it will break at the last punctuation (period, question mark, or exclamation mark.) If the punctuation is followed by a quote, then the quote will serve as the punctuator, to avoid separating the end-quote from its sentence.

If no punctuation is found, it will break at a CR (~).

If no CR exists, it will resort to a word boundary.

Accessing Special Characters

If one needs to actually display any of the special characters, the backslash character (\) can be used. For example:

```
[This will print the \@ character];  
[This will print the @\@ character highlighted!];
```

Other characters not listed in the table above can be accessed normally. For example:

```
["Hello Siddhartha", Joey says.]
```

will display exactly what is between the brackets, including the quote marks.

Loop constructs

While

There are several special purpose looping constructs in AGIL. The simplest loop is the **while** loop, which takes the form:

```
While boolean_expression { while_true_code; }
```

Which will execute the *while_true_code* while the *boolean_expression* remains true.

Foreach

The foreach loop will traverse each element in a list, assigning the current element to a temporary variable. Foreach loops are expressed:

```
Foreach TempVar in List { foreach_code; }
```

For example:

```
Routine KillParty {  
    Party = GetPartyList(); // GetPartyList() returns a list  
    Foreach Member in Party { // Traverse Party and assign to Member  
        Kill (Member);  
    }  
}
```

Converse, Key

The converse loop is designed to make the inherent loop of a conversation easy to specify. It takes the form:

```
converse { while_input_code; }
```

The converse loop automatically prompt the end-user for an entry with the prompt "You say:". The resultant input is stored in a globally accessible variable, and can be compared to expressions using the *key* command.

**Only one key statement can be true per loop of a converse.
All other keys that match are ignored.**

The following example demonstrates a typical conversation:

```
Usable Joey { // Joey is some pre-defined NPC  
               // (using an NPC means talking.)  
    converse {  
        key ("name", "job") { // Compare the input  
                               // to these two strings  
            [I'm joey, the idiot.]; // if they compare,  
                               // then say this.  
        }  
        key ("piss off") {  
            [Hey, screw you!];  
            break; // Terminates the converse  
        }  
    }  
    key * { // The default key matches anything  
        [I don't know what you're talking about!];  
    }  
    YewInformation(); // Calls the YewInformation Routine  
    WorldInformation(); // Calls the WorldInformation Routine  
}
```

```

Routine YewInformation {
    key ("lord", "brit") {
        [Yeah, that Lord British is a real jerk.];
    }
}

Routine WorldInformation {
    key ("lord", "brit") {
        [Yeah, that Lord British is one swell guy.];
    }
    key ("justice") {
        [The town of justice is Yew.];
    }
}

```

Because the *key* operator acts on a global string of last input, one can call a routine in the middle of a converse statement to do a set of comparisons, as *YewInformation* and *WorldInformation* were in the previous example. Notice that this type of construct can be used to overload keywords depending on context because only one key may be matched per converse statement. Therefore, since *YewInformation* is called first, the key "lord" will match to the statement *"Yeah, that Lord British is a real jerk."* instead of the default *WorldInformation* response of *"Yeah, that Lord British is one swell guy."*

The key * is the default response key. Any response other than nothing will match this key. For this reason, the default should always be listed last in the list of keys.

The converse loop is usually terminated by a blank entry from the end-user, but can be terminated pre-maturely with a break statement (described below) as illustrated in the previous example's keyword *"piss off"*.

Converse loops may not be embedded.

Break

The command **break** can be used to jump out of the current loop, as demonstrated with the converse example above. It will work not only on *converse*, but also on *while* and *foreach* statements. However, the break command should be used conservatively as it can lend to confusing programs.

Goto

A goto statement is allowed within a function to cause the program to jump to a label indicated by an @ followed by a label name. The following demonstrates a goto:

Examples of legal goto statements

Examples of an illegal goto

```
Routine DoIt {  
  List = << 1, 2, 3 >>;  
  While True {  
    Foreach X in List {  
      [Hello];  
      if X = 1 {  
        goto DoneJump;  
      }  
    }  
  }  
  @DoneJump; // Label  
}
```

```
Routine GoofyLoop {  
  @Loop; // Label  
  [Hello];  
  goto Loop;  
}
```

```
Routine Print {  
  @JumpInPrint;  
  [This is an illegal goto];  
}
```

```
Routine BadGoto {  
  goto JumpInPrint;  
  // this goto will result  
  // in an error message  
  // "Undefined label  
  // JumpInPrint." because the  
  // label is outside of the  
  // routine.  
}
```

There are only a few circumstances when a goto statement is a good idea. One of these times is demonstrated in the first example above, where the goto is used to jump out of several nested loops. The second example in the first column is a good example of the kind of goto to avoid, as it is difficult to understand, and can be easily replaced by a looping construct such as *while*.

APPENDIX

Operator Tables

+ (Add)	Integer	String	Uninitialized
Integer	Integer Add	Convert String to Integer and Add. If convert fails then add 0	Return Integer
String	Convert Integer to string and concatenate	Concatenate	Return String
Uninitialized	Return Integer	Return String	Return Uninitialized

- (Subtract)	Integer	String	Uninitialized
Integer	Integer Subtract	Convert String to Integer and Subtract. If convert fails then subtract 0	Return Integer
String	Return String	Return String	Return String
Uninitialized	Return -1*Integer	Return Uninitialized	Return Uninitialized

* (Multiply)	Integer	String	Uninitialized
Integer	Integer Multiply	Convert String to Integer and Multiply. If convert fails then return Integer	Return Integer
String	Return String	Return String	Return String
Uninitialized	Return 0	Return Uninitialized	Return Uninitialized

/ (Divide)	Integer	String	Uninitialized
Integer	Integer Divide (Truncate fractional portion). Traps for divide by 0 and returns 32767	Convert String to Integer and Divide. If convert fails then return Integer. Traps for divide by 0 and returns 32767	Return Integer
String	Return String	Return String	Return String
Uninitialized	Return 0	Return Uninitialized	Return Uninitialized

Modulo	Integer	String	Uninitialized
Integer	Integer Modulo (remainder portion of division)	Convert String to Integer and Modulo. If convert fails then return Integer (??)	Return Integer
String	Return String	Return String	Return String
Uninitialized	Return 0	Return Uninitialized	Return Uninitialized

Comparison Tables

=,is (Equal)	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return False	if Integer = 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	if String is empty return True, else return False
Uninitialized	if Integer = 0 then return True, else return False	If string is empty, return True, else return False	Return True

!=,<> (NE)	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return True	if Integer != 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	if String is not empty return True, else return False
Uninitialized	if Integer != 0 then return True, else return False	If string is not empty, return True, else return False	Return False

>	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return False	if Integer > 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	if String is not empty return True, else return False
Uninitialized	if Integer < 0 then return True, else return False	Return False	Return False

<	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return False	if Integer < 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	Return False
Uninitialized	if Integer > 0 then return True, else return False	If string is not empty return True, else return False	Return False

>=	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return False	if Integer >= 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	Return True
Uninitialized	if Integer <= 0 then return True, else return False	If string is empty return True, else return False	Return True

<=	Integer	String	Uninitialized
Integer	Integer compare	Convert String to Integer and compare. If convert fails then return False	if Integer <= 0 then return True, else return False
String	Convert Integer to String and compare.	String compare	if string is empty return True, else False
Uninitialized	if Integer >= 0 then return True, else return False	Return True	Return True