WASP Language Reference Manual

Dustin Burge, John Chung, Tingting Li, Neel Vadoothker

Tabe of contents

1 Introduction

This manual describes the WASP language.

2 Lexical Conventions

2.1 Tokens

There are six classes of tokens: identifiers, keywords, string literals, constants, operators, 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 and keywords.

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 Comment

The characters `\*` introduce a comment, which terminates with the characters `\*`. Comments are not nestable. They do not occur within string or character literals.

2.3 Identifiers

Identifiers are made up of letters and digits; the first character must be a letter.

The underscore `_` counts as a letter. Upper and lower case letters are distinct, so `a` and `A` are two different names.

2.4 Keywords

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

- `bool`
- `int`
- `float`

- `string`
- `if`
- `else`
- `list`
- `opt`
- `Func`
- `Object`
- `Endpoint`
- `while`
- `break`

2.5 Constants

String literals, or a string constant, is a sequence of characters surrounded by double quotes.

Integer constants consist of a sequence of digits that are strictly decimal. No octals or hexadecimal digits are allowed.

Floating constants consist of an integer part and a decimal part. Either the integer part, or the fraction part (not both) may be missing.

3 Meaning of Identifiers

Identifiers, or names, refer to either functions or variables. A variable is a location in storage, and its interpretation depends on two main attributes: and its type and any type qualifiers. The type determines the meaning of the values found in the identified object. Type qualifiers are used to add additional specifications to a variable. The only type qualifier in version 1.0 of WASP is the `opt` qualifier, described in detail in section 3.2. A name also has a scope, which is the region of the program in which it is known.

3.1 Basic Types

Strings

Strings are declared with the `string` type and have the default value of null.

String literals are declared with double quotes.

Special characters must be escaped by using the backslash `\`.

```
string a ;                        /* null */
string a = "Hello" ;              /* Hello */
string a = "\'Hello world\'" ;    /* 'Hello world' */
```

Integers

Integers are declared with the `int` type and have the default value of null.

Integers are `signed` and 8 bytes that can store values between –(2^31) and 2^31–1.

```
int n ;             /* null */
int n = 7 ;         /* 7 */
```

Floating Point Numbers

Floating point numbers are declared by default with the type `float` and have the default value of null.

```
float f ;           /* null */
float f = 0.11 ;    /* 0.11 */
```

Booleans

Boolean values are the two constant objects False and True; they have the default value False.

They are used to represent truth values.

```
True ;          /* True */
False ;         /* False */
1 == 1 ;        /* True */
1 == 2 ;        /* False */
```

3.2 Derived types

Object

WASP allows for a non-primitive data type (Object) that is a collection containing any number of primitive data type variables and/or non-primitive type variables.

Objects are meant to represent the JavaScript Object Notation (JSON) objects that will be used to communicate with a WASP created server.

The `opt` keyword is used to specify that an Object's member is optional for PUT and POST requests.

```
Object Image{
  string username ;
  string imageUrl ;
  string description ;
  opt string geo ;
}
```

The keyword `Endpoint` is used instead of `Object` to define a RESTful endpoint for an Object in addition to its member structure.

List

A list is a non-primitive data type that defines an ordered grouping of objects of the same type (primitive or non-primitive).

List literals are declared with comma-separated values between square brackets.

Empty lists are declared by `list[]`

```
list[] ;                             /* empty list */
list1 = [1, 2, 3, 4, 5 ] ;           /* list of ints */
list2 = ["a", "b", "c", "d"] ;       /* list of strings */
list3 = [True, False, True ] ;       /* list of bools */
```

3.3 Type Qualifiers

An object's type may have an additional qualifier.

Declaring an object `opt` announces that it is not a required value when `PUT`ting or `POST`ing to the web server.

4 Objects and Lvalues

An object is a named region of storage. Every object will have an `ID` attribute generated (this is a reserved attribute that cannot be used). This is a unique identifier to the object. An lvalue refers to an object that persists beyond a single expression.

5 Conversions

Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. Section 5.4 summarizes the conversions demanded by most ordinary operators.

5.1 Integral Conversions

For operators that are applied to expressions of different type, is that the the operands are converted to the type where there is less loss in precision. If both operands have the same type, then no further conversion is needed.

```
int a = 10 ;
float b  = 10.5 ;
int c = a + b \\c is 20 ;
```

5.2 Integer and Floating

When a value of floating type is converted to integral type, the fractional part is discarded. If the resulting value cannot be represented in the integral type, the behavior is undefined (conversion overflows). When a value of integral type 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. Again, if the result is out of range, the behavior is undefined (conversion overflows).

5.3 Floating Types

WASP has only one floating type which will be equivalent to the IEEE double-precision floating point and equivalent to the `C lang's` double.

5.4 Arithmetic Conversions

Operators perform type conversions to bring the operands of an expression to a common type. The purpose is to yield a common type which is also the type of the result.The compiler uses the following two steps to binary operations in the expression.

- If either operand is of type float, the other operand is converted to type float
- If either operand is of type int, the other operand is converted to type int

6 Expressions

6.1 Operator Precedence

Operator Type	Operator	Associativity
Primary Expressions	() []	Left
Unary	! -	! is Nonassoc, - is Right
Binary	. / ./ + .+ - .- %	Left(except ^ which is Right)
Assignment	=	Left

6.2 Primary Expressions

6.2.1 identifier

Identifiers are primary expression. All identifiers have an associated type that is given to them upon declaration (e.g. `float angle_a` declares an identifier named angle_a that is of type `float`).

6.2.2 literals

Literals are primary expression. They are strings, integers, floats, booleans, objects, and lists. Strings are of type `string`, integers are of type `int`, floats are of type `float`, booleans are of type `bool`, objects are of type `Object`, and list are of type `list`.

6.2.3 (expression)

Parenthesized expressions are primary expressions. The type and value of a parenthesized expression is the same as teh type and value of the expression without parenthesis. Parentheses allow expressions to be evaluated in a desired precedence. Parenthesized expressions are evaluated relative to each other starting with the expression that is nested the most deeply and ending with expression that is nested the shallowest.

6.2.3 primary-expression(expression-list)

Primary expressions followed by a parenthsized expression list are primary expressions. This kind of primary expressions are used in the declaration of functions or function calls. The expression list is consist of one or more expressions separated by commas. When it is used in function declarations, they must be preceded by the correct function declaration syntax and each expression in the expression list must evaluate to a type followed by an identifier. When it is used in function calls, each expression in the expression list must evaluate to an identifer.

6.2.4 primary-expression[expression-list]

Primary expressions followed by a square bracketed expression list are primary expressions. This kind of primary expressions are used in the declaration of lists or to access an element of a list. The expression list is consist of expressions separated by commas, and must evaluate to the same type of `string`, `int`, `float`, or `Object`. An example declaration of list of strings is shown below.

```
string a = "hello" ;
string b = "world" ;
list my_list = [a, b, "!" ] ;
```

6.3 Unary Operators

6.3.1 ! expression

The result is a Boolean value of either `True` or `False` indicating the logical not of the expression. The type of the expression must be `bool` or `int`. In the expressions, 0 is considered False and all other values are considered true.

```
!True ;      /* False */
!bool(32) ; /* False */
!1==2 ;      /* True */
```

6.3.2 - expression

The result indicates the arithmatic negative value of the expression. The type allowed for the express is of type either `int` or `double`. The result is of type either `int` or `double`.

6.4 Binary Operators

Binary Operators includes arithmatic operators such as: `+, -, *, /, %, ^, .+, ./, .-, .*` . The dot indicates operators for floats.

6.4.1 `expression + expression` :

The result is the sum of the expressions. The type allowed for the expressions are of type `int` . The result is of type `int` .

6.4.2 `expression .+ expression` :

The result is the sum of the expressions. The type allowed for the expressions are of type `float` . The result is of type `float` .

6.4.3 `expression - expression` :

The result is the difference of the first and second expression. The type allowed for the expressions are of type `int` . The result is of type `int` .

6.4.4 `expression .- expression` :

The result is the difference of the first and second expression. The type allowed for the expressions are of type `float` . The result is of type `float` .

6.4.5 `expression * expression` :

The result is the product of the expressions. The type allowed for the expressions are of type `int` . The result is of type `int` .

6.4.6 `expression .* expression` :

The result is the product of the expressions. The type allowed for the expressions are of type `float` . The result is of type `float` .

6.4.7 `expression ^ expression` :

The result is the exponentiation of the first expression by the second expression. The type allowed for the expressions are of type `int` . The result is of type `int` .

6.4.8 `expression .^ expression` :

The result is the exponentiation of the first expression by the second expression. The type allowed for the expressions are of type `float`. The result is of type `float`.

6.4.9 `expression / expression`:

The result is the quotient of the expressions, where the first expression is the divident and the second is the divisor. The type allowed for the expressions are of type `int`. The result is of type `int` and is rounded towards 0 and truncated. (If an error is occured because of a division by zero, a runtime exception is raised.)

6.4.10 `expression ./ expression`:

The result is the quotient of the expressions, where the first expression is the divident and the second is the divisor. The type allowed for the expressions are of type `float`. The result is of type `float`. (If an error occurs because of a division by zero, a runtime exception is raised.)

6.4.11 `expression % expression`:

The result is the remainder of the division of the expressions, where the first expression is the divident and the second is the divisor. The sign of the divident and the divisor are ignored, so the result returned is always the remainder of the absolute value of the divident divided by the absolute value of the divisor. The type allowed for the expressions are of type `int`. The result is of type `int`.

6.5 Assignment Operators

Assignment operators have left Associativity

`thisValue = expression`

The result is the the assignment of the expression to the `thisValue`. the `thisValue` must have been previously declared. The type of the expression must be of the same that the `thisValue` was declared as. `thisValue` can be declared as `string`, `int`, `float`, `list`, or `Object`.

6.6 Casts

Three types of casts are allowed:

- (int)literal returns int

- (float)literal returns float
- (string)literal returns string

Each cast is a function that must have a signature matching the cast operation.

Each cast function will throw a runtime error as well if the cast cannot be performed properly.

6.7 Multiplicative Operators

The multiplicative operators (`*, /, %`) are defined in section 6.4.

Each of the three operators associates from left to right.

6.8 Additive Operators

The additive operators (`+, -`) are defined in section 6.4.

Both operators associate left to right.

6.9 Other Binary Operators

The set of boolean relational operators allowed are as follows:

- expr == expr, the expr are equivalent to each other
- expr != expr, the expr are not equivalent to each other
- expr < expr, the left expr is less than the right expr
- expr <= expr, the left expr is less than or equal to the right expr
- expr > expr, the left expr is greater than the right expr
- expr >= expr, the left expr is greater than or equal to the right expr
- expr ~= expr, the left expr is contained in the right expr

The resultant of each of the above binary operators is boolean value. The

The set of binary boolean relational operators allowed are as follows:

- expr && expr, evaluates to true if the left and right expr are both true
- expr || expr, evaluates to true if either the left or right expr are true

  The resultant of each of the above binary operators is a boolean value.

The following lists mandate which primitive data types are allowed as left and right expr, after evaluation, for each operator:

bool:

- ==
- !=
- &&
- ||

int:

- ==
- !=
- <
- <=
- `>=`
- >

float:

- ==
- !=
- <
- <=
- `>=`
- >

string:

- ~=
- !=
- ==

6.10 Equality and Relational Operators

The equality operators (`!=, =`) associate from left to right and lower precedence than the relational operators (`<, <=, >, >=`), which also associate from left to right.

6.11 Logical AND Operator

The logical and operator is `&&`. It associates from left to right. The operator takes the form expr && expr, which must evaluate to a boolean scalar value. Each expr must evaluate to a boolean scalar value.

The result of the logical and operator is true if both the left and the eight expressions evaluate to true.

6.12 Logical OR Operator

The logical or operator is `||`. It associates from left to right. The operator takes the form expr || expr, which must evaluate to a boolean scalar value. Each expr must also evaluate to a boolean scalar value as well.

The result of the logical or operator is true if either the left or right expressions evaluates to true.

6.13 Assignment Expressions

The only assignment operator allowed is `=`. The assignment operator allows the left expression to be set to the value of the right expression.

lvalue = expr indicates that the lvalue will be set to the expr. This operator associates from right to left.

6.14 Comma Operator

The comma operator ',' is only used in two locations.

One location is when defining elements of a list.

The second location is when defining elements of the object derived data type.

The operator evalues each expr, discards it, and then evaluates the second expr and continues until no more expr (operands) are available).

6.15 Constant Expressions

Any single valued literal is a constant expression. Any expression involving only constant expressions is itself also a constant expression. Any expression involving function calls or variables is usually not a constant expression.

```
3 ;
"Hello" ;
0.01 ;
```

7 Declarations

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called definitions. Declarations have the form:

```
declaration:
  declaration-specifiers identifier

declaration-specifiers:
  type-specifier
  declarator
```

The declaration-specifiers consist of a sequence of type-specifiers and declarators.

WASP is statically typed which means variables must be explicitly declared before use. Casting between float, int can be done as seen in section 6.6.

7.1 Type Specifiers

Type specifiers in WASP are:

- string
- int
- float
- object
- endpoint

7.2 Declarators

A declarator is the part of a declaration that specifies the name that is to be introduced into the program. WASP allows declarators for data types discussed in section 3. Also, we can declare functions using the declarator `func`.

```
Func float CToF (float c) {
  return c * 9.0 ./ 5.0 .+ 32.0 ;
}
```

WASP has the special Endpoint declarator which will create an endpoint and start a server. The syntax is as follows:

```
Endpoint HelloWorld (GET:) {
  string message = "Hello World." ;
}
```

The parenthesis following an Endpoint's ID contain optional comma separated list of HTTP Methods (DELETE, GET, POST, and/or PUT) followed by a colon and an optional comma separated list of parameters (defined with a type and an ID).

In the above example, an Endpoint with the ID `HelloWorld is created`. It responds only to HTTP GET requests and requires no paramaters to be passed as part of the request's body. The object associated with the `HelloWorld` endpoint will return only a single member `message` which is defined as a constant string "Hello World."

7.3 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers. Two types are the same if their declarators, after deleting any function parameter identifiers, are the same up to equivalence of type specifier lists. Array sizes and function parameter types are significant.

8 Statements

8.1 Expression Statement

Expression statements are used for their side effects in WASP.

They simply have the form: expression;

The semicolon indicates a statement ending.

8.2 Iteration Statement

The while statement is the only iteration statement allowed.

The while statement takes the form: while (expr) { statement }

The expr must evaluate to a boolean value.

The statement is executed repeatedly until the value in expr no longer evaluates to true. If expr evaluates to False before statement is reached, then the loops is essentially skipped.

An optional `break` statement can be called to end the iteration of a while loop early.

8.3 Selection Statements

The control flow keywords are: `if`, `else`.

A standard control flow statement will be as follows:

if (expr) { statement } else { statement }

The else branch is also optional, but there can only be one of them.

The expr in the if branch must evaluate to a boolean expression. The statement inside of the if branch will only be run if the expr evaluates to true.

Only one of the statements can be run inside an if else branch. The previous and latter statements will not be run.

The statement corresponding within the else branch will only be run if all if expr values evaluate to false.

The expr are evaluated in order of appearance starting with the if branch, and subsequent expr will not be evaluated if a previous expr has evaluated to True.

9 Declarations

9.1 Function Definitions

A function is defined as follows:

```
Func float squareFloat (float input) {
  return input .* input ;
}
```

The type following the `Func` keyword is a function's return type. Functions can return only a single value, and there is no void type. Functions are not used for their side effects. The next string is the function's ID. The parenthesis following an function's ID contain an optional comma separated list of list of parameters (defined with a type and an ID).

In the above example, an function with the ID `squareFloat is created`. It takes a single float argument `input` and returns the square of that value using the float multiply binary operator.

10 Scope Rules

Name bindings have a block scope - the scope of a name binding is only valid to a section of code that is grouped together. The name can only be used to associated entity within the block of code. Blocks of code is described by the opening curly brace `{` at the start of the block, and the closing curly brace `}` at the end of the block.

There are three scope rules in WASP. Depends on where each variable is declared, the scope of that variable is different.

- If a variable is defined at the outermost block of a program, it is visible anywhere within the program. In other words, the variable has global scope.
- If a variable is defined as a parameter to a function or is declared within a function, it is visible only within that function. Nested functions will have access to variables declared from its outer functions.
- Declarations made after a specific declaration are not visible to it, or to any other declarations before it.

```
float a = 1.5 ;
float b = a + 5.5 ;/* This is valid */
float c = d + 5.5 ; /* This is invalid */
float d = 2.5 ;

Func int foo (int x) {
  int a = x ;
    int b = 2 ;
    Func int PlusTwo (int y) {
        return y + b ;
    }
    return PlusTwo(a) ; \\will return x + 2
}
```

11 Error Generation

Errors can be thrown through the keyword throw.

The throw statement is completed as follows: throw Error('This is an error');

A throw statement will cause the program to crash and the error message to be printed to stdout.

12 Grammar

```
program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls vdecl { ($2 :: fst $1), snd $1 }
 | decls fdecl { fst $1, ($2 :: snd $1) }
 | decls odecl { fst $1, ($2 :: snd $1) }

vdecl:
    var_type ID
      { { vname = $2;
          vtype = $1; } }

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

var_type:
    INT    { Int }
  | FLOAT  { Float }
  | STRING { String }
  | BOOL   { Bool }
  | LIST   { List }

fdecl:
   FUNC var_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list R
     { { fname = $2;
         formals = $5;
         locals = List.rev $8;
         body = List.rev $9 } }

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

formal_list:
    vdecl                   { [$1] }
  | formal_list COMMA vdecl { $3 :: $1 }
```

```
odecl:
    OBJ ID LPAREN htypes_opt SEMI formals_opt RPAREN LBRACE vdecl_list RBRA
      { { oname = $2;
          otype = Object;
          htypes = $4;
            formals = $6;
            locals = List.rev $9 } }
  | END ID LPAREN htypes_opt SEMI formals_opt RPAREN LBRACE vdecl_list RBRA
      { { oname = $2;
          otype = Endpoint;
          htypes = $4;
          formals = $6;
          locals = List.rev $9 } }

htypes_opt:
    /* nothing */ { [] }
  | htypes_list   { List.rev $1 }
```

```
htypes_list:
    GET                    { [Get] }
  | PUT                    { [Put] }
  | POST                   { [Post] }
  | DEL                    { [Del] }
  | htypes_list COMMA GET  { Get :: $1 }
  | htypes_list COMMA PUT  { Put :: $1 }
  | htypes_list COMMA POST { Post :: $1 }
  | htypes_list COMMA DEL  { Del :: $1 }

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) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
```

```
expr:
    LITERAL            { Literal($1) }
  | ID                 { Id($1) }
  | NOT expr           { Unop(Not, $2) }
  | NEG expr           { Unop(Neg, $2) }
  | 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 FPLUS   expr  { Binop($1, Fadd, $3) }
  | expr FMINUS  expr  { Binop($1, Fsub, $3) }
  | expr FTIMES  expr  { Binop($1, Fmult, $3) }
  | expr FDIVIDE expr  { Binop($1, Fdiv, $3) }
  | expr AND     expr  { Binop($1, And, $3) }
  | expr OR      expr  { Binop($1, Or, $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 CONT    expr  { Binop($1, Cont, $3) }
  | ID ASSIGN expr     { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

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

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