WLPP Programming Language Specification

The WLPP programming language contains a strict subset of the features of C++. A WLPP source file contains exactly one procedure definition.

Lexical Syntax

A procedure definition is a sequence of *tokens* optionally separated by *white space* consisting of spaces, newlines, or comments. Every valid token is one of the following:

- ID: a string consisting of a letter (in the range a-z or A-Z) followed by zero or more letters and digits (in the range 0-9), but not equal to "wain", "int", "if", "else", "while", "println", "return", "NULL", "new" or "delete".
- NUM: a string consisting of a single digit (in the range 0-9) or two or more digits the first of which is not 0
- LPAREN: the string "("
- RPAREN: the string ")"
- LBRACE: the string "{"
- RBRACE: the string "}"
- RETURN: the string "return" (in lower case)
- IF: the string "if"
- ELSE: the string "else"
- WHILE: the string "while"
- PRINTLN: the string "println"
- WAIN: the string "wain"
- BECOMES: the string "="
- INT: the string "int"
- EQ: the string "=="
- NE: the string "!="
- LT: the string "<"
- GT: the string ">"
- LE: the string "<="
- GE: the string ">="
- PLUS: the string "+"
- MINUS: the string "-"
- STAR: the string "*"
- SLASH: the string "/"
- PCT: the string "%"
- COMMA: the string ","
- SEMI: the string ";"
- NEW: the string "new"
- DELETE: the string "delete"
- LBRACK: the string "["
- RBRACK: the string "]"
- AMP: the string "&"
- NULL: the string "NULL"

White space consists of any sequence of the following:

- SPACE: (ascii 32)
- TAB: (ascii 9)
- NEWLINE: (ascii 10)
- COMMENT: the string "//" followed by all the characters up to and including the next NEWLINE

Any pair of consecutive tokens may be separated by white space. Pairs of consecutive tokens that both come from one of the

following sets *must* be separated by white space:

- {ID, NUM, RETURN, IF, ELSE, WHILE, PRINTLN, WAIN, INT, NEW, NULL, DELETE}
- {EQ, NE, LT, LE, GT, GE, BECOMES}

Tokens that contain letters are case-sensitive; for example, `int` is an INT token, while `Int` is not.

Context-free Syntax

A context-free grammar for a valid WLPP program is:

- terminal symbols: the set of valid tokens above
- nonterminal symbols: {procedure, type, dcl, dcls, statements, expr, statement, test, term, factor}
- start symbol: procedure
- production rules:

```
procedure → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE
type → INT
type → INT STAR
dcls →
dcls → dcls dcl BECOMES NUM SEMI
dcls → dcls dcl BECOMES NULL SEMI
dcl → type ID
statements →
statements → statements statement
statement → lvalue BECOMES expr SEMI
statement → IF LPAREN test RPAREN LBRACE statements RBRACE ELSE LBRACE statements RBRACE
statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE
statement → PRINTLN LPAREN expr RPAREN SEMI
statement → DELETE LBRACK RBRACK expr SEMI
test → expr EQ expr
test → expr NE expr
test → expr LT expr
test → expr LE expr
test → expr GE expr
test → expr GT expr
expr → term
expr → expr PLUS term
expr → expr MINUS term
term → factor
term → term STAR factor
term → term SLASH factor
term → term PCT factor
factor → ID
factor → NUM
factor → NULL
factor → LPAREN expr RPAREN
factor → AMP lvalue
factor → STAR factor
factor → NEW INT LBRACK expr RBRACK
lvalue → ID
lvalue → STAR factor
lvalue → LPAREN lvalue RPAREN
```

Context-sensitive Syntax

Any ID in a sequence derived from `dcl` is said to be *declared*. Any ID derived from `factor` or `lvalue` is said to be *used*. Any particular string *x* that is an ID may be declared at most once. A string *x* which is an ID may be used in any number of places, but only if the same string *x* is declared. String comparisons are case sensitive; for example, "FOO" and "foo" are distinct.

Instances of the tokens ID, NUM, NULL and the non-terminals `factor`, `term`, `expr`, and `lvalue` have a *type*, which is either `int` or `int*`. Types must satisfy the following rules:

- The type of an ID is `int` if the `dcl` in which the ID is declared derives a sequence containing a `type` that derives `INT`.
- The type of an ID is `int*` if the `dcl` in which the ID is declared derives a sequence containing a `type` that derives `INT STAR`.
- The type of a NUM is `int`.
- The type of a NULL token is `int*`.
- The type of a `factor` deriving ID, NUM, or NULL is the same as the type of that token.
- The type of an `lvalue` deriving ID is the same as the type of that ID.
- The type of a `factor` deriving LPAREN `expr` RPAREN is the same as the type of the `expr`.
- The type of an `lvalue` deriving LPAREN `lvalue` RPAREN is the same as the type of the derived `lvalue`.
- The type of a `factor` deriving AMP `lvalue` is `int*`. The type of the derived lvalue (i.e. the one preceded by AMP) must be `int`.
- The type of a `factor` or `lvalue` deriving STAR `factor` is `int`. The type of the derived factor (i.e. the one preceded by STAR) must be `int*`.
- The type of a `factor` deriving NEW INT LBRACK `expr` RBRACK is `int*`. The type of the derived `expr` must be `int`.
- The type of a `term` deriving `factor` is the same as the type of the derived `factor`.
- The type of a `term` directly deriving anything other than just `factor` is `int`. The `term` and `factor` directly derived from such a `term` must have type `int`.
- The type of an `expr` deriving `term` is the same as the type of the derived `term`.
- When `expr` derives `expr` PLUS `term`:
  - The derived `expr` and the derived `term` may both have type `int`, in which case the type of the `expr` deriving them is `int`.
  - The derived `expr` may have type `int*` and the derived `term` may have type `int`, in which case the type of the `expr` deriving them is `int*`.
  - The derived `expr` may have type `int` and the derived `term` may have type `int*`, in which case the type of the `expr` deriving them is `int*`.
- When `expr` derives `expr` MINUS `term`:
  - The derived `expr` and the derived `term` may both have type `int`, in which case the type of the `expr` deriving them is `int`.
  - The derived `expr` may have type `int*` and the derived `term` may have type `int`, in which case the type of the `expr` deriving them is `int*`.
  - The derived `expr` and the derived `term` may both have type `int*`, in which case the type of the `expr` deriving them is `int`.
- The second `dcl` in the sequence directly derived from `procedure` must derive a `type` that derives INT.
- The `expr` in the sequence directly derived from `procedure` must have type `int`.
- When `statement` derives `lvalue` BECOMES `expr` SEMI, the derived `lvalue` and the derived `expr` must have the same type.
- When `statement` derives PRINTLN LPAREN `expr` RPAREN SEMI, the derived `expr` must have type `int`.
- When `statement` derives DELETE LBRACK RBRACK `expr` SEMI, the derived `expr` must have type `int*`.
- Whenever `test` directly derives a sequence containing two `expr`s, they must both have the same type.
- When `dcls` derives `dcls dcl` BECOMES NUM SEMI, the derived `dcl` must derive a sequence containing a `type` that derives INT.
- When `dcls` derives `dcls dcl` BECOMES NULL SEMI, the derived `dcl` must derive a sequence containing a `type` that derives INT STAR.

Semantics

Any WLPP program that obeys the lexical, context-free, and context-sensitive syntax rules above is a also a valid C++ program fragment. The meaning of the WLPP program is defined to be identical to that of the C++ program formed by inserting the WLPP program at the indicated location in one of the following C++ program shells:

- When the first `dcl` in the sequence directly derived from `procedure` derives a `type` that derives INT, the WLPP program is inserted into the following shell:

```
int wain(int, int);
void println(int);
```

```
// === Insert WLPP Program Here ===

#include <stdlib.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int a,b,c;
  printf("Enter first integer: ");
  scanf("%d", &a);
  printf("Enter second integer: ");
  scanf("%d", &b);
  c = wain(a,b);
  printf("wain returned %d\n", c);
  return 0;
}
void println(int x){
   printf("%d\n",x);
}
```

- When the first `dcl` in the sequence directly derived from `procedure` derives a `type` that derives INT STAR, the WLPP program is inserted into the following shell:

```
int wain(int*, int);
void println(int);

// === Insert WLPP Program Here ===

#include <stdlib.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int l, c;
  int* a;
  printf("Enter length of array: ");
  scanf("%d", &l);
  a = (int*) malloc(l*sizeof(int));
  for(int i = 0; i < l; i++) {
    printf("Enter value of array element %d: ", i);
    scanf("%d", a+i);
  }
  c = wain(a,l);
  printf("wain returned %d\n", c);
  return 0;
}
void println(int x){
   printf("%d\n",x);
}
```