CS 241 - WLP4 Programming Language Specification

The WLP4 programming language contains a strict subset of the features of C++. A WLP4 source file contains a WLP4 program, which is a sequence of procedure definitions, ending with the main procedure `wain`.

Lexical Syntax

A WLP4 program 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", "putchar", "getchar", "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; the numeric value of a NUM token cannot exceed $2^{31}-1$
- 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"
- PUTCHAR: the string "putchar"
- GETCHAR: the string "getchar"
- 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"

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

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

WLP4 programs are constructed by *tokenizing* (also called *scanning* or *lexing*) an ASCII string. To ensure a unique sequence of tokens is produced, the sequence of tokens is constructed by repeatedly choosing the *longest prefix* of the input that is either a token or white space. If the prefix is a token, it is added to the end of the WLP4 program token sequence. Then the prefix is discarded, and this process repeats with the remainder of the ASCII string input. This continues until either the end of the input is reached, or no prefix of the remaining input is a token or white space. In the latter case, the ASCII string is *lexically invalid* and does not represent a WLP4 program.

Context-Free Syntax

A context-free grammar for a valid WLP4 program is:

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

```
procedures → procedure procedures
procedures → main
procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE
main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE
params →
params → paramlist
paramlist → dcl
paramlist → dcl COMMA paramlist
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 → PUTCHAR 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
factor → GETCHAR LPAREN RPAREN
factor → ID LPAREN RPAREN
factor → ID LPAREN arglist RPAREN
arglist → expr
arglist → expr COMMA arglist
lvalue → ID
lvalue → STAR factor
lvalue → LPAREN lvalue RPAREN
```

Context-Sensitive Syntax

Errors in context-sensitive syntax are referred to as *semantic errors* below. A program that contains a semantic error is not a valid WLP4 program and cannot be compiled.

Names & Identifiers

A *procedure* is any string derived from `procedure` or `main`. If it is derived from `procedure`, the name of the procedure is the lexeme of the `ID` in the grammar rule whose left-hand side is `procedure`. The name of the procedure derived from `main` is `wain`. A procedure is said to be *declared* from the first occurrence of its name in the string that makes up that procedure (i.e., once the name has been encountered in the procedure's header). The following semantic errors exist related to procedure declarations:

- Two procedures with the same name cannot be declared.
- A procedure cannot be called until it has been declared. (Formally, the ID in `factor → ID LPAREN RPAREN` or `factor → ID LPAREN arglist RPAREN` must be the name of a procedure that has been declared).

Thus, a procedure (other than `wain`) may call itself recursively, and a procedure may call procedures declared before itself, but a procedure may not call procedures declared after itself. Consequently, there is no mutual recursion in WLP4.

The procedure `wain` may not call itself recursively. However, this is actually enforced by the context-free grammar (since `wain` is a keyword, not an identifier, and therefore cannot appear as the ID in the procedure call rules `factor → ID LPAREN RPAREN` and `factor → ID LPAREN arglist RPAREN`) and therefore is not considered a semantic error.

Any ID in a sequence derived from `dcl` within a procedure `p` is said to be *declared in p*. Any ID derived from `factor` or `lvalue` within p is said to be *used in p*. The *name* of the ID is the lexeme of the ID token. String comparisons between names are are case sensitive; for example, "FOO" and "foo" are distinct.

The following semantic errors exist related to declarations and uses of IDs in a procedure.

- Two IDs with the same name cannot be declared within the same procedure. (However, declaring two IDs with the same name in different procedures is allowed.)
- An ID cannot be used in a procedure unless it is declared within the same procedure.

An ID may have the same name as a procedure. If an ID `x` is declared in a procedure `p`, all occurrences of `x` within `p` refer to the ID x, even if a procedure named `x` has been declared. The same is true in the special case that `p` = `x`: a declared ID may have the same name as the procedure that contains it; in this case, all occurrences of ID refer to the variable, not the procedure. This rule means there is an additional semantic error related to procedures and IDs:

- A procedure `x` cannot be called from within a procedure `p` if there is an ID `x` declared in `p`. (Formally, if a procedure call rule `factor → ID LPAREN RPAREN` or `factor → ID LPAREN arglist RPAREN` occurs in the procedure `p`, the ID of the procedure being called cannot be the name of an ID that is declared in `p`).

For example, the following program does not contain any semantic errors. The declaration of the ID `p` as a parameter of the procedure `p`, and the use of the ID `p` in the return expression of procedure `p`, are not issues, because within the procedure `p` the ID `p` refers to the parameter variable. Within `wain`, the ID `p` refers to the procedure, since no ID named `p` is declared in `wain`, so the procedure call `p(a)` is valid.

```
int p(int p) { return p; }
int wain(int a, int b) { return p(a); }
```

On the other hand, the following program contains a semantic error. Th only difference is the name of the second parameter of `wain` has been changed from `b` to `p`. Therefore, there is now an ID `p` declared in `wain`, so the procedure call `p(a)` in the return expression of `wain` is not valid.

```
int p(int p) { return p; }
int wain(int a, int p) { return p(a); }
```

Types

An ID whose name occurs in a sequence derived from `dcl` has a *type*, which is either `int` or `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`.
- 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`.

Other IDs (particularly, the IDs corresponding to procedure names) do not have types and are said to be *untyped*.

Every procedure has a *signature*, which is a list of strings, each of which is either `int` or `int*`. The signature of a procedure is the sequence of strings `int` or `int*` that is derived from `params`. Note that this sequence may be empty. The signature indicates the number of arguments expected by the procedure, and the type of each argument.

Instances of the tokens NUM, NULL and the nonterminals `factor`, `term`, `expr`, and `lvalue` also have a *type*, which is either `int` or `int*`. The types of these tokens and nonterminals are determined by the following rules. If the conditions of a rule are not satisfied, the program contains a semantic error.

- The type of a NUM is `int`.
- The type of a NULL token is `int*`.
- The type of a `factor` deriving NUM or NULL is the same as the type of that token.
- When `factor` derives ID, the derived ID must have a type, and the type of the `factor` is the same as the type of the ID.
- When `lvalue` derives ID, the derived ID must have a type, and the type of the `lvalue` is the same as the type of the 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 `factor` deriving GETCHAR LPAREN RPAREN is `int`.
- The type of a `factor` deriving ID LPAREN RPAREN is `int`. The procedure whose name is ID must have an empty signature.
- The type of a `factor` deriving ID LPAREN `arglist` RPAREN is `int`. The procedure whose name is ID must have a signature whose length is equal to the number of `expr` strings (separated by COMMA) that are derived from `arglist`. Furthermore, the types of these `expr` strings must exactly match, in order, the types in the procedure's signature.
- 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`.

Additionally, all of the following conditions must be satisfied, or the program contains a semantic error.

- The second `dcl` in the sequence directly derived from `main` must derive a `type` that derives INT.
- The `expr` in the sequence directly derived from `procedure` must have type `int`.
- The `expr` in the sequence directly derived from `main` 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 PUTCHAR 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.

Behaviour

Any WLP4 program that obeys the lexical, context-free, and context-sensitive syntax rules above is a also a valid C++ program fragment. The behaviour of the WLP4 program is generally expected to be the same as the C++ program formed by inserting the WLP4 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 WLP4 program is inserted into the following shell:

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

// === Insert WLP4 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 WLP4 program is inserted into the following shell:

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

// === Insert WLP4 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);
}
```

Note that `putchar` and `getchar` are provided by the C standard library, and their behaviour is as described by C.

There are some situations where the expected behaviour of a WLP4 program may differ from that of the C++ program shells above:

- Failed allocation with `new`. In C++, this throws an exception. Since WLP4 does not have exceptions, a failed allocation in WLP4 returns NULL.
- Dereferencing a NULL pointer. In C++ this usually crashes the program, but technically the behaviour is undefined, so anything can happen. In WLP4 we *require* a NULL dereference to crash the program.
- Pointer arithmetic expressions that fall outside the bounds of the relevant array, e.g., `array+241` where the array has fewer than 241 elements. This is undefined behaviour in C++, but it is allowed in WLP4, because it is used by Marmoset to test your implementation of pointer comparisons. The expected behaviour in WLP4 is that the result is the same as other (non-out-of-bounds) pointer arithmetic expressions. However, *dereferencing* an out-of-bounds pointer is undefined behaviour in both C++ and WLP4.
- Other behaviour which is undefined in C++ is generally also considered to be undefined in WLP4. A WLP4 program compiled with `wlp4c` that contains undefined behaviour may behave differently from the C++ program shells.