The BPL Programming Language

A Reference Manual and Programmer's Guide

January 25, 2016

BPL is the source language that we will develop a compiler for in CSCI 331. It is based on the programming language C- from Compiler Construction: Principles and Practice by Kenneth Louden (PWS Publishing Company, 1997). There are some significant differences and we will not use Louden's scheme for compiling it.

Overview: BPL is a simple. C-like programming language whose datatypes include integers, strings, pointers and arrays.

Declarations: A BPL program consists of a series of declarations – functions, variables, and arrays. All of the objects declared at the top level have as their scope the remainder of the program – they are global, but everything must be declared before it is used. As with C and Java, one of the functions must be called main( ) and must take no arguments. Note that there is no way to make a constant in BPL.

Functions must indicate their return type; they can take any number of arguments. Functions are block-structured, with curly braces delimiting blocks. Declarations of local variables must occur at the top of a block; the block forms the scope of these variables. Local variables can be integers, strings, pointers or arrays; you cannot make a local function. Here, for example, is a declaration of a function that sums the integers from a to b:

```
int factorial(int a, int b ) {
        int n;
        int total;
        n = a;
        total = 0;
        while (n <= b) {
                total = total + n;
                n = n + 1;
        }
        return total;
}
```

Functions can call other functions (that have already been declared) and can be recursive. All arguments to functions (integers, strings, pointers and arrays) are passed by value.

Data Types: There are six possible types in BPL:

- int
- string
- pointer to int
- pointer to string
- int array
- string array

Integers have the usual arithmetic operators (+, *, -, / and %) as well as the usual comparison operators (<=, <, ==, !=, >, >=). There are read( ) and write( ) operations for integers. Strings are relatively useless; there are no string operators and the only things you can do with strings are to assign them, return them, and print them. Pointers hold addresses, but there is no "poiner arithmetic" as there is in C. The only real use for pointers is in passing pointers to variables into functions, which can then assign to those variables, as a kind of homemade call-by-reference. Arrays are the typical C-style or Java-style array. The length of an array must be known at compile time: you can declare an array as

```
int A[100];
```

but not

```
int A[length];
```

or even

```
int A[50+50];
```

A BPL implementation must perform run-time bounds checking on array indices. Rather than crashing, a BPL program should halt with an error message if an array index is negative or as large or larger than the allocated size of the array.

Input and Output: There are 3 I/O expressions in BPL:

- write(exp) evaluates exp, which should give either an int or a string, and prints it, followed by a space.
- writeln( ) terminates the current line of output
  The statements

```
write( 23 );
write( 45 );
writeln( );
write( 14 );
writeln( );
```

  produce

```
23 45
14
```

- read( ) reads the next item on standard input and tries to interpret it as an integer, which it returns.

Comments: Any text between delimiters /* and */ is considered a comment and ignored by the compiler

Sample Programs: Here are several programs in BPL;

Example 1: Here is a simple factorial program:

```
/* A program to compute factorials */

int fact( int n) {
    if (n == 0)
        return 1;
    else
        return n*fact(n-1);
}

void main(void) {
    int x;
    x = 1;
    while (x < 10) {
        write(x);
        write(fact(x));
        writeln();
        x = x + 1;
    }
}
```

Example 2: The following program inputs a list of 10 integers, sorts them using SelectionSort, and prints the list in sorted order.

```
/* A program to input, sort, and output. */
int x[10];

void switch (int A[], int i, int j) {
    int temp;
    if (i != j) {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
    }
}

void sort( int A[], int first, int last ) {
    int i;
    int j;
    int small;

    i = first;
    while (i < last-1) {
            /* get smallest remaining value and put it at position i */
            j = i;
            small = j;
            while (j < last) {
                    if (A[j] < A[small])
                            small = j;
                    j = j+1;
            }
            switch(A, i, small);
            i = i + 1;
    }
}
```

```
void main(void) {

        int i;

        i = 0;
        while (i < 10) {
                x[i] = read();
                i = i+1;
        }
        sort(x, 0, 10);
        i = 0;
        while (i < 10) {
                write(x[i])
                i = i + 1;
        }
        writeln();
    }
```

Example 3: The following program is just a reminder that we have pointers:

```
/* This outputs 23 */
void f( int *x ) {
    *x = 23;
}

void main(void) {
    int a;
    int *b;
    b = &a;
    a = 4;
    f( b );
    write(a);
    writeln();
}
```

A BNF Grammar for BPL:

Capital letters indicate a non-terminal grammar symbol, as in PROGRAM or STATEMENT
Lower-case letters indicate a terminal grammar symbol, such as **int** or **void**.
Brackets identifiers are placeholders: **<id>**, **<num>** and **<empty>**
Punctuation marks indicate terminal grammar symbols: **({ ;** etc.

1. PROGRAM -> DECLARATION_LIST
2. DECLARATION_LIST -> DECLARATION_LIST DECLARATION | DECLARATION
3. DECLARATION -> VAR_DEC | FUN_DEC
4. VAR_DEC -> TYPE_SPECIFIER < **id> ;**
   | TYPE_SPECIFIER *<id>
   | TYPE_SPECIFIER **<id>[<num>] ;**
5. TYPE_SPECIFIER -> **int** | **void** | **string**
6. FUN_DEC -> TYPE_SPECIFIER **<id> (** PARAMS **)** COMPOUND_STMT
7. PARAMS -> **void** | PARAM_LIST
8. PARAM_LIST -> PARAM_LIST **,** PARAM | PARAM
9. PARAM -> TYPE_SPECIFIER **<id>**
   | TYPE_SPECIFIER ***<id>**
   |TYPE_SPECIFIER **<id>[ ]**
10. COMPOUND_STMT -> **{** LOCAL_DECS STATEMENT_LIST **}**
11. LOCAL_DECS -> LOCAL_DECS VAR_DEC | **<empty>**
12. STATEMENT_LIST -> STATEMENT_LIST STATEMENT | **<empty>**
13. STATEMENT -> EXPRESSION_STMT
    | COMPOUND_STMT
    | IF_STMT
    | WHILE_STMT
    | RETURN_STMT
    | WRITE_STMT
14. EXPRESSION_STMT -> EXPRESSION **;** | **;**
15. IF_STMT -> **if (** EXPRESSION **)** STATEMENT
    | **if (** EXPRESSION **)** STATEMENT **else** STATEMENT
16. WHILE_STMT -> **while (** EXPRESSION **)** statement
17. RETURN_STMT -> **return ;** | **return** EXPRESSION **;**
18. WRITE_STMT -> **write (** EXRESSION **) ;** | **writeln ( ) ;**
19. EXPRESSION -> VAR **=** EXPRESSION | COMP_EXP
20. VAR -> <**id>** | <**id>[** EXPRESSION] | ***<id>**
21. COMP_EXP -> E RELOP E | E
22. RELOP -> **<=** | **<** | **==** | **!=** | **>** | **>=**
23. E -> E ADDOP T | T
24. ADDOP -> **+** | **-**
25. T -> T MULOP F | F
26. MULOP -> ***** | **/** | **%**
27. F -> -F | &Factor | *Factor | Factor
28. Factor -> **(** EXPRESSION **)** | FUN_CALL | **read ( )** | ***<id>** | <**id>** | <**id>**[EXPRESSION] | **<num>** | **<string>**
29. FUN_CALL -> **<id> (** ARGS **)**
30. ARGS -> ARG_LIST | **<empty>**
31. ARG_LIST -> ARG_LIST **,** EXPRESSION | EXPRESSION

Discussion of the grammar:

1. PROGRAM -> DECLARATION_LIST
2. DECLARATION_LIST -> DECLARATION_LIST DECLARATION | DECLARATION
3. DECLARATION -> VAR_DEC | FUN_DEC

A program consists of a sequence of declarations. Both variables and functions need to be declared before they are used; there are no forward references. The last item declared must be the function main()

4. VAR_DEC -> TYPE_SPECIFIER **< id> ;**
   | TYPE_SPECIFIER *<id>
   | TYPE_SPECIFIER **<id>[<num>] ;**
5. TYPE_SPECIFIER -> **int** | **void** | **string**

You can declare string or int variables, string or int pointer variables, and string or int arrays. As with most languages, arrays are indexed 0..length-1. Notes that only one variable can be declared in a declaration. Array sizes are literal numbers, not expressions. void is listed as a type specifier, but is only used as the return type of a function that does not return a value.

6. FUN_DEC -> TYPE_SPECIFIER **<id> (** PARAMS **)** COMPOUND_STMT
7. PARAMS -> **void** | PARAM_LIST
8. PARAM_LIST -> PARAM_LIST **,** PARAM | PARAM
9. PARAM -> TYPE_SPECIFIER **<id>**
   | TYPE_SPECIFIER ***<id>**
   | TYPE_SPECIFIER **<id>[ ]**

The parameter list for a function declaration can either be the word **void** or else a comma-separated list of identifiers. All arguments are passed by value (remember that the value of an array is its starting address.) Functions may be recursive. If the return type is not void the function body should contain a return statement that returns a value of the return type.

10. COMPOUND_STMT -> **{** LOCAL_DECS STATEMENT_LIST **}**
11. LOCAL_DECS -> LOCAL_DECS VAR_DEC | **<empty>**

A compound statement creates a block in the program. It may have its own declarations, whose scope is the extent of the block.

12. STATEMENT_LIST -> STATEMENT_LIST STATEMENT | **<empty>**
13. STATEMENT -> EXPRESSION_STMT
    | COMPOUND_STMT
    | IF_STMT
    | WHILE_STMT
    | RETURN_STMT
    | WRITE_STMT
14. EXPRESSION_STMT -> EXPRESSION **;** | **;**

Assignments and function calls are both expressions; these sometimes need to be used as statements. Note that a single semicolon counts as an expression statement. You need to handle such "empty" statements.

15. IF_STMT -> **if (** EXPRESSION **)** STATEMENT
    | **if (** EXPRESSION **)** STATEMENT **else** STATEMENT

This is the usual if-statement. Note that there is no Boolean type; a 0-value for the expression is interpreted as false, a non-zero value as true. A "dangling else" is resolved in the usual way – an else clause is attached to the nearest un-elsed if.

16. WHILE_STMT -> **while (** EXPRESSION **)** statement

The only loop in BPL is the while loop. The expression is evaluated; if its value is non-zero the body statement is evaluated and the expression is evaluated again. This continues until the expression evaluates to 0.

17. RETURN_STMT -> **return ;** | **return** EXPRESSION **;**

Functions declared void must not return values; functions not declared void must return values. A return statement inside of the main( ) function causes execution to be terminated.

18. WRITE_STMT -> **write (** EXRESSION **) ;** | **writeln ( ) ;**

Unlike most modern language which relegate I/O to libraries, BPL makes I/O an inherent part of the language. The write-statement writes a single value, which may be either integer or string, on the current line of output. writeln( ) terminates the current line of output and moves to the next line.

19. EXPRESSION -> VAR = EXPRESSION | COMP_EXP
20. VAR -> **id** | **id[** EXPRESSION] | *****<**id**>

This syntax for assignment statements allows for chained assignments: since x = 5 is an expression (and so returns the valued assigned), we may say y = x = 5. As in C, if x has type "pointer to int" we may assign 5 to the location x references by *x = 5. The language does not require bounds checking on array indices – behavior in the case of inappropriate indices is unspecified.

21. COMP_EXP -> E RELOP E | E
22. RELOP -> <= | < | == | **!=** | > | >=

The relational operators return values 1 and 0 (for true and false). Note that an unparenthesized expression can contain only one relational operator and there are no local connectives for "boolean" expressions.

23. E -> E ADDOP T | T
24. ADDOP -> **+** | **-**
25. T -> T MULOP F | F
26. MULOP -> ***** | **/** | **%**
27. F -> **-**F | & Factor | *Factor | Factor
28. Factor -> **(** EXPRESSION **)**
    | FUN_CALL
    | **read ( )**
    | ***<id>**
    | <**id**>
    | <**id**>[EXPRESSION]
    | **<num>**
    | **<string>**

Arithmetic expressions are defined only for integer values. These grammar rules give the usual associativity and precedence rules for arithmetic. Note that the only numeric type is integer, so the division operator / produces integer division, dropping any remainder. % is the modulus, or remainder, operator. The & operation obtains the address its operand; you can only take the address of variables and array elements. The * operation dereferences an address; this is only valid for operands of type "pointer to int" and "pointer to string". The read( ) expression expects to see an integer on the input stream and returns this value.

29. FUN_CALL -> **<id> (** ARGS **)**
30. ARGS -> ARG_LIST | **<empty>**
31. ARG_LIST -> ARG_LIST **,** EXPRESSION | EXPRESSION

The arguments to a function call must match the function declaration in both number and type. Functions must be declared before they are called.

For your convenience, here are some handy lists:

The keywords of BPL are

int void string if else while return write writeln read

These are all reserved words; they may not be used as variables.

The special symbols and punctuation marks of BPL are

; , [ ] { } ( ) < <= == != >= > + - * / = % & /* */