Appendix B – The B-Minor Language

B.1 Overview

The B-Minor language is a "little" language suitable for use in an undegraduate compilers class. B-Minor includes expressions, basic control flow, recursive functions, and strict type checking. It is object-code compatible with ordinary C and thus can take advantage of the standard C library, within its defined types.

B-Minor is similar enough to C that it should feel familiar, but has enough variations to allow for some discussion of different language choices affect the implementation. For example, the type syntax of B-Minor is closer to that of Pascal or SQL than of C. Students may find this awkward at first, but its value becomes clearer when constructing a parser and when discussing types independently of the symbols that they apply to. The `print` statement gives an opportunity to perform simple type inference and interact with runtime support. A few unusual operators cannot be implemented in a single assembly instruction, illustrating how complex language intrinsics are implemented. The strict type system gives the students some experience with reasoning about rigorous type algebras and producing detailed error messages.

A proper language definition would be quite formal, including regular expressions for each token type, a context-free-grammar, a type algebra, and so forth. However, if we provided all that detail, it would rob you (the student) of the valuable experience of wrestling with those details. Instead, we will describe the language through examples, leaving it to you to read carefully, and then extract the formal specifications needed for your code. You are certain to find some details and corner cases that are unclear or incompletely specified. Use that as an opportunity to ask questions during class or office hours and work towards a more precise specification.

B.2 Tokens

In B-Minor whitespace is any combination of the following characters: tabs, spaces, linefeed, and carriage return. The placement of whitespace is not significant in B-MinorḂoth C-style and C++-style comments are valid in B-Minor

```
/* A C-style comment */
a=5; // A C++ style comment
```

Identifiers (i.e. variable and function names) may contain letters, numbers, and underscores. Identifiers must begin with a letter or an underscore. These are examples of valid identifiers:

```
i x mystr fog123 BigLongName55
```

The following strings are B-Minor keywords and may not be used as identifiers:

```
array boolean char else false for function if
integer print return string true void while
```

B.3 Types

B-Minor has four atomic types: integers, booleans, characters, and strings. A variable is declared as a name followed by a colon, then a type and an optional initializer. For example:

```
x: integer;
y: integer = 123;
b: boolean = false;
c: char    = 'q';
s: string  = "hello world\n";
```

An `integer` is always a signed 64 bit value. `boolean` can take the literal values `true` or `false`. `char` is a single 8-bit ASCII character. `string` is a double-quoted constant string that is null-terminated and cannot be modified. (Note that, unlike C, `string` is not an array of `char`, it is a completely separate type.)

Both `char` and `string` may contain the following backslash codes. `n` indicates a linefeed (ASCII value 10), `0` indicates a null (ASCII value zero), and a backslash followed by anything else indicates exactly the following character. Both strings and identifiers may be up to 256 characters long. ¡p¿ B-Minor also allows arrays of a fixed size. They may be declared with no value, which causes them to contain all zeros:

```
a: array [5] integer;
```

Or, the entire array may be given specific values:

```
a: array [5] integer = {1,2,3,4,5};
```

B.4 Expressions

B-Minor has many of the arithmetic operators found in C, with the same meaning and level of precedence:

`[] f()`	array subscript, function call
`++ --`	postfix increment, decrement
`- !`	unary negation, logical not
`^`	exponentiation
`* / %`	multiplication, division, modulus
`+ -`	addition, subtraction
`< <= > >= == !=`	comparison
`&& ||`	logical and, logical or
`=`	assignment

B-Minor is *strictly typed*. This means that you may only assign a value to a variable (or function parameter) if the types match *exactly*. You cannot perform many of the fast-and-loose conversions found in C. For example, arithmetic operators can only apply to integers. Comparisons can be performed on arguments of any type, but only if they match. Logical operations can only be performed on booleans.

Following are examples of some (but not all) type errors:

```
x: integer = 65;
y: char = 'A';
if(x>y) ... // error: x and y are of different types!

f: integer = 0;
if(f) ...      // error: f is not a boolean!

writechar: function void ( char c );
a: integer = 65;
writechar(a);  // error: a is not a char!

b: array [2] boolean = {true,false};
x: integer = 0;
x = b[0];      // error: x is not a boolean!
```

Following are some (but not all) examples of correct type assignments:

```
b: boolean;
x: integer = 3;
y: integer = 5;
b = x<y;     // ok: the expression x<y is boolean

f: integer = 0;
```

```
if(f==0) ...    // ok: f==0 is a boolean expression

c: char = 'a';
if(c=='a') ...  // ok: c and 'a' are both chars
```

B.5 Declarations and Statements

In B-Minor you may declare global variables with optional constant initializers, function prototypes, and function definitions. Within functions, you may declare local variables (including arrays) with optional initialization expressions. Scoping rules are identical to C. Function definitions may not be nested.

Within functions, basic statements may be arithmetic expressions, `return` statements, `print` statements, `if` and `if-else` statements, `for` loops, or code within inner groups. B-Minor does not have switch statements, while-loops, or do-while loops, since those are easily represented as special cases of `for` and `if`.

The `print` statement is a little unusual because it is a statement and not a function call. `print` takes a list of expressions separated by commas, and prints each out to the console, like this:

```
print "The temperature is: ", temp, " degrees\n";
```

Note that each element in the list following a `print` statement is an expression of *any* type. The `print` mechanism will automatically infer the type and print out the proper representation.

B.6 Functions

Functions are declared in the same way as variables, except giving a type of `function` followed by the return type, arguments, and code:

```
square: function integer ( x: integer ) = {
    return x^2;
}
```

The return type of a function must be one of the four atomic types, or `void` to indicate no type. Function arguments may be of any type. `integer`, `boolean`, and `char` arguments are passed by value, while `string` and `array` arguments are passed by reference. As in C, arrays passed by reference have an indeterminate size, and so the length is typically passed as an extra argument:

```
printarray: function void
  ( a: array [] integer, size: integer ) = {
    i: integer;
    for( i=0;i<;size;i++) {
        print a[i], "\n";
    }
}
```

A function prototype states the existence and type of the function, but includes no code. This must be done if the user wishes to call an external function linked by another library. For example, to invoke the C function `puts`:

```
puts: function void ( s: string );

main: function integer () = {
    puts("hello world");
}
```

A complete program must have a `main` function that returns an integer. the arguments to `main` may either be empty, or use `argc` and `argv` in the same manner as C. (The declaration of `argc` and `argv` is left as an exercise to the reader.)

B.7 Optional Elements

Creating a complete implementation of the language above from beginning to end should be more than enough to keep a undergraduate class busy for a semester. However, if you need some additional challenge, consider the following ideas:

- Add a new native type `complex` which implements complex numbers. To make this useful, you will need to add some additional functions or operators to construct complex values, perform arithmetic, and extract the real and imaginary parts.
- Improve the safety of arrays by making the array accesses automatically checked at runtime against the known size of the array. This requires making the length of the array a runtime property stored in memory alongside the array data, checking each array access against the boundaries, and taking appropriate action on a violation. Compare the performance of checked arrays against unchecked arrays. (The X86 BOUND instruction might be helpful.)
- Add a new mutable string type `mutstring` which has a fixed size, but can be modified in place, and can be converted to and from a regular `string` as needed.

- Add an alternative control flow structure like `switch`, which evaluates a single control expression, and then branches to alternatives with matching values. For an extra challenge, allow `switch` to select value ranges, not just constant values.
- Implement structure types that allow multiple data items to be grouped together in a simple type. At the assembly level, this is not very different from implementing arrays, because each element is simply at a known offset from the base object. However, parsing and type-checking become more complicated because the elements associated with a structure type must be tracked.