BitTwiddler

a language for binary data parsers

Language Reference Manual

Programming Languages and Translators
COMS W4115 – Fall 2018
Bruno Martins – bm2787

Table of Contents

1. Lexical Conventions

1.1. Comments

Comments start with the character **#** and continue until the end of the line. Comments can only be single line. Example:

```
# This is a comment
template Number {
    var _ : uint32;       # This is also a comment
}
```

1.2. Identifiers

Identifiers are composed of ASCII letters, numbers and the underscore character. They must start with a lower case letter or an underscore. Identifiers are case sensitive. Examples:

```
var name : string;      # name is a valid identifier
var _nAmE_123 : string; # Also valid
```

1.3. Template and type Identifiers

Types that are not the built-in primitive types are composed of ASCII letters, numbers and the underscore character. It must start with an upper case letter. Example:

```
template Number {
    _ : uint32;
}
```

A few other built-in types exist to represent language constructs:

None	Unit type.
Type	A variable of type Type can hold a reference to a type name.
Array	Alias to array types.
Func	A variable can hold a reference to a function. This would be its type.

1.4. Keywords

The following keywords are reserved by BitTwiddler:

```
 int8     int8le     int8be    float32    var     and
uint8    uint8le    uint8be    float64    return  or
 int16    int16le    int16be   bit        if      not
uint16   uint16le   uint16be   string     else    None
 int32    int32le    int32be   template   elif    Type
uint32   uint32le   uint32be   parse      for     Array
 int64    int64le    int64be   match      in      Func
uint64   uint64le   uint64be   func       while
```

1.5. Literal Constants

1.5.1. Integers

Integer literals can be declared as decimal, hexadecimal or binary numbers. Hexadecimal numbers are prefixed by **`0x`** and binary numbers by **`0b`**. Examples:

```
42       0x2a       0b101010
```

1.5.2. Floats

Floating point literals may have four parts to it: the integer part, the decimal point separator, the fractional part and the exponent. The integer part and the fractional part cannot be simultaneously missing. Likewise, the decimal separator and the exponent cannot be simultaneously missing. The integer part and the fractional part are composed of a series of digits. The exponent is composed of the **`e`** or **`E`** character, followed by an optional **`+`** or **`-`** sign, followed by digits. Examples:

```
123e45   .123   .123e45   123.   123.e+45   123.45    1.23e-45
```

1.5.3. Strings

Strings are any sequence of characters enclosed by either two double-quote or two single-quote characters. Examples:

```
"A double-quoted string"     'A single-quoted string'    "Enclosed 'quotes'"
```

1.5.4. Arrays

Array literals can be defined by a sequence of expressions, separated by commas and enclosed by square brackets.

```
array-literal
    [ expression-list_opt ]

expression-list
    expression
    expression-list , expression
```

Examples:

```
var weekdays:string[7] = ['M', 'T', 'W', 'T', 'F'];
var numbers:int32[3] = [1, 2, 3];
```

2. Expressions

An expression in BitTwiddler is any of the following, each of which will be explained in detail in the following sections.

```
expression:
    constant
    identifier
    type-identifier
    ( expression )
    binary-operation
    unary-operation
    function-call
    conditional
    for
    while
    match
```

2.1. Constant

A literal constant expression is composed of any of the literal constants described in the section 1.5 Literal Constants above.

2.2. Identifier and type identifiers

An identifier as described in the section 1.2 Identifiers or a type identifier as described in section 1.3 Template and type Identifiers.

2.3. Parenthesis-enclosed expressions

An expression enclosed by parenthesis. Useful for altering the precedence of evaluation.

2.4. Unary and binary operations

There are a number of binary and unary operations. They are listed here, along with their precedence and associativity.

<table>
<tr><th>Operation</th><th>Name</th><th>Assoc.</th><th>Precedence</th></tr>
<tr><td>expression . expression
expression [expression]</td><td>Access
Subscript</td><td>Left</td><td>1</td></tr>
<tr><td>not expression
~ expression
+ expression
- expression</td><td>Logical not
Bitwise not
Unary plus
Unary minus</td><td>Right</td><td>2</td></tr>
<tr><td>expression * expression
expression / expression
expression % expression</td><td>Multiplication
Division
Remainder</td><td>Left</td><td>3</td></tr>
<tr><td>expression + expression
expression - expression</td><td>Addition
Subtraction</td><td>Left</td><td>4</td></tr>
<tr><td>expression << expression
expression >> expression</td><td>Bitwise left shift
Bitwise right shift</td><td>Left</td><td>5</td></tr>
<tr><td>expression < expression
expression <= expression
expression >= expression
expression > expression</td><td>Logical less than
Logical less than or equal to
Logical greater than or equal to
Logical greater than</td><td>Left</td><td>6</td></tr>
<tr><td>expression == expression
expression != expression</td><td>Logical equal to
Logical not equal to</td><td>Left</td><td>7</td></tr>
<tr><td>expression & expression</td><td>Bitwise and</td><td>Left</td><td>8</td></tr>
<tr><td>expression | expression</td><td>Bitwise or</td><td>Left</td><td>9</td></tr>
<tr><td>expression and expression</td><td>Logical and</td><td>Left</td><td>10</td></tr>
<tr><td>expression or expression</td><td>Logical or</td><td>Left</td><td>11</td></tr>
<tr><td>expression = expression</td><td>Assignment</td><td>Right</td><td>12</td></tr>
</table>

2.5. Function Call

A function call expression has the form:

```
function-call
    id ( expression-list_opt )

expression-list
    expression
    expression-list , expression
```

The value of the function call expression is the value returned by the function being called. Example:

```
sum(1, 1)
```

2.6. Conditional

A conditional expression has the form:

```
conditional
    if elseifs_opt else_opt

if
    if expression block

elseifs:
    elseif
    elseifs elseif

elseif:
    elif expression block

else:
    else block
```

Note that it doesn't require parenthesis around the expression being tested. For a conditional, any value that evaluates to 0 or to empty string is considered false, otherwise it is considered true. Since conditionals are *expressions*, it's value is the value of the last statement executed inside its *block*. Example:

```
if x == 1 {
    "one";
} elif x == 2 {
    "two";
} else {
    "not one nor two";
}
```

2.7. For

The for expression has the following form:

```
for
    for forvars in expression block

forvars
    id
    forvars , id
```

The identifiers in *forvars* will be bound to values destructured from the value of *expression* until there are no more values to be destructured. Example:

```
for i in [1, 2, 3] {
    i;
}
```

Note that since **for** is an expression, its value will be that of the last statement executed. In this example, 3.

2.8. While

The **while** expression repeatedly executes a block of instructions while the evaluated expression remains true.

```
while
    while expression block
```

Example:

```
var i = 0;
while i < 10 {
    i = i + 1;
};
```

2.9. Match

The match expression is similar to C's **`switch`**. Its purpose is to test a single expression against possible values and execute the block associated with the matched expression. If there is no match, the program halts with an error.

```
match
    match expression match-block

match-block
    { match-arms }

match-arms
    match-arm
    match-arms match-arm

match-arm
    expression -> block
```

Example:

```
match 3 {
    1 -> { "one";   }
    2 -> { "two";   }
    3 -> { "three"; }
}
```

3. Blocks, declarations and scope

Blocks are used in a few of BitTwiddler's constructs. They are not expressions; they can only appear in certain specific places. A **declaration** can be either a variable, a function or a template declaration.

```
block
    { block-statements_opt }

block-statements
    block-statement
    block-statements block-statement

block-statement
    expr ;
    decl
    return expr ;

decl
    template
    function
    variable
```

Hence, each block statement can be an expression, a declaration or a **`return`** statement (**`return`** statements are only semantically valid in **`function`** blocks).

Variables, functions and templates that are declared inside a block have the local scope of the block.

3.1. Variables

Variables are used as labels to values in memory. The same syntax is used to define both global and local-scope variables and template fields:

```
variable
    var @opt id : type = expression ;                          (1)
    var @opt id        = expression ;                          (2)
    var @opt id : type              ;                          (3)
    var [ expression ] : [ expression ] = expression ;    (4)
    var [ expression ]                  = expression ;    (5)
    var [ expression ] : [ expression ]              ;    (6)
```

Forms (1)-(3) assign the value of the *expression* on the right hand side to the identifier *id*, with the specified *type*. If *type* is not specified, *id* will take the type of the expression on the right hand side. **If an expression is not specified, then the value is read from the standard input[1].** When declaring fields inside a **template**, the field can be declared *hidden* by prefixing the field id with the **@** character. Hiding a field is semantically invalid everywhere else. The usefulness of hiding a field is explained in the 3.3 Templates section.

Forms (4)-(6) are **exclusive** for templates. In those cases, the *expression* right after the **var** keyword must evaluate, at runtime, to a **string**. The expression right after the **:** character must evaluate, at runtime, to a **Type** value.

1 *This is an unique BitTwiddler feature. If a variable is declared without and immediate value, an appropriate value is read from the standard input.*

3.2. Functions

Functions are named blocks of code that can be executed with parameters. They are defined as:

```
function
    func id parameters_opt : type block

parameters
    ( parameters-list )

parameters-list
    parameter
    parameters-list , parameter

parameter
    id : type
```

Example:

```
func sum (a:int64, b:int64) : int64 {
    a + b;
}
```

3.3. Templates

Template is one of BitTwiddler's unique features. Templates are akin to C **struct**s, but dynamic. A template declaration defines a new type that can be used later as a type for variables. A template consists of typed fields. Syntactically, they are defined as:

```
template
    template type-identifier parameters_opt block
```

3.3.1. Field scope

Variable declarations inside templates have a different semantic meaning. Any variable declared anywhere inside a template block will **not** be local to the block; instead, **it will be a field of the enclosing template**. For example:

```
template Demo {
    var a : uint32;     # no assigned value: it will be read from stdin
    if a == 0 {
        var b : uint32 = 123;
    } else {
        var c : uint32 = 456;
    };
}
```

In the example, **a** will always be a field of the template Demo. Then, either **b** or **c** will be a field of Demo (**not** local variables of the conditional blocks); this will be decided at runtime, depending on the value read in **a**, when Demo is instantiated.

3.3.2. Hiding fields

Field declarations with the @ marker will not be part of the enclosing template. For example:

```
template Demo {
    var@ a : uint32;
    if a == 0 { var b : uint32 = 123; } else { var c : uint32 = 456; };
}
```

Here, **a** will not be part of Demo, only **b** or **c**.

3.3.3. Aliasing

Inside template blocks, the field **_** has a special meaning: it is an alias to the whole template. There can be only one **_** field in a template *and* **_** must be the only visible field inside the template. For example:

```
template LString {
    var@ len : uint32;
    var _ : uint8[len];
}

parse {
    var name : Demo;
}
```

In this example, the program will read 4 bytes into the hidden **len** field, and then **len** bytes into the **_** field. **LString** is an alias to **uint8[]** type, so it can be used wherever a **uint8[]** can be used.

3.3.4. Parameters

Much like functions, templates can also receive parameters when instantiated. For example:

```
template ANumber (n : uint32 ) { var _ : uint32 = n; }
parse { var x:ANumber(42); }
```

4. Program

A BitTwiddler program has the following structure:

```
program
    decls_opt parse block EOF

decls
    decl
    decls decl

decl
    template
    function
    variable
```

Variables, templates and functions declared in *decls* have global scope. The **parse** block is the entry point of the program.

5. Standard library functions

BitTwiddler has a number of functions that are built-in, part of the standard library.

5.1. emit

```
emit (val : string) : None
```

Emits **`val`** to the standard output.

5.2. print

```
print (val : string) : None
```

Prints **`val`** to standard error.

5.3. fatal

```
fatal (val : string) : None
```

Prints **`val`** to standard error; exits from the program.

5.4. enumerate

```
enumerate (val : *) : Array
```

Similar to Python's enumerate. Returns an array of two-element arrays, the first element being a zero-based index, and the second element being a value depending on the type of **val**:

- If **val** is an array, the second element will be the value at that index
- If **val** is a template, the second element will be the name of the field at that index

5.5. typeof

```
typeof (val : *) : Type
```

Returns the type of **val**.

5.6. len

```
len (val : *) : uint64
```

Returns the length of **val**:

- If **val** is an array, returns the length of the array;
- If **val** is a string, returns the number of characters;
- If **val** is a template, returns the number of fields in the template;

5.7. map

```
map (val : Array, f : Func) : Array
```

Returns an array with **f** applied to every value of **val**.

5.8. join

```
join (c : string, val: Array) : string
```

Returns a new concatenated string with values of **`val`** insterspersed with **`c`**.

6. Example Program: self-describing binary data

Consider a hypothetical computer game that stores character attributes in self-describing binary files, and the following content for one of these files encoding a character's name and experience (numbers are in hexadecimal):

<table>
<tr><td>02</td><td>00</td><td>04</td><td>'n'</td><td>'a'</td><td>'m'</td><td>'e'</td><td>01</td><td>02</td><td>'x'</td><td>'p'</td><td>03</td><td>'A'</td><td>'n'</td><td>'n'</td><td>64</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>Two fields</td><td colspan="6">First field
type 0 = string
name = “name”</td><td colspan="4">Second field
type 1 = uint32
name = “xp”</td><td colspan="4">First field
value
"Ann"</td><td colspan="4">Second field
value
100</td></tr>
</table>

```
template AttrString {                # Represents an encoded string
  var@ len : uint8;                  # len will not be a field of AttrString
  var _ : uint8[len];                # AttrString will be an "alias" to uint8[]
}

template AttrDesc {
  var@ typeCode : uint8;
  var type : Type = match typeCode { # If there's no match, the program aborts with an error
    0x00 -> { AttrString; }
    0x01 -> { uint32;     }
  };
  var name : AttrString;
}

template Character(attrs:AttrDesc[]) {
  for attr in attrs {                # Character's field names will come from strings
    var [attr.name] : [attr.type];  # Auto type conversion: AttrString -> uint8[] -> string
  };
}

parse {                              # Entry point
  var numAttrs: uint8;               # Reads in the number of attributes
  var attrs: AttrDesc[numAttrs];     # Reads in the attribute descriptions
  var character: Character(attrs);  # Reads character info based on attribute descriptions

  emit('{');
  for i, attr in enumerate(character) {
    emit('{attr}:');
    match typeof(character.attr) {
      AttrString -> { emit('"{character.attr}"'); }
      uint32     -> { emit('{character.attr}');   }
    };

    if i < len(character) - 1 {          # len of a template is its number of fields
      emit(',');
    };
  }
  emit('}\n');
}
```

7. Example Program: gcd

```
func gcd (a:uint64, b:uint64) : uint64 {
    if b == 0 {
        a;           # return keyword is not necessary
    } else {
        gcd(b, a % b);
    };
}

parse {
     var a : uint64;      # Read inputs from standard input
     var b : uint64;

     var r = gcd(a, b);   # Automatic type for r (uint64)
     emit('gcd({a}, {b}) = {r}\n');
}
```