Language Reference

Compiler usage

Description

Biscuit language compiler is standalone terminal application called blc. It can be compiled from source code found on GitHub repository or downloaded from home page as prebuilded binary executable. All three major operating systems (Windows, macOS and Linux) are supported, but current active development is done on Windows and it usually takes some time to port latest changes to the other platforms. Compiler executable can be found in bin directory it's usually good idea to add executable location to system PATH to be accessible from other locations.

There are several options which can be passed to the compiler.

Compiler options:

Configuration

Use `bl-config` tool to change compiler configuration.

This tool generates configuration file `etc/bl.conf` containing all required information needed by compiler during compilation.

Execution status

- After regular compilation process *blc* return 0 on success or numeric maximum error code on fail.
- When *--run*|*--run-script* flag is specified *blc* return status returned by executed *main* function on success or numeric maximum error code on fail (compilation error or compile time execution error).
- When *--run-tests* flag is specified *blc* return count of failed tests on success or numeric maximum error code on fail.

Language

Base syntax

Basically every construct in bl follows the same rules of declaration syntax. We define name of the entity, type and optionally some initial value. Name can be usually used to reference the entity later in code and type describes layout of data represented by the entity. It could be a number, text or more complex types.

Possible declarations:

```
<name>: <type>;              // mutable declaration
<name>: [type] = <value>;    // mutable declaration
<name>: [type] : <value>;    // immutable declaration (value can be set only once)
```

```
foo: s32;                // integer variable without initial value
name: string = "Martin"; // string variable
name: string : "Martin"; // string constant
```

When we decide to explicitly specify initial value, data type can be inferred from this value. In such case the type is optional.

```
name := "Martin"; // string variable
name :: "Martin"; // string constant
```

Comments

Comment lines will be ignored by compiler.

```
// this is line comment
/*
 this
 is
 multi line
 comment
*/
```

Data types

Fundamental data types

Fundamental types are atomic basic types builtin into BL compiler.

Name	Description
s8	Signed 8-bit number.
s16	Signed 16-bit number.
s32	Signed 32-bit number.
s64	Signed 64-bit number.
u8	Unsigned 8-bit number.
u16	Unsigned 16-bit number.
u32	Unsigned 32-bit number.
u64	Unsigned 64-bit number.
usize	Unsigned 64-bit size.
bool	Boolean. (true/false)
f32	32-bit floating point number.
f64	64-bit floating point number.
string	String slice.

Pointer

Represents the address of some allocated data.

```
*<T>
```

Example:

```
#import "std/test"

pointers :: fn () #test {
```

```
    i := 666;
    i_ptr : *s32 = &i; // taking the address of 'i' variable and set 'i_ptr'
    j := ^i_ptr;       // pointer dereferencing

    test_true(j == i);
};
```

Array

Array is aggregate type of multiple values of the same type. Size value must be known in compile time.

```
[<size>] <T>
```

Arrays can be inline initialized with compound block, type is required. Zero initializer can be used for zero initialization of whole array storage, otherwise we must specify value for every element in an array.

```
{:<T>: [val], ...}
```

Example:

```
array_type :: fn () #test {
    arr1 : [10] s32; // declare zero initialized array variable
    arr1[0] = 666;

    arr1.len; // yields array element count (s64)
    arr1.ptr; // yields pointer to first element '&arr[0]'

    // inline initialization of array type
    arr2 := {:[10]s32: 0 };         // initialize whole array explicitly to 0
    arr3 := {:[4]s32: 1, 2, 3, 4 }; // initialize array to the sequence 1, 2, 3, 4
};
```

Arrays can be implicitly converted to slice:

```
array_to_slice :: fn () #test {
    arr : [10] s32;
    slice : []s32 = arr;
};
```

String

String type in Biscuit is slice containing pointer to string data and string length. String literals are always zero terminated.

Example:

```
string_type :: fn () #test {
    msg : string = "Hello world\n";
    msg.len; // character count of the string
    msg.ptr; // pointer to the string content
};
```

Slice

Array slice is consist of pointer to the first array element and array length.

Syntax:

```
[] <type>
```

Slice layout:

```
Slice :: struct {
    len: s64;
    ptr: *T
};
```

Example:

```
array_slice :: fn () #test {
    arr :: {:[4]s32: 1, 2, 3, 4};
    slice : []s32 = arr;
    loop i := 0; i < slice.len; i += 1 {
        print("%\n", slice[i]);
    }
};
```

> ***Hint***
>
> :ref:`slice_init` can be used to allocate slice on the heap using context allocator.

Structure

Structure is a composite type representing group of data as a single type. Structure is as an array another way to define user data type, but types of structure members could be different. It can be used in situations when it's better to group data into one unit instead of interact with separate units.

Structure can be declared with use of struct keyword.

```
Person :: struct {
    id: s32;
    name: string;
    age: s32;
}
```

Structure Person in example is consist of id, name and age. Now we can create variable of this type and fill it with data. To access person's member fields use . operator.

```
main :: fn () s32 {
    my_person: Person; // Create instance of type Person
    my_person.id = 1;
    my_person.age = 20;
    my_person.name = "Martin";
```

```
    return 0;
}
```

Inline initialization is also possible. We can use compound expression to set all members at once.

```
main :: fn () s32 {
    my_person1 := {:Person: 0}; // set all data in person to 0
    my_person2 := {:Person: 1, "Martin", 20};

    return 0;
}
```

Structure content can be printed by print function.

```
main :: fn () s32 {
    my_person := {:Person: 1, "Martin", 20};
    print("%\n", my_person);

    return 0;
}
```

```
Person {id = 1, name = Martin, age = 20}
```

Due to lack of OOP support we cannot declare member functions in structures and there is no class or object concept in the language. Common way to manipulate with data is passing them into the function as an argument.

```
person_add_age :: fn (person: *Person, add: s32) {
    person.age += add;
}
```

Structure can extend any type with use of `#base <T>`. This is kind of inheritance similar to C style where inheritance can be simulated by composition. The `#base <T>` basically insert `base: T;` as the first member into the structure. The compiler can use this information later to provide more inheritance related features like merging of scopes to enable direct access to base-type members via `.` operator or implicit cast from child to parent type.

Example of struct extension:

```
Entity :: struct {
    id: s32
}

// Player has base type Entity
Player :: struct #base Entity {
    // base: Entity; is implicitly inserted as first member
    name: string
};

Wall :: struct #base Entity {
    height: s32
};
```

```
Enemy :: struct #base Entity {
    health: s32
};

// Multi-level extension Boss -> Enemy -> Entity
Boss :: struct #base Enemy {
    // Extended struct can be empty.
};

struct_extending :: fn () #test {
    p: Player;
    p.id = 10; // direct access to base-type members
    p.name = "Travis";
    assert(p.base.id == 10); // access via .base

    w: Wall;
    w.id = 11;
    w.height = 666;

    e: Enemy;
    e.id = 12;
    e.health = 100;

    b: Boss;
    b.id = 13;

    // implicit down cast to entity
    update(&p);
    update(&w);
    update(&e);
    update(&b);
}

update :: fn (e: *Entity) {
    print("id = %\n", e.id);
}
```

Union

Union is special composite type representing value of multiple types. Union size is always equal to size of the biggest member type and memory offset of all members is same. Union is usually associated with some enum providing information about stored type.

Example:

```
Token :: union {
    as_string: string;
    as_int: s32;
}

Kind :: enum {
    String;
    Int;
}
```

```
test_union :: fn () #test {
    token1: Token;
    token2: Token;

    // Token has total size of the biggest member.
    assert(sizeof(token1 == sizeof(string));

    token1.as_string = "This is string";
    consumer(&token, Kind.String);

    token2.as_int = 666;
    consumer(&token, Kind.Int);
}

consumer :: fn (token: *Token, kind: TokenKind) {
    switch kind {
        Kind.String { print("%\n", token.as_string); }
        Kind.Int    { print("%\n", token.as_int); }
        default { panic(); }
    }
}
```

Any

The Any type is special builtin structure containing pointer to TypeInfo and pointer to data. Any value can be implicitly casted to this type on function call.

Any type layout:

```
Any :: struct #compiler {
    type_info: *TypeInfo;
    data: *u8
};
```

Remember that the Any instance does not contains copy of the value but only pointer to already stack or heap allocated data. The Any instance never owns pointed data and should not be responsible for memory free.

Since Any contains pointer to data, we need to generate temporary storage on stack for constant literals converted to Any.

```
...
foo(10); // temp for '10' is created here
...

foo :: fn (v: Any) {}
```

For types converted to the Any compiler implicitly sets `type_info` field to pointer to the TypeType type-info and data field to the pointer to actual type-info of the converted type.

```
...
foo(s32); // Type passed
...

foo :: fn (v: Any) {
```

```
    assert(v.type_info.kind == TypeKind.Type);

    data_info := cast(*TypeInfo) v.data;
    assert(data_info.kind == TypeKind.Int);
}
```

Any can be combined with vargs, good example of this use case is print function where args argument type is vargs of Any (... is same as ...Any). The print function can take values of any type passed in args.

```
print :: fn (format: string, args: ...) {
    ...
};
```

Enum

The enum allows the creation of type representing one of listed variants. Biscuit enums can represent variants of any integer type (s32 by default). All variants are grouped into enum's namespace.

Example:

```
// Enum declaration (base type is by default s32)
Color : type : enum {
    Red;    // default value 0
    Green;  // default value 1
    Blue    // default value 2
};

simple_enumerator :: fn () #test {
    assert(cast(s32) Color.Red == 0);
    assert(cast(s32) Color.Green == 1);
    assert(cast(s32) Color.Blue == 2);

    // Base type is s32
    assert(sizeof(Color) == 4);

    // Declare variable of type Color with value Red
    color := Color.Red;
    assert(cast(s32) color == 0);
};

// Enum declaration (base type is u8)
Day :: enum u8 {
    Sat :: 1; // first value explicitly set to 1
    Sun;      // implicitly set to previous value + 1 -> 2
    Mon;      // 3
    Tue;      // ...
    Wed;
    Thu;
    Fri
};

test_enumerator :: fn () #test {
    /* Day */
    assert(cast(s32) Day.Sat == 1);
    assert(cast(s32) Day.Sun == 2);
```

```
    assert(cast(s32) Day.Mon == 3);

    // Base type is u8
    assert(sizeof(Day) == 1);
};
```

Type aliasing

It's possible to create alias to any data type except function types, those can be referenced only by pointers.

```
<alias name> :: <type>;
```

Example:

```
alias :: fn () #test {
    T :: s32;
    i : T;
    i = 10;
    print("%\n", i);
};
```

Function type

Type of function.

```
fn ([arguments]) [T|(T1, T2)]
```

```
// type of function without arguments and without return value
fn ()

// type of function without arguments, returning value of 's32' type
fn () s32

// type of function with two arguments, returning value of 's32' type
fn (s32, bool) s32
```

Type casting

Change type of value to the other type. Conventions between integer types, from pointer to bool and from array to slice are generated implicitly by the compiler.

```
cast(<T>) <expr>
```

Example:

```
type_cast :: fn () #test {
    // default type of integer literal is 's32'
    i := 666;

    // type of the integer literal is changed to u64
    j : u16 = 666;
```

```
    // implicit cast on function call
    fn (num: u64) {
    } (j);

    // explicit cast of 'f32' type to 's32'
    l := 1.5f;
    m := cast(s32) l;
};
```

Biscuit type casting rules are more strict compared to C or C++, there are no void pointers or implicit conversion between integers and enums etc. Despite this fact an explicit cast can be in some cases replaced by auto cast. The auto cast operator does not need explicit destination type notation, it will automatically detect destination type based on expression if possible. When auto operator cannot detect type, it will keep expression's type untouched. In such case auto does not generate any instructions into IR.

```
auto <expr>
```

Example:

```
type_auto_cast :: fn () #test {
    s32_ptr : *s32;
    u32_ptr : *u32;

    // auto cast from *u32 to *s32
    s32_ptr = auto u32_ptr;

    // keep expession type s32
    i := auto 10;
};
```

Literals

Simple literals

```
b :: true;         // bool true literal
b :: false;        // bool false literal
ptr : *s32 = null; // *s32 null pointer literal
```

Integer literals

Biscuit language provides constant integer literals written in various formats showed in example section. Integer literals has volatile type, when desired type is not specified compiler will choose best type to hold the value. Numbers requiring less space than 32 bits will be implicitly set to s32, numbers requiring more space than 31 bits and less space than 64 bits will be set to s64 and numbers requiring 64 bits will be set to u64 type. Bigger numbers are not supported and compiler will complain. When we specify type explicitly (ex.: foo : u8 : 10;), integer literal will inherit that type.

Example:

```
i     :: 10;      // s32 literal
i_u8  : u8 : 10;  // u8 literal
i_hex :: 0x10;    // s32 literal
i_bin :: 0b1011;  // s32 literal
f     :: 13.43f;  // f32 literal
d     :: 13.43;   // f64 literal
char  :: 'i';     // u8 literal
```

Operators

Binary

Symbol	Relevant for types	Description
+	Integers, Floats	Addition.
-	Integers, Floats	Subtraction.
*	Integers, Floats	Multiplication.
/	Integers, Floats	Division.
%	Integers, Floats	Remainder division.
+=	Integers, Floats	Addition and assign.
-=	Integers, Floats	Subtraction and assign.
*=	Integers, Floats	Multiplication and assign.
/=	Integers, Floats	Division and assign.
%=	Integers, Floats	Remainder division and assign.
<	Integers, Floats	Less.
>	Integers, Floats	Greater.
<=	Integers, Floats	Less or equals.
>=	Integers, Floats	Greater or equals.
==	Integers, Floats, Booleans	Equals.
&&	Booleans	Logical AND
||	Booleans	Logical
<<	Integers, Floats	Bitshift left.
>>	Integers, Floats	Bitshift right.

Usage:

```
<expr> <op> <expr>
```

Unary

Symbol	Relevant for types	Description
+	Integers, Floats	Positive value.
-	Integers, Floats	Negative value.

^	Pointers Pointer	Pointer dereference.
&	Allocated value	Address of.

Usage:

```
<op> <expr>
```

Special

Symbol	Relevant for types	Description
sizeof	Any	Determinates size in bytes.
alignof	Any	Determinates alignment of type.
typeinfo	Any	Determinates TypeInfo of type.
typekind	Any	Determinates TypeKind of type.

Type Info

Biscuit language provides type reflection allowing access to the type structure of the code. Pointer to the type information structure can be yielded by typeinfo(:raw-html-m2r:`<T>`) builtin operator call. Type information can be yielded in compile time and also in runtime, with low additional overhead for runtime (only pointer to the TypeInfo constant is pushed on the stack).

Example:

```
RTTI :: fn () #test {
    // yields pointer to TypeInfo constant structure
    info := typeinfo(s32);

    if info.kind == TypeKind.Int {
        // safe cast to *TypeInfoInt
        info_int := cast(*TypeInfoInt) info;

        print("bit_count = %\n", info_int.bit_count);

        if info_int.is_signed {
            print("signed\n");
        } else {
            print("unsigned\n");
        }
    }
};
```

By calling the `typeinfo` operator compiler will automatically include desired type information into output binary.

Hash directive

Hash directives specify special compile-time information used by compiler. They are introduced by # character followed by directive name and optionally some other information.

#load

Load source file into the current assembly. Every file is included into the assembly only once even if we load it from multiple locations.

Lookup order:

- Current file parent directory.
- BL API directory set in install location/etc/bl.conf.
- System PATH environment variable.

```
#load "<bl file>"
```

#import

Import module into current assembly.

```
#import "<bl module>"
```

#private

Creates private (file scope) block in the file. Everything after this is going to be private and visible only inside the current file.

Example:

```
// main is public
main :: fn () s32 {
    foo(); // can be called only inside this file.
    return 0;
};

#private

// private function can be called only inside this file
foo :: fn () {
};

// private constant
bar :: 10;
```

#scope

Creates new named scope i.e. `#scope String`. Every symbol written after the *scope* tag lives in named scope (aka namespace). This prevents possible symbol collisions and makes local names shorter. Named scope cannot be nested in another one and can be specified only once per file unit. Scopes with the same name defined in multiple units are merged into one.

To refer to public symbols from the outside of the named scope use the scope name followed by the dot operator. (i.e. `String.compare`)

#extern

Used for marking entities as an external (imported from dynamic library). Custom linkage name can be specified since version 0.5.2 as a string `#extern "malloc"`, when linkage name is not explicitly specified compiler will use name of the entity as linkage name.

Example:

```
// libc functions
malloc :: fn (size: usize) *u8 #extern;
// since 0.5.2
my_free :: fn (ptr: *u8) #extern "free";
```

#export

Mark symbol to be exported when compile into library. This can be used only for functions for now.

Example:

```
// libc functions
my_func :: fn () #export {
    print("Hello!\n");
}
```

#compiler

Used for marking entities as an compiler internals.

> ***Warning***
>
> This directive is compiler internal.

#test

Introduce test case function. The test case function is supposed not to take any arguments and return always *void*. All function with *test* hash directive are automatically stored into builtin implicit array and can be acquired by *testcases()* function call. Every test case is stored as :ref:`TestCase` type.

Example:

```
this_is_my_test :: fn () #test {
   ...
}
```

#line

Fetch current line in source code as s32.

#file

Fetch current source file name string.

#noinit

Disable variable default initialization. This directive cannot be used with global variables (those must be initialized every time).

Example:

```
test_no_init :: fn () #test {
    my_large_array: [1024]u8 #noinit;
}
```

#call_location

This directive yields pointer to static :ref:`CodeLocation` structure generated by compiler containing call-side location in code. The `call_location` can be used only as function argument default value. It's useful in cases we want to know from where function was called.

Example:

```
test_call_location :: fn () #test {
    print_location();
}

print_location :: fn (loc := #call_location) {
    print("%\n", loc);
}
```

#inline and #no_inline

Function related directives giving the compiler information about possibility of inlining marked function during optimization pass.

Example:

```
my_inline_function :: fn () #inline {
   ...
}
```

#base

Specify base type of structure.

Example:

```
Type :: struct #base s32 {
   ...
}
```

#entry

Specify executable entry function.

> ***Warning***
>
> This directive is compiler internal.

#build_entry

Specify build system entry function.

#tags

Specify struct member tags. This value can be evaluated by type info.

Example:

```
NO_SERIALIZE :: 1;

Type :: struct {
    i: s32 #tags NO_SERIALIZE;
}
```

#intrinsic

Mark external function as compiler specific intrinsic function.

> ***Warning***
>
> This directive is compiler internal.

Variable

Variable associate name with value of some type. Variables in BL can be declared as mutable or immutable, value of immutable variable cannot be changed and can be set only by variable initializer. Type of variable is optional when value is specified. Variables can be declared in local or global scope, local variable lives only in particular function during function execution, global variables lives during whole execution.

Variables without explicit initialization value are `zero initialized` (set to default value). We can suppress this behaviour by `#noinit` directive. Global variables must be initialized every time (explicitly or zero initialized) so `#noinit` cannot be used.

Example:

```
mutable_variables :: fn () #test {
    i : s32 = 666;
    j := 666; // type is optional here
    i = 0; // value can be changed
};

immutable_variables :: fn () #test {
    i : s32 : 666;
    j :: 666; // type is optional here
    // value cannot be changed
};

variable_initialization :: fn () #test {
    i: s32; // implicitly initialized to 0
    arr: [1024]u8 #noinit; // not initialized
}
```

> ***Hint***
>
> Prefer immutable variables as possible, immutable value can be effectively optimized by compiler and could be evaluated in compile time in some cases.

Compound expression

Compound expression can be used for inline initialization of variables or directly as value. Implicit temporary variable is created as needed. Zero initializer can be used as short for memset(0) call.

Example:

```
array_compound :: fn () #test {
    // print out all array values
    print_arr :: fn (v: [2]s32) {
        loop i := 0; i < v.len; i += 1 {
            print("v[%] = %\n", i, v[i]);
        }
    };

    // create array of 2 elements directly in call
    print_arr({:[2]s32: 10, 20});

    // create zero initialized array
    print_arr({:[2]s32: 0});
};

struct_compound :: fn () #test {
    Foo :: struct {
        i: s32;
        j: s32
    };

    print_strct :: fn (v: Foo) {
        print("v.i = %\n", v.i);
        print("v.j = %\n", v.j);
    };

    // create structure in call
    print_strct({:Foo: 10, 20});

    // create zero initialized structure
    print_strct({:Foo: 0});
};
```

Function

Function is chunk of code representing specific piece of program functionality. Function can be called with call operator `()`, we can provide any number of arguments into function and get return value back on call-side.

Functions can be declared in global or local scope (one function can be nested in other).

Named function

Function associated with name can be later called by this name. In this case we treat function like immutable variable.

Example:

```
// named function
my_function :: fn () {
    print("Hello!!!\n");
};

my_function_with_return_value :: fn () s32 {
    return 10;
};

my_function_with_arguments :: fn (i: s32, j: s32) s32 {
    return i + j;
};

test_fn :: fn () #test {
    // call function by name
    my_function();
    result1 :: my_function_with_return_value();
    result2 :: my_function_with_arguments(10, 20);
}
```

Anonymous function

Functions can be used without explicit name defined and can be directly called.

Example:

```
test_anonymous_function :: fn () #test {
    i := fn (i: s32) s32 {
        return i;
    } (666);
    print("%\n", i);
}
```

Function pointer

Functions can be called via pointer. Call on null pointer will produce error in interpreter.

Example:

```
test_fn_pointers :: fn () #test {
    foo :: fn () {
        print("Hello from foo!!!\n");
    };

    bar :: fn () {
        print("Hello from bar!!!\n");
    };
```

```
    // Grab the pointer of 'foo'
    fn_ptr := &foo;

    // Call via pointer reference.
    fn_ptr();

    fn_ptr = &bar;
    fn_ptr();
};
```

Function with variable argument count

Biscuit supports functions with variable argument count of the same type. VArgs type must be last in function argument list. Compiler internally creates temporary array of all arguments passed in vargs. Inside function body variable argument list acts like regular array slice.

Example:

```
sum :: fn (nums: ...s32) s32 {
    // nums is slice of s32
    result := 0;
    loop i := 0; i < nums.len; i += 1 {
        result += nums[i];
    }

    return result;
};

test_vargs :: fn () #test {
    s := sum(10, 20, 30);
    assert(s == 60);

    s = sum(10, 20);
    assert(s == 30);

    s = sum();
    assert(s == 0);
};
```

Local function

Function can be declared even in local scope of another function. Local-scoped functions does not capture variables from parent scope (scope of the upper_func in example), this leads to some restrictions. You cannot access i variable declared in upper_func from the inner_func.

Example:

```
upper_func :: fn () {
    i := 10; // local for upper_func

    inner_func :: fn () {
        i := 20; // local for inner_func (no capture)
    };
}
```

Default argument value

Function arguments can use default value if value is not provided on call side. Default value must be known in compile time.

Example:

```
foo :: fn (i: s32, j := 10) {}

test_foo :: fn () #test {
    // here we call foo only with one argument so j will
    // use default value 10
    foo(10);
}
```

Explicit function overloading

More functions can be associated with one name with explicit function overloading groups. Call to group of functions is replaced with proper function call during compilation, based on provided arguments.

Example:

```
group :: fn { s32_add; f32_add; }

s32_add :: fn (a: s32, b: s32) s32 {
    return a + b;
}

f32_add :: fn (a: f32, b: f32) f32 {
    return a + b;
}

test_group :: fn () #test {
    i :: group(10, 20);
    j :: group(0.2f, 13.534f);
    print("i = %\n", i);
    print("j = %\n", j);
}
```

Multiple return values

Function in BL can return more than one value, this can be useful i.e. in cases we want to return value and error code. There is no explicit limitation of returned value count. Return value can be also named to make the function interface more readable.

Returned values are implicitly converted to anonymous structure instances with possibility to implicitly unroll results on caller side.

Example of multiple return:

```
foo :: fn () (s32, bool) {
    return 666, true;
}

main :: fn () s32 {
    int1, boolean1 := foo();
```

```
    // no all values must be captured
    int2 := foo();
}
```

Example of multiple return with named values:

```
foo :: fn () (number: s32, boolean: bool) {
    return 666, true;
}

main :: fn () s32 {
    int1, boolean1 := foo();
}
```

Block

Block can limit scope of the variable.

Example:

```
#import "std/test"

blocks :: fn () #test {
    a := 10;

    {
    // this variable lives only in this scope
       i := a;
       assert(i == 10);
    }

    i := 20;
    assert(i == 20);
};
```

if - else

If represents condition statement which can change program flow. If executes following code block only if passed condition is `true`, otherwise skip the block and continue on next statement after block. We can specify `else` block which is executed only if condition is `false`.

Example:

```
test_ifs :: fn () #test {
    b := true;
    if b {
        print("b is true!\n");
    } else {
        print("b is false!\n");
    }
};
```

Loop

Example:

```
simple_loops :: fn () #test {
    count :: 10;
    i := 0;

    loop {
        i += 1;
        if i == count { break; }
    }

    i = 0;
    loop i < count {
        i += 1;
    }

    loop j := 0; j < count; j += 1 {
        // do something amazing here
    }
};
```

Break and continue

Break/continue statements can be used in loops to control execution flow.

Example:

```
break_and_continue :: fn () #test {
    i := 0;
    loop {
        i += 1;
        if i == 10 {
            break;
        } else {
            continue;
        }
    }
};
```

Switch

Switch can compare one numeric value against multiple values and switch execution flow to matching case. The *default* case can be used for all other values we don't explicitly specify case for.

Example:

```
test_switch :: fn () #test {
    i := 1;
    switch i {
        0 { print("Zero!\n"); }
        1 { print("One!\n"); }
        default { print("Other!\n"); }
```

```
    }
}
```

Switch can be also used with enumerators, in such case we have to specify cases for all enumerator variations or specify *default* one.

Example:

```
Color :: enum {
    Red;
    Green;
    Blue;
}

test_switch :: fn () #test {
    c := Color.Blue;
    switch c {
        Color.Red   { print("Red!\n");   }
        Color.Green { print("Green!\n"); }
        Color.Blue  { print("Blue!\n");  }
        // default is not needed here, we covered all variants.
    }
}
```

It's also possible to define one execution block for multiple cases.

Example:

```
Color :: enum {
    Red;
    Green;
    Blue;
}

test_switch :: fn () #test {
    c := Color.Blue;
    switch c {
        Color.Red,
        Color.Green { print("Red or green!\n"); }
        Color.Blue  { print("Blue!\n");  }
    }
}
```

Defer statement

The defer statement can be used for defering execution of some expression. All deferred expressions will be executed at the end of the current scope in reverse order. This is usually useful for calling cleanup functions. When scope is terminated by return all previous defers up the scope tree will be called after evaluation of return value.

Example:

```
test_defer_example :: fn () #test {
    defer print("1\n");
```

```
    {
        defer print("2 ");
        defer print("3 ");
        defer print("4 ");
    } // defer 4, 3, 2

    defer_with_return();

    defer print("5 ");
} // defer 5, 1

defer_with_return :: fn () s32 {
    defer print("6 ");
    defer print("7 ");

    if true {
        defer print("8 ");
        return 1;
    } // defer 8, 7, 6

    defer print("9 "); // never reached
    return 0;
};
```

Output:

```
4 3 2 8 7 6 5 1
```

Main function

The *main* function is mandatory entry function which should be defined in every program. It's basically entry point of your application. Main function must return *s32* execution state, zero in this case indicates successful execution.

Example:

```
main :: fn () s32 {
    // some useful stuff goes here.
    return 0;
}
```

> ***Hint***
>
> Command line arguments are not passed directly as parameter in BL. Use :ref:`command_line_arguments` builtin array.

Modules and import

The module system can be used to split source into chunks (modules) which can be later imported into assembly by *#import* directive. Modules can distinguish between platforms and load different sources on them during the compilation process. Entry files for every platform must be explicitly defined in the *module.conf* file located in the module root directory. Name of the root directory is used as a module name during import.

See :ref:`ModuleImportPolicy` for more information about module import policy.

> ***Note***
>
> Module root directory usually contains all source files, libraries and unit tests realted to the module.

> ***Warning***
>
> The configuration file must be located in the module root folder and named *module.conf*.

Example of the module structure:

```
thread/
  module.conf       - module config
  _thread.win32.bl  - windows implementation
  _thread.posix.bl  - posix implementation
  thread.bl         - interface
  thread.test.bl    - unit tests
```

Example of the module config:

```
WIN32_ENTRY "_thread.win32.bl"
LINUX_ENTRY "_thread.posix.bl"
MACOS_ENTRY "_thread.posix.bl"
```

To import out *thread* module use:

```
#import "path/to/module/thread"
```

List of module config entries

- `VERSION "<N>` - Module version number used during import to distinguis various versions of same module, see also :ref:`ModuleImportPolicy` for more information.

Following entries are platform specific, replace `<platform>` with `WIN32`, `LINUX` or `MACOS`.

- `<platform>_ENTRY "<file path>"` - Interface file path. (mandatory, relative to module root)
- `<platform>_LIB_PATH "<lib path>"` - Library search path.
- `<platform>_LINK "<lib name>"` - Library name to link.
- `<platform>_LINKER_OPT "<opt>"` - Additional linker options.

Unit testing

Biscuit compiler provides unit testing by default.

Create unit test case:

```
#import "std/test"

// function to be tested
add :: fn (a: s32, b: s32) s32 {
    return a + b;
};

this_is_OK :: fn () #test {
    assert(add(10, 20) == 30);
};

this_is_not_OK :: fn () #test {
    assert(add(10, 20) != 30);
};

main :: fn () s32 {
    test_run();
    return 0;
}
```

Run tests:

```
$ blc -rt test.bl
Compiler version: 0.7.0, LLVM: 10
Compile assembly: out [DEBUG]
Target: x86_64-pc-windows-msvc

Testing start in compile time
--------------------------------------------------------------------------------
[ PASS |      ] this_is_OK (0.021000 ms)
assert [test.bl:21]: Assertion failed!
execution reached unreachable code
C:/Develop/bl/lib/bl/api/std/debug.bl:113:5
  112 |     if IS_DEBUG { _os_debug_break(); }
> 113 |     unreachable;
      |     ^^^^^^^^^^^
  114 | };
called from:
C:/Develop/bl/tests/test.bl:21:11
  20 | this_is_not_OK :: fn () #test {
> 21 |     assert(add(10, 20) != 30);
     |            ^
  22 | };
[      | FAIL ] this_is_not_OK (1.630000 ms)

Results:
--------------------------------------------------------------------------------
[      | FAIL ] this_is_not_OK (1.630000 ms)
--------------------------------------------------------------------------------
Executed: 2, passed 50%.
--------------------------------------------------------------------------------
```

Build System

Biscuit has an integrated build system replacing CMake or similar tools. Main advantage is integration of the build system directly into the compiler. All you need is *build.bl* file containing *#build_entry* function. Setup file is nothing more than a simple BL program executed in compile time with some special features enabled. See :ref:`Build_System` for more information.

Script mode

Programs written in Biscuit can easily act like shell scripts on UNIX systems due to support of *shebang* character sequence specified at the first line of the entry file. The *-rs* aka *--run-script* option passed to the compiler reduces all compiler diagnostic output and executes the *main* function in compile time. No output binary is produced in such a case. Folowinfg example can be directly executed in *bash* as it was executable.

```
#!/usr/local/bin/blc -rs

main :: fn () s32 {
    print("Hello!!!\n");
    return 0;
}
```

```
$ chmod +x main.bl
$ ./main.bl
```

All additional arguments after *-rs* option are automaticaly forwarded into the executed script and can be accessed via *command_line_arguments* builtin variable during compile-time execution. First argument (index 0) contains the script name everytime.

Automatic Documentation

Integrated self-ducumentation tool can be used to generate RST files from Biscuit source files automatically. Documentation text can be attached to file by *//!* comment prefix or to declaration by *///* comment prefix. Such comments will be recognised by the compiler and attached to propper declaration or file compilation unit. Use *-docs* compiler flag followed by a list of files you want to generate documentation for. Documentation output will be written to *out* directory into the current working directory.

Use marker `@INCOMPLETE` in documentation comment to mark it as incomplete. Compiler will warn you about symbols with incomplete documentation during generation.

Documentation rules:

- Only files listed in compiler input are used as generation input (no loaded or imported files are included).
- Documentation is generated from AST; only parsing is required, after that compiler exits.
- When *out* directory already exists, compiler will only append new files and override old in case of collision.
- Only global and non-private declarations can be documented.
- Declaration name and declaration itself are included automatically.
- Separate *rst* file is produced for every declaration in sub-directory named after file in which symbol is declared.

Example of documented *print* function:

```
/// Write string to the standart output (stdout). Format string can include
/// format specifiers `%` which are replaced by corresponding argument value
/// passed in `args`. Value-string conversion is done automatically, we can
/// pass values of any type as an arguments, even structures or arrays.
///
/// The `print` function accepts C-like escape sequences as `\n`, `\t`, `\r`, etc.
///
/// Pointers to :ref:`Error` are dereferenced automatically; so the `print` function
/// can print out errors directly.
///
/// Count of printed bytes is returned.
///
/// Example
/// -------
/// .. literalinclude:: /examples/docs/007.bl
///    :language: bl
print :: fn (format: string, args: ...) s32 {
    buf: [PRINT_MAX_LENGTH]u8 #noinit;
    w := _print_impl(buf, format, args);
    __os_write(OS_STDOUT, buf.ptr, auto w);
    return w;
};
```

Execution of *blc -docs print.bl* will produce following output:

```
.. _print:

print
=====
.. code-block:: bl

    print :: fn (format: string, args: ...) s32

Write string to the standart output (stdout). Format string can include
format specifiers `%` which are replaced by corresponding argument value
passed in `args`. Value-string conversion is done automatically, we can
pass values of any type as an arguments, even structures or arrays.

The `print` function accepts C-like escape sequences as `\n`, `\t`, `\r`, etc.

Pointers to :ref:`Error` are dereferenced automatically; so the `print` function
can print out errors directly.

Count of printed bytes is returned.

Example
-------
.. literalinclude:: /examples/docs/007.bl
:language: bl
```

Use BL code from C/C++

Since BL compiler supports compilation into shared library, ABI compatible with C, BL code can be easily called from C/C++ program. Use `#export` to mark function to be exported from library and `--shared` flag to create shared object (`so` on Linux). See example bellow on Linux.

```
// Content of my-lib.bl
my_bl_function :: fn (count: s32) #export {
```

```
    loop i := 0; i < count; i += 1 {
        print("Hello from foo library!!!\n");
    }
}
```

```c
// Content of main.c
void my_bl_function(int count);

int main(int argc, char *argv[]) {
    my_bl_function(100);
    return 0;
}
```

```
$ blc --shared my-lib.bl
$ gcc -L. -o test main.c -lout
$ export LD_LIBRARY_PATH=. && ./test
```

Builtin variables

List of builtin variables set by compiler.

- `IS_DEBUG` Is bool immutable variable set to true when assembly is running in debug mode.
- `IS_COMPTIME_RUN` Is bool immutable variable set to true when assembly is executed in compile time.