BitTwiddler

a language for binary data parsers

Final Report

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

Table of Contents

1. Introduction

BitTwiddler was intended to be a powerful tool for inspecting and parsing binary data. Its main goal was to allow programs that turn binary data into textual representations to be easily written. Writing a compiler for this language, however, turned out to be a major undertaking. "The paper accepts everything", the saying goes, but I found the LLVM framework to not be nearly as forgiving.

In the course of BitTwiddler's implementation many features were dropped, either for their infeasibility or for the time to implement them running out. In the end, I opted for having a set of features that would make BitTwiddler minimally useful while still resembling the original whitepaper.

In this report I'll detail the process of building BitTwiddler's compiler. This document is divided into the following sections:

- **Introduction:** this section.
- **Proposal:** the original proposal, annotated with what was cut and what stayed in the language.
- **Language Tutorial:** a short tutorial on how to use BitTwiddler.
- **Language Manual:** the final Language Reference Manual
- **Project Plan:** how the project was planned and built.
- **Architectural Design**: a description of the compiler's architecture
- **Test Plan:** how the compiler was tested
- **Lessons Learned:** what I took from this project.
- **Code Listing:** a listing of the final version of the compiler's code.

2. Proposal

The proposal here was copied as-is from the original submitted proposal, with notes added to point out what made it to the final version. Refer to Language Manual section for the specification of the final version of the language.

2.1. Motivation

Parsing binary data is tricky, especially in high level languages. Python, for example, makes the programmer deal with the cumbersome `struct` module. The C language makes it somewhat easier to map individual bytes or fixed-size chunks of bytes into structures, as long as the programmer takes care of the alignment carefully. Reading in variable-sized items, however, is more complicated. Parsing self-describing binary data can get ugly fast.

2.2. Description

BitTwiddler's primary goal is to **make it easy to describe and read binary-encoded data** in any format and then **parse it into a textual format** of the programmer's choice. In order to achieve this goal, BitTwiddler was designed to be a data-centric programming language. It's main feature is the **template**: an object with typed fields and embedded code to build its members.

> *Note: Templates ended up not being implemented. It's proposed dynamic nature of having a potentially different set of fields on each implementation proved to be extremely tricky: a whole runtime type system would have to be developed.*
>
> *I considered implementing simpler templates, equivalent to C structs, but I didn't due to not having enough time. They wouldn't be too difficult to implement, it would just be more code.*

2.3. Features

- Concise and descriptive code that reads almost as documentation on the binary data being parsed;
- First class functions and types;
- Strong type checking, with reasonable automatic casts;
- All programs read from the standard input and write their results to the standard output, debug/log/info/error messages are written to the standard error output;
- Automatically reads from standard input into variables with no assigned value;
- Basic integral types with different bit widths.

Note: of these, point 1 was partially not implemented (the language ended up being not as concise as initially planned) and the items in point 2 were not implemented at all.

2.4. Comparison with other languages

Consider a game that stores a character's name and health as follows (read from stdin) and parsers in three different languages that output a JSON object.

0x06	'M'	'a'	'r'	'v'	'i'	'n'	0x42	0x00	0x00	0x00
Character's name							Character's health			

```
# Python

from struct import unpack
from sys import stdin

n = unpack('B', stdin.read(1))[0]
name    =    unpack('%ds'    %    n,
stdin.read(n))[0]
health  =  unpack('I',  stdin.read(4))
[0]

print('{"name":"%s","health":%d}' %
    (name, health))
```

```
// C

#include <stdio.h>        // printf
#include <stdint.h>  // uintXX_t
#include <stdlib.h>       // malloc
#include <unistd.h>       // read

int main() {
    uint8_t n;
    read(0, (void*)&n, sizeof(n));
    char *name = (char*)malloc(n+1);
    read(0, (void*)name, n);
    name[n] = 0;
    uint32_t health;
    read(0, (void*)&health, sizeof(health));
    printf("{\"name\":\"%s\",\"health\":%u}\n",
        name, health);
    free(name);
    return 0;
}
```

```
# BitTwiddler

parse {                            # Reads from stdin automatically.
    n:uint8;                       # Declaring without assignment: reads from stdin.
    name:uint8[n];                 # Array declared in terms of previous fields.
    health:uint32;                 # Defaults to native byte order.

    emit('{');                     # emit writes to stdout.
    emit('"name": "{name}",');     # Automatic formatting from uint8[] to string.
    emit('"health": {health}');    # And from uint32 to string.
    emit('}');
}
```

Note: a program like this proposed one is impossible to write in BitTwiddler's final version: there's no automatic conversion from uint8[] to string. It would be easy to implement, but wasn't due to lack of time. Also, strings must be double-quoted delimited only in the final version. The rest, however, made it to the final version.

2.5. Data Types

Type	Description

`{u}int8{le,be}`
`{u}int16{le,be}`
`{u}int32{le,be}`
`{u}int64{le,be}`
	Integer types. Unsigned if prefixed by **`u`**, signed otherwise. Can be suffixed with **`le`** (little endian) or **`be`** (big endian). If the suffix is not specified, native endianess is assumed.
	
	Note: endianess specifiers were dropped.

`float32, float64`	Floating point numbers, 32- or 64-bit wide.
`bit`	Single bit. *Note: this was dropped.*
`string`	Single or several characters. Example: **`hello: string = "world"`**.
`Type`	A basic type or a template type. *Note: this, and the following explicit types were intended to make these concepts first-class citizens of the language. They were dropped.*

`Array<type>`	Array of elements of type *type*.
`Func<r, a1, a2...>`	Function that takes arguments of types *a1, a2...* and returns a value of type *r*.
`Template`	Base type for all templates.
`None`	Unit type, analog to the () type in OCaml. *Note: this type made it to the final version of the language. It is used mainly to annotate functions that don't return a value.*

2.6. Keywords

Keyword	Description
`parse`	The entry point of a program. Must be present exactly once. *Note: this was changed to "main".*
`template`	Used to declare templates, akin to **`dict`** in Python, but smarter. *Note: this was dropped.*

`_`	Means *self* inside a **`template`**, means *any* in **`match`**. *Note: templates were dropped. This represents the "default" case in a **match** expression.*
`func`	Declare a function.
`return`	Return early from a function. *Note: this is mandatory to be used to return from a function that returns a value.*
`if, else`	Conditional execution.
`for, in`	Iteration over all items of an iterable.
`match`	Pattern matching (similar to Rust's **`match`** operator). *Note: match was simplified to be very similar to C's switch statement: only matches on equality.*
`->`	**`match`** arm.

`:`	Type annotation.
`;`	End of statement.
`@`	Prevent embedding a field into a **`template`**. *Note: dropped.*
`{ }`	Code block delimiter.
`#`	Comment.
`' "`	**`string`** delimiters. *Note: single quote as delimiter was dropped.*

2.7. Operators

Operators	Description
`+ - / * %`	Arithmetic plus, minus, divide, multiply, remainder (numbers).

`+`	Concatenate (strings or arrays).
`<< >> | & ~`	Bitwise shift left, shift right, *or*, *and* and *not*, respectively.
`and or not`	Boolean *and*, *or* and *not*, respectively.
`< <= == >= >`	Number comparison.
`==`	Equality (string).
`=`	Assignment.
`[]`	Access an element of an array or field of a template.
`.`	Access a template field. *Note: dropped.*

2.8. Built-in functions

Function	Description
`emit: Func<None, string>`	Writes to stdout.
`print: Func<None, string>`	Writes to stderr.
`fatal: Func<None, string>`	Writes to stderr and ends the program immediately.
`typeof: Func<Type, type>`	Returns the type of a variable. *Note: dropped.*

`len:`
`Func<uint64, string>`
`Func<uint64, Array<type>>`
`Func<uint64, Template>`
	Returns the length of a variable:
	For strings, the number of characters;
	For arrays, the number of elements;
	For templates, the number of fields;
	Note: dropped for templates only.

`enumerate:` `Func<Array<Array<uint64, type>>, Array<type>>`	Returns an array of two-element arrays: the first element is an index into **v**, the second element is the value at that index. *Note: dropped.*

<pre>map: Func<
  Array<*type2*>,
  Array<*type1*>,
  Function<*type2*, *type1*>></pre>
	Maps elements of an array **a** of type *type* to a function **f** that accepts one argument of type *type.* Returns an array of type *type2*, which is **f**'s return type.
	
	Note: dropped.

<pre>join: Func<string, string, Array<string>></pre>	Concatenate strings in the second argument interspersed with the string in the first arg. *Note: dropped.*

2.9. 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):

02	00	04	'n'	'a'	'm'	'e'	01	02	'x'	'p'	03	'A'	'n'	'n'	64	00	00	00
Two fields	First field type 0 = string name = “name”						Second field type 1 = uint32 name = “xp”				First field value "Ann"				Second field value 100			

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

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

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

parse {                                   # Entry point
       numAttrs: uint8;                   # Reads in the number of attributes
       attrs: AttrDesc[numAttrs];         # Reads in the attribute descriptions
       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');
}
```

Note: since templates were dropped, this program is completely invalid in the final version of the language.

2.10. Example Program: gcd

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

parse {
     a : uint32;     # Read inputs from standard input
     b : uint32;

     r = gcd(a, b);       # Automatic upcast uint32 -> uint64, automatic type for r (uint64)
     emit('gcd({a}, {b}) = {r}\n');
}
```

> *Note: gcd can (and was) implemented in a very similar form to the one presented here.*

3. Language Tutorial

Welcome to BitTwiddler! This tutorial will teach you how to get BitTwiddler's compiler up and running and walk you through all of the language's features.

3.1. Getting started

In order to run all the examples in this tutorial you must first obtain the sources for the compiler and build it. Tagged sources are available at:

https://github.com/brunoseivam/bittwiddler/releases

Download the latest release, uncompress it and build it by running:

```
$ make
```

After **make** is done you should have the **bittwiddler.native** program available. This is a compiler from BitTwiddler to LLVM IR language. A second step is necessary to compile from LLVM IR to a binary executable. For your convenience, both steps are executed at once by the **compile.sh** script, distributed with BitTwiddler.

3.2. Hello World

Any tutorial worth its salt will start with a "Hello, world" program, and this one is no exception. So, start by creating a new file called **hello.bt** and typing this into it:

```
main {
    emit("Hello, world!\n"); # Greet the world
}
```

In order to compile this file, you should use the **compile.sh** script:

```
$ ./compile.sh hello.bt
```

After the compiler is done, you will have a new executable file, hello.exe, and, if you run it, you will see the following output:

```
$ ./hello.exe
Hello, world!
```

These steps will need to be performed on every example from here on, but they will be omitted from this tutorial for conciseness.

Now, getting back to the source code, there are a few notes to be made about BitTwiddler.

```
main {
    emit("Hello, world!\n"); # Greet the world
}
```

Source files typically have the **.bt** extension. Every BitTwiddler program must have a **main** block, and the **main** block must be the *last* element in a source file. All function and global variable declarations must appear before **main**.

Comments start with the character **#** and go until the end of the line. There are no multi-line comments.

A block statement comprised of an expression on a single line (the **emit** function call, in this case) has to be ended by a semicolon.

emit is a function, a built-in function in fact, and here it is being called with the string literal **"Hello, world!\n"**.

BitTwiddler's *raison d'être* is to read in binary data and output (**emit**) a textual representation of said data. Hence, the chief printing function here is called **emit**. Other built-in functions will be explored next.

3.3. Built-ins

There are four built-in functions.

<table>
<tr><th>Name</th><th>Input</th><th>Output</th><th>Description</th></tr>
<tr><td><code>emit</code></td><td rowspan="3">A single <code>string</code> literal.</td><td rowspan="3"><code>None</code></td><td>Prints its argument to the standard output.</td></tr>
<tr><td><code>print</code></td><td>Prints its argument to the standard error.</td></tr>
<tr><td><code>fatal</code></td><td>Prints its argument to the standard error. Exits the program.</td></tr>
<tr><td><code>len</code></td><td>A single array or <code>string</code> argument.</td><td><code>uint64</code></td><td>Returns the length of its arguments:
• Number of elements in array
• Number of characters in string</td></tr>
</table>

Here's an example that uses all four built-ins:

Source code

```
main {
    var x = "12345";
    var l = len(x);

    emit("x has length {l}\n");
    print("We emitted something!\n");
    fail("OH NO!\n");
}
```

Standard output

```
x has length 5
```

Standard error

```
We emitted something!
OH NO!
```

Since the whole point of this language is to transform binary data into text, **`emit`** is the only built-in function that can write to the standard output; **`print`** is used for debugging, and **`fatal`** for aborting the program. That's why the last two write to standard error: so they don't "pollute" the standard output.

Another important thing to notice is that the printing functions do **string interpolation**: if the literal string that was passed in has a variable name delimited by curly brackets, it will be replaced by the variable's value. However, this is done at **compile time**, which is why their argument must be a literal string, and not a string-typed expression.

BitTwiddler is a typed language, but notice how the variable declarations don't mention any types: they were inferred. The next section will talk more about variable declarations and types.

3.4. Variables and types

BitTwiddler implements several types:

- Integer: **`int8`**, **`int16`**, **`int32`**, **`int64`**, **`uint8`**, **`uint16`**, **`uint32`**, **`uint64`**

  Signed and unsigned integral types, with different bit widths

- Floating point: **`float32`**, **`float64`**

  Two floating point types with different bit widths. Both are IEEE 754 floats.

- String: **`string`**
- Boolean: **`bool`**
- Void type: **`None`**

These types work as one would expect them to, no surprises here. The void type it used exclusively for function return type; a variable **cannot** have **`None`** type. Now, speaking of variables, we saw previously one way to declare them. There are three ways to declare local variables:

```
main {
    var x:uint32 = 42;          # x has both a type declaration and an initial
                                # value

    var y = "some string";      # y will have the inferred string type, an
                                # explicit type declaration is not needed
    emit("y=\"{y}\"\n");

    var z:int8;                 # There's no initial value; instead, z's value
                                # will be read automatically from the standard
                                # input once this line is executed
}
```

Notice how variables can be declared **anywhere** in a block.

We'll get back to automatic input reading in a bit. Besides local variables, as shown here, a programmer can also declare global variables, like so:

```
var g:int32[] = [1, 2, 3, 4, 5];

main {
    var first_g = g[0];
    emit("g's first value is {first_g}\n");
}
```

Here, **`g`** is a global variable of type **`int32[]`** (which is an array type), and has a literal array as its value. Global variables are more restricted than local variables: they **cannot** have their value read from the standard input. In fact, they cannot be initialized by most expressions, only by literal values.

3.5. Automatic input reading

As we have seen before, variables that have no initializer will have their value read from the standard input. In order to help with testing this feature, there is a tool shipped with BitTwiddler that helps with binary data generation. For the following code example, let's generate some input data:

```
$ ./gen_bin_data i32 42 i8 1 i8 2 i8 3 > auto_input.in
```

This will generate a file with the following contents:

```
$ hexdump -C auto_input.in
00000000  2a 00 00 00 01 02 03                              |*......|
00000007
```

Now, let's write a program, **`auto_input.bt`**, that is capable of reading this data in:

```
main {
    var the_answer:int32;
```

```
    var one_two_three:int8[3];
    var one = one_two_three[0];
    var two = one_two_three[1];
    var three = one_two_three[2];
    emit("The answer is: {the_answer}\n");
    emit("{one} {two} {three}\n");
}
```

If we compile and run this program, we will see the following:

```
$ ./compile.sh auto_input.bt
$ ./auto_input.exe < auto_input.in
The answer is: 42
1 2 3
```

And that's it. Automatic input works for the following types, with the noted restrictions:

- Integer, integer arrays
- Floating point, floating point arrays
- Strings (input must be a C (a.k.a. NUL-terminated) string).

3.6. A note on memory allocation

BitTwiddler allocates memory in a few different places, depending on the variable kind and type:

- Global variables are statically allocated.
- Arrays, strings and arrays of strings are allocated on the heap; they are freed only when the program exits.
- All other values are allocated on the stack.

3.7. Literals

We used literals a few times already, so it is worth mentioning what they can be. Here are a few examples (refer to the Language Reference Manual for exact specs):

- Integers[1]: `1` `-5` `0x42` `0b101010`
- Floats[1]: `5e5` `1.0e2` `.423e-5`
- Boolean: `true` `false`
- String: `"abc"` `"x\"y\" 'z'"`
- Numeric arrays[1]: `[1, 2, 3]` `[4.0, 5.0, 6.0]`

3.8. Expressions

So far BitTwiddler has already been shown to be able to do some pretty neat things. However, any ***computer*** programming language should be able to, well, ***compute*** values. That's where expressions come in. BitTwiddler has plenty of them:

Description	Operators	Operates on	Example	Expression Type
Binary arithmetic	`+ - * /`	Numeric types (floats, integers)	`1.0 + 1.0`	That of the operands
Unary arithmetic	`+ - ~`		`-7`	
Binary comparison	`< <= == != >= >`	Both operands must have the same type	`1 < 2`	`bool`
Binary integer	`% << >> | &`	Integer types	`2 << 4`	That of the operands
Unary integer	`~`		`~1`	
Binary boolean	`and or`	`bool`	`true or false`	`bool`
Unary boolean	`not`	`bool`	`not true`	`bool`
Array subscript	`[]`	Array, integer	`a[i]`	That of an element of a
Assignment	`=`	All types	`a = b`	That of the operands
String concatenation	`+`	`string`	`"x" + "y"`	`string`

1 Numeric types in literal values have an "abstract" type. Their concrete type is inferred by BitTwiddler wherever possible.

All arithmetic, comparison, integer and array subscript operators have the same meaning as in C. Boolean operators and string concatenation have the same meaning as in Python. Refer to the language reference manual for more details.

Granted, these expressions are boring, almost every language has them. The next ones, however, are more interesting.

3.9. Conditional expressions

Conditional execution can be written in BitTwiddler similar to how one would write them in, say, Python:

```
main {
    var x:int8 = 0;
    var y = "";

    if x == 0 {
        y = "x is zero";
    } elif x == 1 {
        y = "x is one";
    } else {
        y = "x is not zero nor one";
    };

    emit("{y}\n");
}
```

However, **`if`** is not a statement; it is an expression (that's why it has to have that pesky semicolon at the end). So, it is perfectly valid to write the following instead:

```
main {
    var x:int8 = 0;

    var y = if x == 0 {
        "x is zero";
    } elif x == 1 {
        "x is one";
    } else {
        "x is not zero nor one"
    };

    emit("{y}\n");
}
```

Both these programs will have the exact same output. Notice how the whole **`if`** expression has type **`string`**, which was inferred. Now, for the last case, BitTwiddler provides a syntactic sugar:

```
main {
    var x:int8 = 0;

    var y = match x {
        0 -> { "x is zero"; }
        1 -> { "x is one";  }
        _ -> { "x is not zero nor one"; }
    };

    emit("{y}\n");
}
```

This **match** expression is semantically equivalent to the previous **if** expression. In fact, under the hood, this **match** expression is transformed into the previous conditional! It is just a more succinct way of writing this kind of pattern.

3.10. Other expressions

There are two kinds of expressions that we used but didn't mention before: identifiers and function calls.

Identifiers are used like so:

```
main {
    var x:int8 = 0;
    var y:int8 = x;
}
```

The third line's initializer expression has a lone **x**, which is an identifier (declared on the previous line). Its type is, as one would expect, the type that was declared in the variable declaration.

Function calls are similar:

```
main {
    var x = "abc";
    var y = len(x);
}
```

They are composed of an identifier (in this case, **len**) and parameters (in this case, a lone **x**). Their type is the return type of the called function.

3.11. Statements

A block of code has statements, which can be of five kinds. Here, we'll describe three.

```
1 + 1;        # A lone expression
return 9.0;   # An expression preceeded by the 'return' keyword
var x = "x";  # A variable declaration
```

Lone expressions are useful for **`if`** and **`match`** blocks: they are the value of that block.

Return expressions are similar, but used exclusively inside functions. They are used for (you guessed it!) returning a value from a function.

Finally, a variable declaration binds a value to a name; we've seen them before.

3.12. Loop statements

The remaining two kinds of statements are loop statements: **`while`** and **`for`**. A **`while`** statement has the form:

```
main {
    var x:int64 = 1;
    while x <= 3 {
        emit("{x} ");
        x = x + 1;
    }
    emit("\n");
}
```

Again, this is pretty straightforward. The **`while`** block executes repeatedly until its predicate evaluates to **`false`**. In the example, **`1 2 3`** will be printed.

BitTwiddler, however, has another kind of loop statement, one that iterates over the indexes and values of an array: it is the **`for..in`** statement. It is used like so:

```
main {
    var x:int8 = [1, 2, 3];
    for idx, val in x {
        emit("{val} ");
    }
    emit("\n");
}
```

The output of this program will be exactly the same. However, this is a more syntactically powerful construct: the names **idx** and **val** (**for..in** requires two names to be given to it) will have types **uint64** and **int8** (the type of an **x** element), respectively, and their scope is the **for..in** block scope. **idx** will hold the index (0-based, of course) into **x**, and **val** will hold the value of the element at index **idx** in **x**.

3.13. Functions

Sometimes we want to separate code used in more than one place into a function. Sometimes we want a function that can call itself to be able to implement succinct algorithms. Fortunately, BitTwiddler's functions allow you to do just that!

Remember, functions have to be defined before the main block. Since "a picture is worth a thousand words", here's an example (which is not, admittedly, a picture, but you get the picture):

```
func sum (a:int32, b:int32):int32 {
    return a + b;
}

func double (x:int32):int32 {
    return sum(x, x);
}

main {
    var double_four = double(4);
    emit("{double_four}\n");      # Will print 8
}
```

A function is defined by having the keyword **func**, followed by the function's name, followed by a parenthesis-delimited list of arguments, followed by the function's return type. If the function doesn't return anything, the return type should be **None**. If the function doesn't take any parameters in, its list of arguments should be **()**.

3.14. A complete example

A good demonstration of BitTwiddler's features is the good old GCD function.

```
func gcd(a:uint64, b:uint64):uint64  {
    return match b {
        0 -> { a;             }
        _ -> { gcd(b, a % b); }
    };
}

main {
    var a : uint64;
    var b : uint64;
    var r1 = gcd(a, b);
    var r2 = gcd(b, a);
    emit("gcd({a}, {b}) = {r1}\n");
    emit("gcd({b}, {a}) = {r2}\n");
}
```

If two 64-bit integers, 10 and 5, are being passed to the following program's standard input, the standard output will be:

```
gcd(10, 5) = 5
gcd(5, 10) = 5
```

3.15. JSON Parsing example

All of this tutorial led to this moment. The moment where we will use BitTwiddler for its main purpose: parsing binary data into a textual representation (in this example, JSON):

```
# Assume that data is laid out in standard input as follows:
# A NUL-terminated string, followed by a 32-bit integer, representing a
# person's name and age.

main {
    var name:string;
    var age:int32;

    emit("{\n");
    emit("  \"{name}\": { \"age\": {age} }\n");
    emit("}\n");
}
```

3.16. Conclusion

And here we finish our BitTwiddler tour. I encourage you to look into BitTwiddler's source code; the tests folder is full of examples of BitTwiddler's functionality.

4. Language Manual

4.1. Lexical Conventions

4.1.1. 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
}
```

4.1.2. Identifiers

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

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

4.1.3. Keywords

The following keywords are reserved by BitTwiddler:

```
int8      uint8    float32     None        main     if       for      and
int16     uint16   float64     true        var      elif     in       or
int32     uint32   string      false       func     else     match    not
int64     uint64   bool                    return   while
```

4.1.4. Literal Constants

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
```

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
```

Strings

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

```
"A double-quoted string"     "Enclosed \"quotes\""
```

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];
```

4.2. 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
    ( expression )
    binary-operation
    unary-operation
    function-call
    conditional
    match
```

4.2.1. Constant

A literal constant expression is composed of any of the literal constants described in the section 4.1.4 Literal Constants above. The type of a literal constant expression is the type of the constant itself.

4.2.2. Identifier

An identifier as described in the section 4.1.2 Identifiers. The type of an identifier expression is the type of the variable associated with that identifier.

4.2.3. Parenthesis-enclosed expressions

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

4.2.4. 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]</td><td>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>

Subscript

```
expression [ expression ]
```

Constraints: The left hand sub-expression must be the id of a variable of array type or `string` type. The sub-expression inside the square brackets must be of integer type.

Type: If the left-hand sub-expression is a `string`, the type of this expression is `int8`. Otherwise, it will be the type of an element of the array.

Unary plus and minus

```
+ expression
- expression
```

Constraints: The sub-expression must be of integer or floating point type.

Type: same as the subexpression.

String concatenation

```
expression + expression
expression - expression
```

Constraints: Both sub-expressions must be of `string` type.

Type: `string`.

Arithmetic operations

```
expression * expression
expression / expression
expression % expression
expression + expression
expression - expression
```

Constraints: Both sub-expressions must have the exact same type. That type must be numeric, i.e., integer or floating point.

Type: same as the type of the sub-expressions.

Bitwise operations

```
~ expression
expression << expression
expression >> expression
expression & expression
expression | expression
```

Constraints: The sub-expressions must be of integer type. For binary operations, both sub-expressions must have the exact same type.

Type: same as the type of the sub-expressions.

Boolean operations

```
not expression
expression < expression
expression <= expression
expression == expression
expression != expression
expression >= expression
expression > expression
expression and expression
expression or expression
```

Constraints: The sub-expressions must be of `bool` type.

Type: `bool`

Assignment

```
expression = expression
```

Constraints: The left-hand sub-expression must be an id. Both sub-expressions must have the same type.

Type: same as the type of the right-hand sub-expression.

4.2.5. 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)
```

The type of a function call expression is the return type of the called function.

4.2.6. 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. The predicate must evaluate to a boolean value. 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";
}
```

A conditional expression block must end with an expression statement. The type of a conditional expression block is the type of the last expression in the block. All blocks must have the same type. The type of the whole conditional expression is the type of its blocks.

4.2.7. 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"; }
    _ -> { "something else"; }
}
```

A match arm block must end with an expression statement. The type of a match arm block is the type of the last expression in the block. All arm blocks must have the same type. The type of the whole match expression is the type of its blocks.

4.3. Blocks and block statements

Blocks are used in a few of BitTwiddler's constructs. They are not expressions; they can only appear in certain specific places. When they are used in if and match expressions, blocks can have a type; the type of a block is the type of its last statement, if the last statement is an expression.

block
    { *block-statements*$_{opt}$ **}**

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

block-statement
    var
    expr **;**
    return *expr* **;**
    for
    while

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

Variables that are declared inside a block have the local scope of the block.

4.3.1. 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 id : type = expression ;        (1)
    var id        = expression ;        (2)
    var id : type              ;        (3)
```

Form (1) assigns the value of the *expression* on the right hand side to the identifier *id*, with the specified *type*.

In form (2) *id* will take the type of the expression on the right hand side.

Now, in form (3), when an expression is not specified, **then the value is automatically read from the standard input[2].**

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

4.3.2. For

The for statement iterates over the items of an iterable expression. It has the following form:

```
for
    for id₁, id₂ in expression block
```

This is somewhat a shorthand for:

```
var <id₁>:uint64 = 0;
var <id₂>:<type of array element> = <empty>;

while <id₁> < len(<expression>) {
    <id₂> = <expression>[<id₁>];
    <block>
    <id₁> = <id₁> + 1;
}
```

Caveat: *id_1* and *id_2* are scoped to the inner block.

Constraints: *expression* must be of type `string` or of type array.

Example:

```
var one_two_three:uint8[] = [1, 2, 3];

for idx, item in one_two_three {
    emit("one_two_three[{idx}] = {item}\n");
}

# Prints:
#   one_two_three[0] = 1
#   one_two_three[1] = 2
#   one_two_three[2] = 3
```

4.3.3. While

The **while** statement 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;
};
```

4.4. 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;
}
```

4.5. Program

A BitTwiddler program has the following structure:

```
program
    decls_opt main block EOF

decls
    decl
    decls decl

decl
    variable
    function
```

Variables and functions declared in *decls* have global scope. The **`main`** block is the entry point of the program.

4.6. Type inference

Wherever possible, BitTwiddler can infer types. For example, in the following snippet:

```
var x = true;
```

The variable `x` has its type inferred to be `bool`. However, this is not always possible. Take, for example:

```
var y = 1;
```

The value `1` is an integer, but BitTwiddler has several integer types; therefore, the compiler will not be able to infer `y`'s type in this case, and compilation will fail. Type inference works even for more complicated expressions:

```
var z = if x { var w:int8 = 1; w; } else { var q:int8; q; };
```

The variable `z` will have `int8` type.

4.7. Standard library functions

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

4.7.1. emit

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

Emits **`val`** to the standard output. **`val`** is parsed at compile time, so it **must** be a literal string.

4.7.2. print

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

Prints **`val`** to standard error. **`val`** is parsed at compile time, so it **must** be a literal string.

4.7.3. fatal

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

Prints **`val`** to standard error; exits from the program. **`val`** is parsed at compile time, so it **must** be a literal string.

4.7.4. 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;

4.8. Example Program: binary data

Consider a hypothetical computer game that stores character attributes in binary files, and the following content for one of these files encoding two characters name and level (numbers are in hexadecimal):

<table>
<tr><td>02</td><td>'B'</td><td>'a'</td><td>'r'</td><td>'r'</td><td>'y'</td><td>00</td><td>0A</td><td>00</td><td>'A'</td><td>'n'</td><td>'n'</td><td>00</td><td>37</td><td>00</td></tr>
<tr><td>Two characters
(uint8)</td><td colspan="6">Name (string)
“Barry”</td><td colspan="2">Level (uint16)
10</td><td colspan="4">Name
"Ann"</td><td colspan="2">Level
55</td></tr>
</table>

```
func read_character():None {
    var name:string;
    var level:uint16;

    emit("  \"{name}\": {\"level\": {level} }\n");
}

main {                                  # Entry point
  var numChars: uint8;                  # Reads in the number of characters in the game

  emit("{\n");
  var i:uint8 = 0;
  while i < numChars {
    read_character();
    i = i + 1;
  }
  emit("}\n");
}
```

4.9. Example Program: gcd

```
func gcd (a:uint64, b:uint64) : uint64 {
    if b == 0 {
        return a;
    } else {
        return gcd(b, a % b);
    };
}

main {
     var a : uint64 = 10;
     var b : uint64 = 5;

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

5. Project Plan

Since this was a one-man project, the planning that went into it was somewhat less strict. I didn't have to coordinate the development with anyone else so I could work on it in my own pace, and in the hours that I have free (nights and weekends) due to my full time job.

Since the beginning I wanted to have a good test suite, so I could iterate rapidly without fear of breaking what I had already done. So for the first part of the project (parser), I built a custom testing script that would run test cases through the parser and make sure that all of them passed. Some of them are still in the tests/parser folder, but they probably don't work anymore since the language changed so much.

Overall, I followed a process that resembles Test-Driven Development: I'd choose one or two features to implement, implement them, write a test for them and then make sure that the previous tests were still passing. This gave me great confidence moving forward.

Choosing which features to implement first generally came down to picking the low-hanging fruit first. In the few weeks before this project is due I realized that I wouldn't have nearly enough time to implement all of the features that I wanted, so I decided to prune what seemed to me like the most difficult spec of the language: templates. On the other hand, I tried my best to make sure that what *got* implemented would be done well (i.e., not haphazardly).

5.1. Programming style guide

For OCaml code, I tried to follow roughly these guidelines:

- 80-char lines
- Spaces, not tabs
- 4-spaces indentation
- Prefer lots of named functions instead of nesting unnamed functions
- Prefer `let _ = ... in` instead of `ignore(...)`
- Put a comment above large logical blocks of code
- Use as few parenthesis as possible (for instance, `Some _` instead of `Some(_)`)
- Put the `in` of a `let ... in` block on its own line, to resemble the aesthetics of a closing curly brace.
- A string concatenation expression that is too long for a single line must continue on the next line, and the `^` character must be on that new line, aligned to where the string expression starts.

For C code (runtime, helpers), I followed loosely the Linux Kernel style guide, using spaces instead of tabs for indentation.

5.2. Software development environment

All tools used were either free or open source:

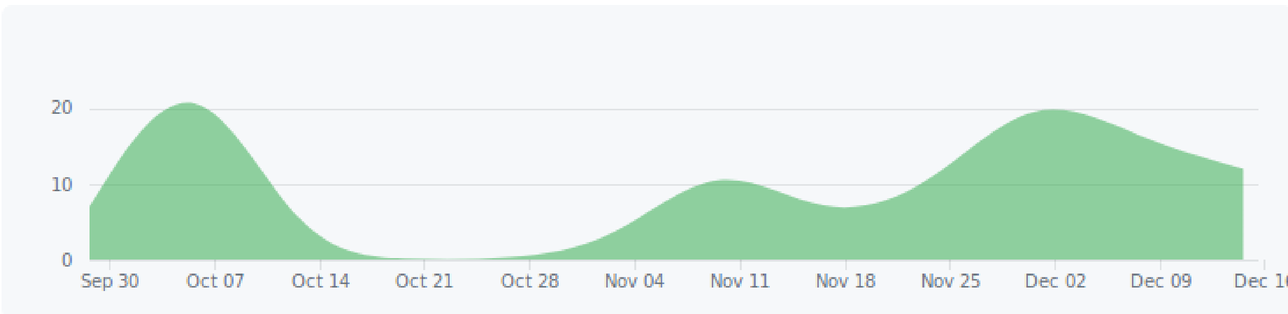
- Ubuntu 16.04: Operating System
- LibreOffice Writer: documentation
- Inkscape: documentation graphics
- OCaml, C, bash: programming languages
- Make: building
- utop: interpreter for testing quick OCaml snippets
- vim: text editor
- git: version control
- GitHub: project hosting
- hexdump: checking of binary data files

5.3. Project log

Sep 30, 2018 – Dec 17, 2018

Contributions: **Commits**

Contributions to master, excluding merge commits



git log --pretty=format:"%ad   %s" --reverse --date=format:"%F %H:%M"

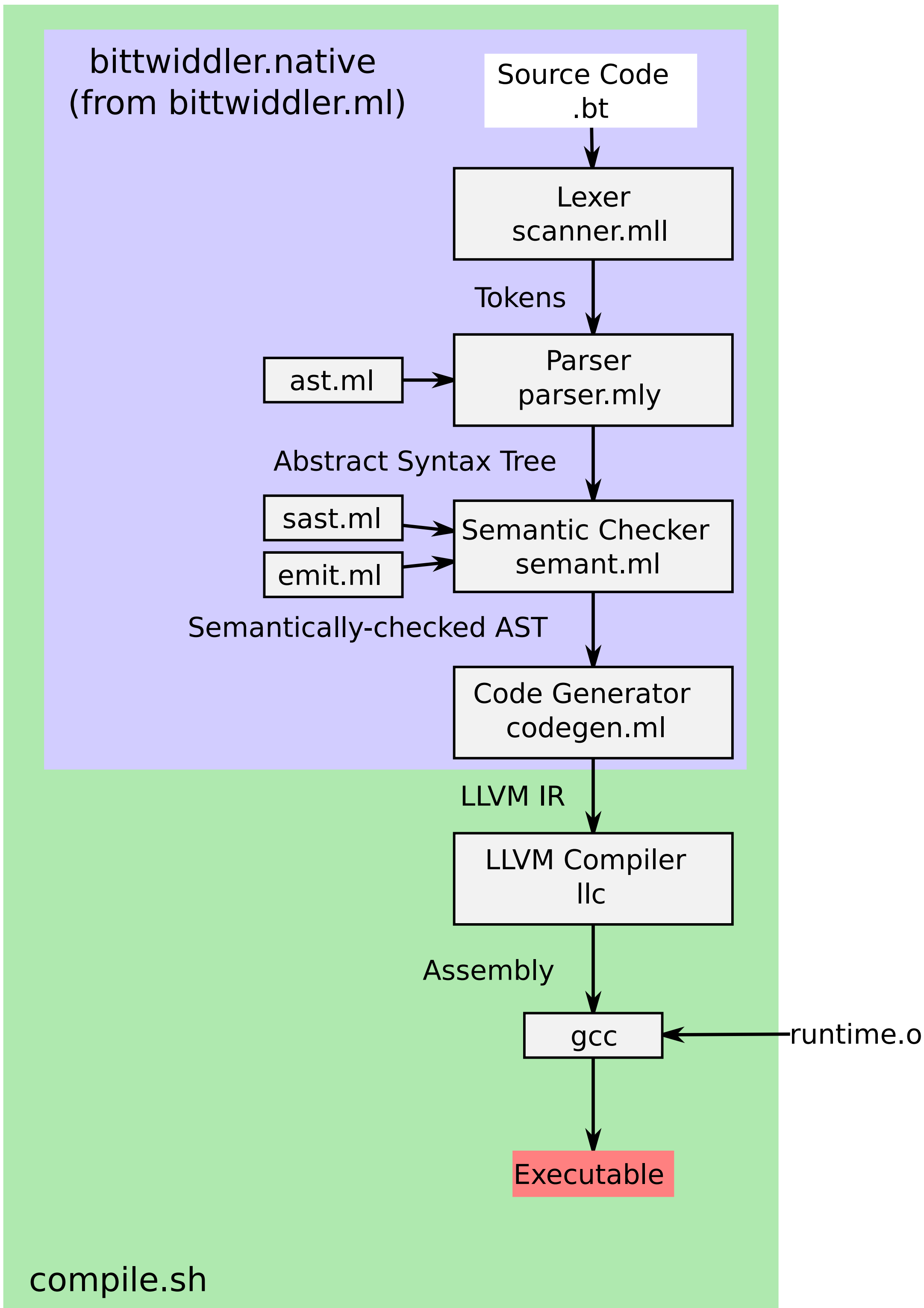
```
2018-10-06 10:06   Initial commit
2018-10-06 10:08   Add proposal documents
2018-10-06 16:23   First version of scanner
2018-10-06 16:26   Add src Makefile
2018-10-06 18:14   Scanner: remove quotes as tokens, add ID
2018-10-06 18:41   Scanner: fixes
2018-10-06 18:43   Add dummy parser, ast and compiler that accept programs
                   that are a single string literal
2018-10-07 10:38   Improve Makefile
2018-10-07 13:00   First functional, partial parser and parser printer
                   (pparser)
2018-10-07 13:00   Add tests for parser
2018-10-07 13:01   Ignore pparser
2018-10-07 13:09   Correctly capture types
2018-10-07 20:02   First attempt at whole grammar (non-working state)
2018-10-09 21:44   Fix most of the grammar
2018-10-09 21:47   Add !=
2018-10-09 21:48   Add and improve tests
2018-10-09 21:58   Fix != and Var
2018-10-09 21:59   Make sure make clean deletes pparser
2018-10-09 22:38   Fix empty block, improve 'compiler'
2018-10-10 20:06   Fix variables
2018-10-10 20:33   Add function call
2018-10-10 20:51   Add rudimentary test runner
2018-10-10 21:08   Add verbose flag to parser executable
2018-10-10 23:57   Several improvements
2018-10-11 00:34   Allow for parameterized templates; change function
```

```
                   definition
2018-10-11 21:08   Fix IF
2018-10-11 22:01   Fix for .. in
2018-10-11 22:05   Add test for IF
2018-10-11 22:34   Add array literal and unary minus, fix integer literals
2018-10-11 22:37   Add log notes
2018-10-13 17:44   Add return, while and assignment expression; change
                   template vars
2018-10-13 17:54   Rename block lines as block statements
2018-10-13 20:07   Simplify program grammar rule
2018-10-13 20:07   Add almost final version of LRM
2018-10-13 20:09   Add dev-tools notes
2018-10-13 20:10   Add log entry
2018-10-14 14:52   Add Language Reference Manual
2018-11-10 16:15   AST: add pretty printing functions
2018-11-10 16:15   Parser: fix type inside returned 'Program'
2018-11-11 16:50   AST: small fixes
2018-11-11 16:54   First stab at end-to-end compilation
2018-11-11 16:56   Reorganize tree pt. 1: rename test/ -> tests/; remove
                   unused Makefiles and scripts
2018-11-11 16:58   Reorganize tree pt. 2: mv src/* to root
2018-11-11 17:06   Add test script and first regression test
2018-11-11 20:24   Fix segfault due to printf; fix main; fix testall script
2018-11-11 20:30   Improve README
2018-11-15 19:20   Simplify ast implementation
2018-11-15 20:29   Simplify and clarify different kinds of blocks
2018-11-15 20:42   Rename the entry point from 'parse' to 'main', because
                   life is too short to figure out how to change the entry
                   point in LLVM
2018-11-15 21:02   testall.sh: test for success and failures again
2018-11-15 21:02   Test for global duplicates
2018-11-15 21:11   Test for hidden global variables
2018-11-15 21:39   Fix type declaration: only Templates can have arguments
2018-11-18 12:04   Add None and abstract Int and Float types
2018-11-18 13:03   Add builtins and checks for bindings
2018-11-23 18:54   AST: add Type to represent types; alias uint8 to boolean
2018-11-23 18:55   First stab at semantically checked expressions
2018-11-23 23:59   First stab at parsing the emit format string
2018-11-25 21:13   Milestone: implement context
2018-11-25 23:32   SAST: Implement pretty-printing
2018-11-25 23:41   Codegen: add some Binops
2018-11-26 20:48   Remove integer endianess from the language
2018-11-28 23:55   Implement code generation for functions, global vars;
                   simplify SAST
2018-11-30 23:03   Improve Makefile
2018-11-30 23:39   Progress with stack variables and function declarations;
                   small fixes
2018-11-30 23:40   Add debug to _tags: whenever OCAMLRUNPARAM=b, a stack
                   trace will be printed on exception
2018-11-30 23:41   Add test
2018-12-02 13:18   Fix precedence: it was backwards...
2018-12-02 13:37   Fix unsignedness being thrown out by the scanner
```

```
2018-12-02 14:57   Fix sast sblock pretty printing
2018-12-02 14:57   Fix indentation
2018-12-02 14:57   Fix codegen of boolean operations
2018-12-02 17:28   Allow escaped characters in string literals
2018-12-02 17:28   Implement unary operations
2018-12-02 18:10   Fix NOT unary operation
2018-12-02 18:59   Fix binary operations
2018-12-02 19:03   Fix builtin emit: treat % character properly, since it'll
                   be passed to printf
2018-12-02 19:04   Add tests for binary and unary operations
2018-12-02 22:54   Implement conditionals in semant
2018-12-04 21:21   Implement 'if' expressions
2018-12-04 23:33   Introduce booleans into the language
2018-12-06 00:18   Fix type coercion; add conditional tests
2018-12-06 21:16   Fix typo that prevented variables from showing up in the
                   local context
2018-12-06 21:39   MILESTONE: gcd works!
2018-12-08 17:31   Add runtime and runtime types: string and array
2018-12-08 17:33   Remove warnings
2018-12-08 18:15   Add testall.log to .gitignore
2018-12-08 19:09   Implement rest of built-in printing functions
2018-12-08 23:36   Implement automatic input reading; a few warts remain
2018-12-08 23:37   Add binary generator utility to Makefile, .gitignore
2018-12-11 15:08   Implement match
2018-12-11 17:16   Have testall test programs which require input
2018-12-11 17:26   Remove the concept of 'hidden var' from the language
2018-12-11 17:46   Implement string concatenation
2018-12-11 18:10   Remove unused SMatch from sast
2018-12-11 19:05   Implement while
2018-12-11 23:54   Simplify array definitions
2018-12-12 18:58   Revert "Simplify array definitions"
2018-12-12 20:11   Simplify runtime design
2018-12-12 22:20   First stab at implementing array operations
2018-12-12 23:43   Implement 'len' builtin
2018-12-13 20:12   Implement while and for loops as statements instead of
                   expressions
2018-12-13 20:15   Rename block_item -> stmt
2018-12-13 22:41   Implement for..in; still incomplete, needs [] operator
2018-12-15 19:42   Implement [] operator; still needs to be properly
                   implemented for strings
2018-12-15 22:25   Fix for..in for strings
2018-12-16 00:56   Implement array reading; introduce semantic-checked types
2018-12-16 01:10   Cleanup: remove templates and types as first class from
                   language
2018-12-16 01:13   Cleanup: id can start with uppercase letter
2018-12-16 15:59   Add new tests; small tweaks to semant.ml
2018-12-16 18:22   Remove dot operator (was intended to be used for templates
2018-12-16 22:58   Strings can only be double-quoted
2018-12-16 23:09   Small fixes to function declaration and generation
2018-12-16 23:24   Don't mix fgetc and read
2018-12-16 23:25   Add test from final report
2018-12-16 23:41   First draft for final report
```

```
2018-12-17 20:05   Change compile.sh to its final version
2018-12-17 22:37   New final report version; almost done
2018-12-18 23:00   Final report
```

6. Architectural Design



As seen in the picture above, the BitTwiddler compilation occurs in two big stages. The first stage executable is obtained by compiling **`bittwiddler.ml`** and its dependencies. The object input to the second stage, **`runtime.o`**, is obtained by compiling **`runtime.c`**. Both these compilations are done by the project's Makefile.

6.1. First stage: LLVM IR generation

The first stage, performed by **`bittwiddler.native`**, takes a source code file written in BitTwiddler and outputs LLVM IR code, These are the steps performed to get there:

1) Lexing, done by **`scanner.ml`**: the source file is read and broken down into tokens, which must respect the lexical rules of the language. If the lexer detects an invalid token, the program is rejected.
2) Parsing, done by **`parser.ml`**, which includes **`ast.ml`**: the tokens from the previous step are taken in and used to assemble the abstract syntax tree, a representation of the language's syntax. If syntax errors are detected at this step, the program is rejected.
3) Semantic checking, done by **`semant.ml`** which includes **`sast.ml`** and **`emit.ml`**: the AST built in the previous step is analyzed for semantic errors and the program's constructs are type-checked. Also, the argument to every **`emit`** built-in call are is and transformed into a format string and a series of arguments for the (future) **`printf`** call to which **`emit`** will be mapped. A new abstract tree, the Semantically-checked AST, is produced.
4) Code generation, done by **`codegen.ml`**: the SAST is taken from the previous step and transformed into LLVM IR language, by using the LLVM library. The LLVM IR code produced is then ready to be taken by the next stage in order to be compiled into machine code.

6.2. Second stage: machine code generation

`compile.sh` performs three steps:

1) Call **`bittwiddler.native`** (described above, first stage) to obtain the LLVM IR.
2) Feed the generated LLVM IR to the LLVM compiler to obtain assembly code.
3) Finally, use **`gcc`** to compile the assembly code and the **`runtime.o`** object together into the final executable.

7. Test plan

For compiler testing, the following tests were used:

Tests expected to pass	Tests expected to fail
`test-001-hello.bt`	`fail-001-dup-global.bt`
`test-002-emit.bt`	`fail-002-dup-function.bt`
`test-003-emit-str.bt`	`fail-003-lit-array-empty.bt`
`test-100-binops-unops.bt`	`fail-004-lit-array-mixed-types.bt`
`test-101-conditionals.bt`	`fail-005-expr-incompatible-types-1.bt`
`test-102-more-conditionals.bt`	`fail-006-expr-incompatible-types-2.bt`
`test-103-match.bt`	`fail-007-expr-incompatible-types-3.bt`
`test-104-while.bt`	`fail-008-expr-op-not-defined.bt`
`test-105-arrays.bt`	`fail-009-undeclared-ident.bt`
`test-106-for.bt`	`fail-010-cant-emit.bt`
`test-107-more-conditionals-2.bt`	`fail-011-non-bool-if-predicate.bt`
`test-200-auto-inputs.bt`	`fail-012-non-bool-while-predicate.bt`
`test-201-more-auto-inputs.bt`	`fail-013-conditional-diff-types.bt`
`test-300-str.bt`	`fail-014-emit-arg.bt`
`test-900-gcd.bt`	`fail-015-len-arg.bt`
`test-901-read-characters.bt`	`fail-016-len-arg-2.bt`
	`fail-017-array-size.bt`
	`fail-018-for-arg.bt`
	`fail-019-for-arg-2.bt`
	`fail-020-illegal-param.bt`

The main goal was to write passing tests that would exercise all implemented features and failing tests that would trigger all possible error paths. In order to automatically run these tests, the **`testall.sh`** script (from MicroC's code) was used, with modifications. The main modification to **`testall.sh`** is that, when executing a passing test, it looks for a file with the **`.in`** suffix and, if this file exists, it feeds this input file to the compiled test's standard input.

The testing script is presented in section 9.14 testall.sh.

7.1. Representative test: gcd

This is an interesting test because it shows many language features concisely: recursive function calls, automatic input reading, type inference (for **r1** and **r2**), the **match** expression and string interpolation for the **emit** call:

```
func gcd (a:uint64, b:uint64):uint64  {
    return match b {
        0 -> { a;            }
        _ -> { gcd(b, a % b); }
    };
}

main {
    var a : uint64;
    var b : uint64;
    var r1 = gcd(a, b);
    var r2 = gcd(b, a);
    emit("gcd({a}, {b}) = {r1}\n");
    emit("gcd({b}, {a}) = {r2}\n");
}
```

Generated LLVM IR:

```
; ModuleID = 'BitTwiddler'

@0 = private unnamed_addr constant [21 x i8] c"gcd(%lu, %lu) = %lu\0A\00"
@1 = private unnamed_addr constant [21 x i8] c"gcd(%lu, %lu) = %lu\0A\00"

declare void @__bt_emit(i32, i8*, ...)

declare i8 @__bt_read_i8()

declare i16 @__bt_read_i16()

declare i32 @__bt_read_i32()

declare i64 @__bt_read_i64()

declare float @__bt_read_f32()

declare double @__bt_read_f64()

declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_read()

declare { i64, i64, i8, i8*, i8* }* @__bt_arr_read(i64, i64)

declare { i64, i64, i8, i8*, i8* }* @__bt_arr_new(i64, i64, i8*)

declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*)
```

```
declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_concat({ i64, { i64, i64, i8,
i8*, i8* }*, i8* }*, { i64, { i64, i64, i8, i8*, i8* }*, i8* }*)

define void @emit() {
entry:
  ret void
}

define i64 @len() {
entry:
  ret i64 0
}

define i64 @gcd(i64, i64) {
entry:
  %blres = alloca i64
  %b = alloca i64
  %a = alloca i64
  store i64 %0, i64* %a
  store i64 %1, i64* %b
  %b1 = load i64, i64* %b
  %tmp = icmp eq i64 %b1, 0
  br i1 %tmp, label %then, label %else

then:                                             ; preds = %entry
  %a2 = load i64, i64* %a
  store i64 %a2, i64* %blres
  br label %merge

else:                                             ; preds = %entry
  %b3 = load i64, i64* %b
  %a4 = load i64, i64* %a
  %b5 = load i64, i64* %b
  %tmp6 = urem i64 %a4, %b5
  %gcd_result = call i64 @gcd(i64 %b3, i64 %tmp6)
  store i64 %gcd_result, i64* %blres
  br label %merge

merge:                                            ; preds = %else, %then
  %res = load i64, i64* %blres
  ret i64 %res
}

define i32 @main() {
entry:
  %r2 = alloca i64
  %r1 = alloca i64
  %b = alloca i64
  %a = alloca i64
  %__bt_read = call i64 @__bt_read_i64()
  store i64 %__bt_read, i64* %a
  %__bt_read1 = call i64 @__bt_read_i64()
  store i64 %__bt_read1, i64* %b
  %a2 = load i64, i64* %a
  %b3 = load i64, i64* %b
  %gcd_result = call i64 @gcd(i64 %a2, i64 %b3)
  store i64 %gcd_result, i64* %r1
  %b4 = load i64, i64* %b
  %a5 = load i64, i64* %a
  %gcd_result6 = call i64 @gcd(i64 %b4, i64 %a5)
  store i64 %gcd_result6, i64* %r2
```

```
  %__bt_str_new = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*
getelementptr inbounds ([21 x i8], [21 x i8]* @0, i32 0, i32 0))
  %arr_ptr = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, { i64,
i64, i8, i8*, i8* }*, i8* }* %__bt_str_new, i32 0, i32 1
  %arr = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr
  %data_ptr = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }*
%arr, i32 0, i32 3
  %data = load i8*, i8** %data_ptr
  %a7 = load i64, i64* %a
  %b8 = load i64, i64* %b
  %r19 = load i64, i64* %r1
  call void (i32, i8*, ...) @__bt_emit(i32 0, i8* %data, i64 %a7, i64 %b8, i64 %r19)
  %__bt_str_new10 = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*
getelementptr inbounds ([21 x i8], [21 x i8]* @1, i32 0, i32 0))
  %arr_ptr11 = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, {
i64, i64, i8, i8*, i8* }*, i8* }* %__bt_str_new10, i32 0, i32 1
  %arr12 = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr11
  %data_ptr13 = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }
* %arr12, i32 0, i32 3
  %data14 = load i8*, i8** %data_ptr13
  %b15 = load i64, i64* %b
  %a16 = load i64, i64* %a
  %r217 = load i64, i64* %r2
  call void (i32, i8*, ...) @__bt_emit(i32 0, i8* %data14, i64 %b15, i64 %a16, i64 %r217)
  ret i32 0
}
```

7.2. Representative test: read character data

This second test was chosen because it does precisely what BitTwiddler was primarily designed for: parsing of binary encoded data into a textual format (in this case, JSON). In particular, this test expect the input to be a 8-bit number N and then N structures each composed of a C-string (name) and a 16-bit integer (level). This binary data represents hypothetical characters in a hypothetical game.

```
# Input data generated with
# ./gen_bin_data u8 2 s "Barry" u16 10 s "Ann" u16 55

func read_character():None {
    var name : string;
    var level : uint16;

    emit("  \"{name}\": {\"level\": {level} }\n");
}

main {
    var num_chars : uint8;

    emit("{\n");
    var i : uint8 = 0;
    while i < num_chars {
        read_character();
        i = i + 1;
    }
    emit("}\n");
}
```

Generated LLVM IR:

```
; ModuleID = 'BitTwiddler'

@0 = private unnamed_addr constant [24 x i8] c"  \22%s\22: {\22level\22: %u }\0A\00"
@1 = private unnamed_addr constant [3 x i8] c"{\0A\00"
@2 = private unnamed_addr constant [3 x i8] c"}\0A\00"

declare void @__bt_emit(i32, i8*, ...)

declare i8 @__bt_read_i8()

declare i16 @__bt_read_i16()

declare i32 @__bt_read_i32()

declare i64 @__bt_read_i64()
```

```
declare float @__bt_read_f32()

declare double @__bt_read_f64()

declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_read()

declare { i64, i64, i8, i8*, i8* }* @__bt_arr_read(i64, i64)

declare { i64, i64, i8, i8*, i8* }* @__bt_arr_new(i64, i64, i8*)

declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*)

declare { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_concat({ i64, { i64, i64, i8, i8*, i8* }*, i8* }*, { i64, { i64, i64, i8, i8*, i8* }*, i8* }*)

define void @emit() {
entry:
  ret void
}

define i64 @len() {
entry:
  ret i64 0
}

define void @read_character() {
entry:
  %level = alloca i16
  %name = alloca { i64, { i64, i64, i8, i8*, i8* }*, i8* }*
  %__bt_read = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_read()
  store { i64, { i64, i64, i8, i8*, i8* }*, i8* }* %__bt_read, { i64, { i64, i64, i8, i8*, i8* }*, i8* }** %name
  %__bt_read1 = call i16 @__bt_read_i16()
  store i16 %__bt_read1, i16* %level
  %__bt_str_new = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @0, i32 0, i32 0))
  %arr_ptr = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, { i64, i64, i8, i8*, i8* }*, i8* }* %__bt_str_new, i32 0, i32 1
  %arr = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr
  %data_ptr = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }* %arr, i32 0, i32 3
  %data = load i8*, i8** %data_ptr
  %name2 = load { i64, { i64, i64, i8, i8*, i8* }*, i8* }*, { i64, { i64, i64, i8, i8*, i8* }*, i8* }** %name
  %arr_ptr3 = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, { i64, i64, i8, i8*, i8* }*, i8* }* %name2, i32 0, i32 1
  %arr4 = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr3
  %data_ptr5 = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }* %arr4, i32 0, i32 3
  %data6 = load i8*, i8** %data_ptr5
  %level7 = load i16, i16* %level
  call void (i32, i8*, ...) @__bt_emit(i32 0, i8* %data, i8* %data6, i16 %level7)
  ret void
}

define i32 @main() {
entry:
  %i = alloca i8
  %num_chars = alloca i8
  %__bt_read = call i8 @__bt_read_i8()
```

```
  store i8 %__bt_read, i8* %num_chars
   %__bt_str_new = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*
getelementptr inbounds ([3 x i8], [3 x i8]* @1, i32 0, i32 0))
  %arr_ptr = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, { i64,
i64, i8, i8*, i8* }*, i8* }* %__bt_str_new, i32 0, i32 1
  %arr = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr
  %data_ptr = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }*
%arr, i32 0, i32 3
  %data = load i8*, i8** %data_ptr
  call void (i32, i8*, ...) @__bt_emit(i32 0, i8* %data)
  store i8 0, i8* %i
  br label %while

while:                                            ; preds = %while_body, %entry
  %i2 = load i8, i8* %i
  %num_chars3 = load i8, i8* %num_chars
  %tmp4 = icmp ult i8 %i2, %num_chars3
  br i1 %tmp4, label %while_body, label %merge

while_body:                                       ; preds = %while
  call void @read_character()
  %i1 = load i8, i8* %i
  %tmp = add i8 %i1, 1
  store i8 %tmp, i8* %i
  br label %while

merge:                                            ; preds = %while
   %__bt_str_new5 = call { i64, { i64, i64, i8, i8*, i8* }*, i8* }* @__bt_str_new(i8*
getelementptr inbounds ([3 x i8], [3 x i8]* @2, i32 0, i32 0))
  %arr_ptr6 = getelementptr inbounds { i64, { i64, i64, i8, i8*, i8* }*, i8* }, { i64, { i64,
i64, i8, i8*, i8* }*, i8* }* %__bt_str_new5, i32 0, i32 1
  %arr7 = load { i64, i64, i8, i8*, i8* }*, { i64, i64, i8, i8*, i8* }** %arr_ptr6
  %data_ptr8 = getelementptr inbounds { i64, i64, i8, i8*, i8* }, { i64, i64, i8, i8*, i8* }*
%arr7, i32 0, i32 3
  %data9 = load i8*, i8** %data_ptr8
  call void (i32, i8*, ...) @__bt_emit(i32 0, i8* %data9)
  ret i32 0
}
```

8. Lessons learned

Compilers are hard.

This class sure made me appreciate all the effort that is required to build a solid programming language. Even seemingly simple concepts, like conditional branches, take a little bit of work to be translated into LLVM.

This class also made me appreciate functional languages. Even though OCaml seems limited at first glance, its expressiveness is incredible. The type system, once you get used to it, forces you to write correct code. One funny thing that I noticed is that every time I ended up writing contrived code to accomplish something, I was doing something wrong.

Lastly, I truly enjoyed the subject matter, Compilers. Akin to Operating Systems, this subject requires the understanding and use of several Computer Science concepts: data structures, computer architecture, data representation, etc. It is satisfying experiencing all of this coming together.

8.1. Advice for future teams

Read the "advice for future teams" from past projects. Really. Print some of them and put them up on a wall so you look at them every day.

I could write here the same thing that most of them mentioned: start early, work on it often, study a lot. Instead, I'll focus on advice for people doing the project by themselves (like me): don't overpromise in the language specification; you'll be doing the work of 4-5 people all by yourself. It is a lot. It'll feel like a breeze when implementing the lexer, perhaps a little bit more difficult when writing the parser. However, the difficulty curve is much steeper when you approach the semantic analyzer and the code generation. It can get overwhelming pretty quickly. Also, work on having a solid testing framework first. It'll make the rest way more manageable.

9. Code listing

9.1. bittwiddler.ml

```
(* Top-level of the BitTwiddler compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module. Based off of MicroC's top-level. *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./bittwiddler.native [-a|-s|-l|-c] [file.bt]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
      | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
sast))
    | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

9.2. scanner.mll

```
{
    open Parser

    let unescape s =
        Scanf.sscanf("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

(* integer unsignedness and width *)
let int_uns  = 'u'?
let int_wid  = '8'|"16"|"32"|"64"

(* float width *)
let float_wid = "32"|"64"

let digit  = ['0'-'9']
let ucase  = ['A'-'Z']
let lcase  = ['a'-'z']
let letter = ['A'-'Z' 'a'-'z' '_']
let id     = (letter | '_')(letter|digit)*

let num  = digit
let dot  = '.'
let sign = ['-' '+']
let exp  = ['e' 'E'] sign? num+

(* float_lit adapted from my homework 2 *)
let float_lit = sign? ((num+ dot? num* exp)|(num* dot num+ exp?)|(num+ dot))
let int_lit = sign? num+
let hex_lit = sign?"0x" ['0'-'9' 'a'-'f' 'A'-'F']+
let bin_lit = sign?"0b" ['0' '1']+

(* String literal parsing copied, with modifications, from the DECAF
 * project (Spring 2017 *)
let ascii_dquote = [' ' -'!' '#'-'[' ']'-'~'] (* ascii without double quote *)
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']

rule token = parse
    (* Whitespace and comments (ignored) *)
    [' ' '\t' '\r' '\n'] { token lexbuf   }
    | '#'                { comment lexbuf }

    (* Delimiters *)
    | '('  { LPAREN } | ')'  { RPAREN }
    | '{'  { LBRACE } | '}'  { RBRACE }
    | '['  { LBRACK } | ']'  { RBRACK }

    (* Keywords *)
    | "main"     { MAIN      }
```

```
    | "func"      { FUNCTION  } | "return" { RETURN }
    | "for"       { FOR       } | "in"     { IN     }
    | "while"     { WHILE     }
    | "match"     { MATCH     } | "->"     { ARM    }
    | "if"        { IF        } | "else"   { ELSE   }
    | "var"       { VAR       } | "elif"   { ELIF   }

    | ',' { COMMA     }
    | ':' { COLON     } | '_'  { WILDCARD }
    | ';' { SEMICOLON } | '='  { ASSIGN   }

    (* Arithmetic *)
    | '+' { PLUS  } | '-' { MINUS }
    | '*' { TIMES } | '/' { DIV   }
    | '%' { REM   }

    (* Bitwise *)
    | "<<" { LSHIFT } | ">>" { RSHIFT }
    | '|'  { BWOR   } | '&'  { BWAND  }
    | '~'  { BWNOT  }

    (* Boolean *)
    | "and" { AND } | "or" { OR } | "not" { NOT }

    (* Comparison *)
    | '<'  { LT } | "<=" { LTEQ } | '>' { GT } | ">=" { GTEQ }
    | "==" { EQ } | "!=" { NEQ  }

    (* Builtin Types *)
    | (int_uns as u) "int" (int_wid as w)
      { INT_T(u="u", int_of_string w) }

    | "float"(float_wid as w) { FLOAT_T(int_of_string w) }
    | "bool"     { BOOL_T   }
    | "string"   { STRING_T }
    | "None"     { NONE_T   }

    (* Literals *)
    | (int_lit | hex_lit | bin_lit) as i    { INT(int_of_string i)     }
    | float_lit as f                        { FLOAT(float_of_string f) }
    | "true"                                { BOOL(true)               }
    | "false"                               { BOOL(false)              }
    | '"' ((ascii_dquote|escape)* as s) '"' { STRING(unescape s) }

    (* Identifier *)
    | id as id  { ID(id) }

    | eof   { EOF }

and comment = parse
     '\n' { token lexbuf }
    | eof { EOF }
```

```
  | _   { comment lexbuf }
```

9.3. parser.mly

```
%{ open Ast %}

%token WILDCARD
%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK LANGLE RANGLE
%token MAIN FUNCTION RETURN VAR
%token FOR IN WHILE MATCH ARM IF ELSE ELIF
%token DOT COMMA COLON SEMICOLON ASSIGN
%token PLUS MINUS TIMES DIV REM
%token LSHIFT RSHIFT BWOR BWAND BWNOT
%token AND OR NOT
%token LT LTEQ EQ NEQ GT GTEQ
%token <string> ID
%token <bool * int> INT_T
%token <int> FLOAT_T
%token STRING_T
%token BOOL_T
%token NONE_T
%token <int> INT
%token <float> FLOAT
%token <string> STRING
%token <bool> BOOL
%token EOF

%right AT
%right COMMA
%right ASSIGN
%left OR
%left BWOR
%left AND
%left BWAND
%left EQ NEQ
%left LT LTEQ GT GTEQ
%left LSHIFT RSHIFT
%left PLUS MINUS
%left TIMES DIV REM
%right NOT BWNOT
%left DOT LBRACK RBRACK

%start program
%type <Ast.program> program

%%

typename:
    INT_T               { TInt $1      }
  | FLOAT_T             { TFloat $1    }
  | STRING_T            { TString      }
  | BOOL_T              { TBool        }
```

```
  | NONE_T              { TNone       }

type_:
    typename                    { ScalarType $1          }
  | typename LBRACK RBRACK      { ArrayType($1, None)    }
  | typename LBRACK expr RBRACK { ArrayType($1, Some $3) }

param:
    ID COLON type_ { Param($1, $3) }

params:
    param               { [$1]     }
  | params COMMA param { $3 :: $1 }

params_opt:
    LPAREN RPAREN        { [] }
  | LPAREN params RPAREN { $2 }

var:
    VAR ID COLON type_ ASSIGN expr SEMICOLON {  Var($2, Some $4, Some $6) }
  | VAR ID COLON type_             SEMICOLON {  Var($2, Some $4, None    ) }
  | VAR ID             ASSIGN expr SEMICOLON {  Var($2, None,    Some $4) }

match_:
    MATCH expr match_block { Match($2, $3) }

match_block:
    LBRACE match_arms RBRACE { List.rev $2 }

match_arms:
    match_arm            { [$1]     }
  | match_arms match_arm { $2 :: $1 }

match_arm:
    expr ARM block     { (Some $1, $3) }
  | WILDCARD ARM block { (None,    $3) }

if_:
    IF expr block { [(Some $2, $3)] }

opt_elseifs:
    /* empty */ { [] }
  | elseifs     { $1 }

elseifs:
    elseif         { [$1]   }
  | elseifs elseif { $2::$1 }

elseif:
    ELIF expr block { (Some $2, $3) }

opt_else:
```

```
    /* empty */     { [] }
  | ELSE block      { [(None, $2)] }

conditional:
    if_ opt_elseifs opt_else { Cond($1 @ (List.rev $2) @ $3) }

expr_list:
    /* empty */          { []     }
  | expr                 { [$1]   }
  | expr_list COMMA expr { $3::$1 }

expr:
    INT    { LInt $1      }
  | FLOAT  { LFloat $1    }
  | STRING { LString $1   }
  | BOOL   { LBool $1     }

  | LBRACK expr_list RBRACK { LArray(List.rev $2) }

  | expr PLUS   expr { Binop($1, Plus,   $3) }
  | expr MINUS  expr { Binop($1, Minus,  $3) }
  | expr TIMES  expr { Binop($1, Times,  $3) }
  | expr DIV    expr { Binop($1, Div,    $3) }
  | expr REM    expr { Binop($1, Rem,    $3) }
  | expr LSHIFT expr { Binop($1, LShift, $3) }
  | expr RSHIFT expr { Binop($1, RShift, $3) }
  | expr BWOR   expr { Binop($1, BwOr,   $3) }
  | expr BWAND  expr { Binop($1, BwAnd,  $3) }
  | expr AND    expr { Binop($1, And,    $3) }
  | expr OR     expr { Binop($1, Or,     $3) }
  | expr LT     expr { Binop($1, Lt,     $3) }
  | expr LTEQ   expr { Binop($1, LtEq,   $3) }
  | expr EQ     expr { Binop($1, Eq,     $3) }
  | expr NEQ    expr { Binop($1, NEq,    $3) }
  | expr GTEQ   expr { Binop($1, GtEq,   $3) }
  | expr GT     expr { Binop($1, Gt,     $3) }
  |      MINUS  expr { Unop(Neg,   $2) }
  |      BWNOT  expr { Unop(BwNot, $2) }
  |      NOT    expr { Unop(Not,   $2) }

  | expr LBRACK expr RBRACK { Binop($1, Subscr, $3) }
  | expr ASSIGN expr        { Binop($1, Assign, $3) }

  | LPAREN expr RPAREN { $2 }
  | match_      { $1 }
  | conditional { $1 }

  | ID       LPAREN expr_list RPAREN { Call($1, List.rev $3)  }

  | ID       { Id $1  }

block_stmt:
```

```
    expr SEMICOLON                 { Expr $1          }
  | var                            { LVar $1          }
  | RETURN expr SEMICOLON          { Return $2        }
  | FOR ID COMMA ID IN expr block { For($2,$4,$6,$7) }
  | WHILE expr block               { While($2,$3)     }

block_stmts:
    block_stmt             { [$1] }
  | block_stmts block_stmt { $2 :: $1 }

block:
    LBRACE block_stmts RBRACE { List.rev $2 }
  | LBRACE RBRACE             { []          }

program_decl:
     FUNCTION ID params_opt COLON type_ block { Func($2, $5, List.rev $3, $6)
}
   |  var                                                        {  GVar($1)
}

program_decls_opt:
    /* empty */   { [] }
  | program_decls { $1 }

program_decls:
    program_decl               { [$1] }
  | program_decls program_decl { $2 :: $1 }

program:
    program_decls_opt MAIN block EOF { Program(List.rev $1, $3) }
```

9.4. ast.ml

```
(* Abstract Syntax Tree *)

type op =
    Plus | Minus | Times | Div | Rem | LShift | RShift
  | BwOr | BwAnd | And | Or | Lt | LtEq | Eq | NEq | GtEq | Gt
  | Subscr | Assign
type uop = BwNot | Not | Neg

type ptype =
  | TInt of (bool * int)
  | TFloat of int
  | TString
  | TBool
  | TAInt (* 'abstract' integer (no size info) *)
  | TAFloat (* 'abstract' float (no size info) *)
  | TNone

and var =
    Var of string * type_ option * expr option

and stmt =
    LVar of var (* local variable declaration *)
  | For of string * string * expr * stmt list
  | While of expr * stmt list
  | Expr of expr
  | Return of expr

and expr =
    LInt of int
  | LFloat of float
  | LString of string
  | LBool of bool
  | LArray of expr list
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Match of expr * (expr option * stmt list) list
  | Cond of (expr option * stmt list) list
  | Call of string * expr list

and type_ =
    ScalarType of ptype
  | ArrayType of ptype * expr option

and param =
    Param of string * type_

type program_decl =
```

```
    Func of string * type_ * param list * stmt list
  | GVar of var (* global variable *)

type program = Program of program_decl list * stmt list

(* Pretty-printing functions *)

let string_of_op = function
    Plus   -> "+"   | Minus  -> "-"   | Times  -> "*"  | Div    -> "/"
  | Rem    -> "%"   | LShift -> "<<"  | RShift -> ">>" | BwOr   -> "|"
  | BwAnd  -> "&"   | And    -> "and" | Or     -> "or" | Lt     -> "<"
  | LtEq   -> "<="  | Eq     -> "=="  | NEq    -> "!=" | GtEq   -> ">="
  | Gt     -> ">"   | Subscr -> "[]"  | Assign -> "="

let string_of_uop = function
    BwNot -> "~" | Not -> "!" | Neg -> "-"

let rec string_of_ptype = function
    TInt(u, w) -> (if u then "uint" else "int") ^ string_of_int w
  | TFloat(w) -> "float" ^ string_of_int w
  | TString -> "string"
  | TBool -> "bool"
  | TAInt -> "int"
  | TAFloat -> "float"
  | TNone -> "None"

and string_of_expr = function
    LInt(i) -> string_of_int i
  | LFloat(f) -> string_of_float f
  | LString(s) -> "\"" ^ s ^ "\""
  | LBool(b) -> string_of_bool b
  | LArray(a) -> "[" ^ String.concat "," (List.map string_of_expr a) ^ "]"
  | Id(id) -> id
  | Binop(e1, Subscr, e2) ->
      string_of_expr e1 ^ "[" ^ string_of_expr e2 ^ "]"
  | Binop(e1, op, e2) ->
        string_of_expr e1
        ^ " " ^ string_of_op op
        ^ " " ^ string_of_expr e2
  | Unop(op, e) -> string_of_uop op ^ string_of_expr e
  | Match(e, arms) ->
        "match " ^ string_of_expr e ^ " {\n"
        ^ string_of_arms arms
        ^ "\n}"
  | Cond(conds) -> string_of_cond conds
  | Call(id, exprs) ->
      id ^ "(" ^ String.concat "," (List.map string_of_expr exprs) ^ ")"

and string_of_block stmts =
    "{\n"
    ^ String.concat "\n" (List.map string_of_stmt stmts)
    ^ "\n}"
```

```
and string_of_stmt = function
    LVar v -> string_of_var v ^ ";"
  | While(e, b) -> "while " ^ string_of_expr e ^ string_of_block b
  | For(id1, id2, e, b) ->
        "for " ^ id1 ^ ", " ^ id2 ^ " in " ^ string_of_expr e ^ " "
        ^ string_of_block b
  | Expr e -> string_of_expr e ^ ";"
  | Return e -> "return " ^ string_of_expr e ^ ";"

and string_of_arm = function
        (Some e, b) -> string_of_expr e ^ " -> " ^ string_of_block b
      | (None, b)   -> "_ -> " ^ string_of_block b

and string_of_arms (arms) =
        String.concat "\n" (List.map string_of_arm arms)

and string_of_cond = function
    (Some e, b)::elses ->
        "if " ^ string_of_expr e ^ " " ^ string_of_block b
        ^ String.concat "" (List.map string_of_else elses)
  | _ -> ""

and string_of_else = function
    (e, b) ->
        (match e with Some e -> "elif " ^ string_of_expr e ^ " "
                    | None -> "else ")
        ^ string_of_block b

and string_of_type = function
    ScalarType ptype -> string_of_ptype ptype
  | ArrayType(ptype, count) ->
        string_of_ptype ptype
        ^ "["
        ^ (match count with Some(e) -> string_of_expr e | None -> "")
        ^ "]"

and string_of_param = function
    Param(id, type_) -> id ^ ":" ^ string_of_type type_

and string_of_params = function
    []     -> ""
   | params -> "(" ^ String.concat "," (List.map string_of_param params) ^ ")
"

and string_of_var = function
    Var(id, type_, expr) ->
        "var "
        ^ id
        ^ (match type_ with Some(t) -> " : " ^ string_of_type t | None -> "")
        ^ (match expr with Some(e) -> " = " ^ string_of_expr e | None -> "")
```

```
let string_of_pdecl = function
    Func(id, type_, params, block) ->
        "func " ^ id ^ " "
        ^ string_of_params params
        ^ ":" ^ string_of_type type_ ^ " "
        ^ string_of_block block
  | GVar(v) ->
        string_of_var v

let string_of_program = function
    Program(pdecls, block) ->
        String.concat "\n" (List.map string_of_pdecl pdecls)
        ^ "\n"
        ^ "main" ^ string_of_block block
```

9.5. sast.ml

```
(* Semantically-checked Abstract Syntax Tree *)

open Ast

type stype =
    SScalar of ptype
  | SArray of ptype * sexpr option

and sexpr = stype * sx

and sx =
    SLInt of int
  | SLFloat of float
  | SLString of string
  | SLBool of bool
  | SLArray of sexpr list
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SIf of sexpr * sstmt list * sstmt list
  | SCall of string * sexpr list

and sstmt =
    SLVar of svar (* local variable declaration *)
  | SExpr of sexpr
  | SReturn of sexpr
  | SFor of svar * svar * sexpr * sstmt list
  | SWhile of sexpr * sstmt list

and svar = string * stype * sexpr option

type sparam = string * stype

type sfunc = string * stype * sparam list * sstmt list

type sprogram_decl =
    SFunc of sfunc
  | SGVar of svar (* global variable *)

type sprogram = sprogram_decl list

let size_t = SScalar (TInt(true,64))
let char_t = SScalar (TInt(false,8))

let is_integer = function
    SScalar(TInt(_,_)) | SScalar TAInt -> true
  | _ -> false
```

```
let is_float = function
    SScalar (TFloat _) | SScalar TAFloat -> true
  | _ -> false

let is_number x = (is_integer x) || (is_float x)

let is_bool = function SScalar TBool -> true | _ -> false

(* Pretty-printing *)

let rec string_of_stype = function
    SScalar ptype -> string_of_ptype ptype
  | SArray (ptype, count) ->
        string_of_ptype ptype
        ^ "["
        ^ (match count with Some(e) -> string_of_sexpr e | None -> "")
        ^ "]"

and string_of_sstmt = function
    SLVar v     -> string_of_svar v ^ ";"
  | SExpr e     -> string_of_sexpr e ^ ";"
  | SReturn e   -> "return " ^ string_of_sexpr e ^ ";"
  | SFor(idx_sv, item_sv, e, b) ->
          "for "
          ^ string_of_svar idx_sv ^ ","
          ^ string_of_svar item_sv ^ " in "
          ^ string_of_sexpr e
          ^ string_of_sblock b
  | SWhile(e, b) ->
          "while " ^ string_of_sexpr e ^ ")" ^ string_of_sblock b

and string_of_sblock b =
    "{\n" ^ (String.concat "\n" (List.map string_of_sstmt b)) ^ "\n}\n"

and string_of_sarm arm =
    let (e, block) = arm in
    (string_of_sexpr e) ^ " -> " ^ (string_of_sblock block)

and string_of_sx = function
    SLInt i -> string_of_int i
  | SLFloat f -> string_of_float f
  | SLString s -> "\"" ^ s ^ "\""
  | SLBool b -> string_of_bool b
  | SLArray lx -> "[" ^ (String.concat "," (List.map string_of_sexpr lx)) ^
"]"
  | SId id -> id
  | SBinop(e1,Subscr,e2) ->
          string_of_sexpr e1 ^ "[" ^ string_of_sexpr e2 ^ "]"
  | SBinop(e1,op,e2) ->
           string_of_sexpr e1 ^ " " ^ string_of_op op ^ " " ^ string_of_sexpr
e2
```

```
  | SUnop(uop,e) ->
        string_of_uop uop ^ " " ^ string_of_sexpr e
  | SIf(pred,then_,else_) ->
      "if " ^ string_of_sexpr pred
      ^ string_of_sblock then_
      ^ "else " ^ string_of_sblock else_
  | SCall(id,el) ->
        id ^ " ("
        ^ (String.concat ", " (List.map string_of_sexpr el)) ^ ")"

and string_of_sexpr (stype,sx) =
    "(" ^ string_of_sx sx ^ "):" ^ string_of_stype stype

and string_of_svar (id, stype, value) =
    "var " ^ id ^ ":" ^ string_of_stype stype ^ " = "
    ^ (match value with Some e -> string_of_sexpr e | None -> "")

let string_of_sparam (id, stype) =
    id ^ ":" ^ string_of_stype stype

let string_of_spdecl = function
    SFunc(id, stype, params, body) ->
      "func " ^ id ^ ":" ^ string_of_stype stype
      ^ " (" ^ (String.concat ", " (List.map string_of_sparam params))
      ^ ")\n" ^ string_of_sblock body
  | SGVar(v) ->
      "SGVar(" ^ (string_of_svar v) ^ ")"

let string_of_sprogram prog =
    String.concat "\n" (List.map string_of_spdecl prog)
    ^ "\n"
```

9.6. semant.ml

```
(* Semantic checking for the BitTwiddler compiler *)

open Ast
open Sast
open Emit

module StringMap = Map.Make(String)

type context = {
    variables : svar StringMap.t;
    functions : sfunc StringMap.t;
}

(* Shorthand *)
let fail s = raise (Failure s)

let string_of_ctx ctx =
    let string_of_sfunc (f:sfunc) = string_of_spdecl (SFunc f) in

    let fold f m = StringMap.fold (fun k v r -> k^":"^(f  v)^"\n"^r) m "" in

    "context\n\n"
    ^ "variables\n---------\n\n" ^ (fold (string_of_svar) ctx.variables)
    ^ "functions\n---------\n\n" ^ (fold (string_of_sfunc) ctx.functions)

(* Helper functions to add elements to maps *)
let add_to_map map id elem kind =
    match id with
        _ when StringMap.mem id map ->
          fail ("duplicate " ^ kind ^ " declaration: " ^ id)
      | _ -> StringMap.add id elem map

let add_var map = function
    (id,_,_) as v -> add_to_map map id v "variable"

let add_func map = function
    (id,_,_,_) as f -> add_to_map map id f "function"

let find_elem map id =
    try StringMap.find id map
    with Not_found -> fail ("undeclared identifier " ^ id)

(* Built-in functions transformations *)
let check_emit_fmt fname ctx emit_fmt =
    let emit_err id t =
        fail ("don't know how to emit " ^ id ^ ":" ^ string_of_stype t)
    in
```

```
    let fkind = (SScalar (TInt(false, 32)), SLInt(match fname with
        "emit" -> 0 | "print" -> 1 | "fatal" -> 2
      | _ -> fail ("invalid 'emit' kind: '" ^ fname ^ "'")))
    in

    (* Builds the printf-ready format string and list of variable ids *)
    let rec build_args fmt args exprs = match exprs with
        [] -> (fmt, args)
      | (STR s)::tl -> build_args (fmt ^ s) args tl
      | (VAR id)::tl -> (match find_elem ctx.variables id with
            (id, SScalar pt,_) ->
                let tfmt = (match pt with
                    TInt(true,64) -> "%lu"
                  | TInt(false,64) -> "%ld"
                  | TInt(true,_) -> "%u"
                  | TAInt | TInt(false,_) -> "%d"
                  | TAFloat | TFloat(_) -> "%f"
                  | TString -> "%s"
                  | TBool -> "%d"
                  | _ -> emit_err id (SScalar pt)
                ) in
                build_args (fmt ^ tfmt) ((SScalar pt, SId id)::args) tl
          | (id,t,_) -> emit_err id t
        )
    in

    (* Note: args is reversed here *)
    let (fmt, args) = build_args "" [] (parse_emit_fmt emit_fmt) in
    ([fkind; (SScalar TString, SLString fmt)] @ (List.rev args))

(* Type compatibility: 'abstract' types are promoted to concrete types *)
(* TODO: upcast for different integer/float sizes *)
let check_type_compat t1 t2 =
    let failure = "Incompatible types " ^ string_of_stype t1
        ^ " and " ^ string_of_stype t2
    in

    let upcast t1 t2 = match (t1,t2) with
        (TInt(_,_), TAInt) -> (t1, t1)
      | (TAInt, TInt(_,_)) -> (t2, t2)
      | (TFloat(_), TAFloat) -> (t1, t1)
      | (TAFloat, TFloat(_)) -> (t2, t2)
      | (TAInt, TAInt) -> (TInt(false,64), TInt(false,64))
      | (TAFloat, TAFloat) -> (TFloat 64, TFloat 64)
      | (_, _) when t1 = t2 -> (t1, t2)
      | _ -> fail failure
    in

    match (t1, t2) with
        (SScalar st1, SScalar st2) ->
            let (st1', st2') = upcast st1 st2 in
                (SScalar st1', SScalar st2')
```

```
      | (SArray(st1,l1), SArray(st2,l2)) ->
            let (st1', st2') = upcast st1 st2 in
                (SArray(st1',l1), SArray(st2',l2))
      | _ -> fail failure

let rec type_of_arr_lit = function
    [] -> fail "Can't determine type of empty literal array"
  | [(SScalar t,_)] -> t
  | (SScalar t,_)::tl when t = type_of_arr_lit tl -> t
  | _ -> fail "Array literal has mixed or invalid types"

(*
 * Checkers that return semantically checked elements
 *)

(* Check expression. Return sexpr. *)
let rec check_expr ctx = function
    LInt l -> (SScalar TAInt, SLInt l)
  | LFloat l -> (SScalar TAFloat, SLFloat l)
  | LString l -> (SScalar TString, SLString l)
  | LBool l -> (SScalar TBool, SLBool l)
  | LArray el ->
        (* sel = semantically-checked expression list *)
        let sel = List.map (check_expr ctx) el in
        (SArray(type_of_arr_lit sel, Some((size_t,SLInt(List.length sel)))),
         SLArray sel)
  | Id s ->
        let (_,type_,_) = find_elem ctx.variables s in (type_, SId s)

    (* A Subscr a[i] will be transformed into:
     *
     *    if i >= 0 and i < len(a) {
     *        a[i];
     *    } else {
     *        fail("array access out of bounds\n");
     *        0;
     *    }
     *
     *)
  | Binop(a, Subscr, i) ->
        let (a_t, a') = check_expr ctx a
        and (i_t, i') = check_expr ctx i in

        let failure =
            "Operator " ^ string_of_op Subscr ^ " not defined for types "
            ^ string_of_stype a_t ^ " and " ^ string_of_stype i_t
        in

        (* Check that the types are compatible *)
        let ty = match (a_t, is_integer i_t) with
            (SArray (t,_), true) -> SScalar t
          | (SScalar TString, true) -> char_t
```

```
          | _ -> fail failure
        in

        let i_t = if i_t=SScalar TAInt then size_t else i_t in

        let pty =
            match ty with SScalar t -> t | _ -> fail "internal error"
        in

        (* Generate bounds-checked array access *)
        let pred = check_expr ctx (Binop(
            Binop(i,GtEq,LInt 0),
            And,
            Binop(i,Lt,Call("len",[a]))
        )) in

        let then_ = [SExpr(ty, SBinop((a_t,a'), Subscr, (i_t,i')))] in

        let else_ = [
            SExpr(check_expr ctx (
                Call("fatal", [LString "array access out of bounds"])
            ));
            SExpr(ty, match pty with
                TInt _   -> SLInt 0
              | TFloat _ -> SLFloat 0.0
              | TString  -> SLString ""
              | _ -> fail ("internal error: don't know what to return"
                        ^ " for out of bounds access in "
                        ^ string_of_expr (Binop(a,Subscr,i)))
            )
        ] in

        (ty, SIf(pred, then_, else_))

  | Binop(e1,op,e2) ->
        let failure t1 t2 =
            "Operator " ^ string_of_op op ^ " not defined for types "
            ^ string_of_stype t1 ^ " and " ^ string_of_stype t2
        in

        let (t1, e1') = check_expr ctx e1
        and (t2, e2') = check_expr ctx e2 in

        (* Promote types and check that they are compatible *)
        let (t1', t2') = check_type_compat t1 t2 in

        let ty = match op with
            (* Overloaded Plus *)
            Plus -> (match (t1', t2') with
                (* String concatenation *)
                (SScalar TString, _) -> t1'
                (* Array concatenation *)
```

```
              | (SArray(st1',Some(_, SLInt l1)),
                 SArray(_,Some(_, SLInt l2))) ->
                    SArray(st1',Some(size_t, SLInt(l1+l2)))
                (* Number addition *)
              | (t1',_) when is_number t1' -> t1'
              | _ -> fail (failure t1' t2'))

          (* Arithmetic: defined for numbers *)
        | Minus | Times | Div when is_number t1' -> t1'
          (* Defined for Integers only *)
        | Rem | LShift | RShift | BwOr | BwAnd when is_integer t1' -> t1'
          (* Boolean operations *)
        | And | Or when is_bool t1' -> SScalar TBool
        | Lt | LtEq | Eq | NEq | GtEq | Gt -> SScalar TBool
        | Assign -> t2'
        | _ -> fail (failure t1' t2')
      in
      (ty, SBinop((t1',e1'), op, (t2',e2')))

  | Unop(uop, e) ->
      let (t, e') = check_expr ctx e in
      let ty = match (t, uop) with
          (SScalar TBool, Not) -> t
        | (SScalar (TInt _), BwNot)
        | (SScalar (TInt _), Neg)
        | (SScalar (TFloat _), Neg) -> t
        | _ -> fail ("Operator " ^ string_of_uop uop
                                ^ " not defined for type "
                                ^ string_of_stype t)
      in
      (ty, SUnop(uop, (t, e')))

    (* Match will be transformed into Cond *)
  | Match(matching, arms) ->
      let to_cond = function
          (Some m, block) -> (Some(Binop(matching, Eq, m)), block)
        | (None, block) -> (None, block)
      in
      check_expr ctx (Cond(List.map to_cond arms))

  | Cond(conds) ->
      (* conds is (expr option * stmt list) list
       *            vvvvvvvvvvv   vvvvvvvvvvvvvvv
       *             condition          block
       *
       * So we iterate over conds and enforce the following constraints:
       *   - All conditions must have boolean type
       *   - All blocks must have the same type
       *
       * Then we do a transformation to make it easy for codegen:
       *
       *   if (pred1) {B1} else if(pred2) {B2} else {B3}
```

```
         *
         * Becomes:
         *
         *   if (pred1) {B1} else { if(pred2) {B2} else {B3} }
         *)

        (* An expression fed to an if or else if clause must be boolean *)
        let check_cond_expr = function
            None -> None
          | Some e ->
                (match (check_expr ctx e) with
                    (SScalar TBool,_) as e' -> Some e'
                  | (_,sx) -> fail ("Non-boolean expression in conditional: "
                                      ^ string_of_sx sx)
                )
        in

        let rec check_conds = function
            [] -> fail "internal error: empty conditional?"
          | [(e, b)] ->
                let (se, (t, sb)) = (check_cond_expr e, check_block ctx b) in
                (t, [(se, sb)])
          | (e, b)::tl ->
                  let (se, (t1, sb)) = (check_cond_expr e, check_block ctx b)
in
                let (t2, r) = check_conds tl in
                if t1=t2 then
                    (t1, (se, sb)::r)
                else
                    fail "conditional blocks have different types"
        in

        (* Transform into simpler nested if/else blocks *)
        let rec simplify_conds t = function
            [(Some se,sb)] -> SIf(se, sb, [])
          | [(Some se1,sb1);(None,sb2)] -> SIf(se1, sb1, sb2)
          | (Some se, sb)::tl -> SIf(se, sb, [SExpr(t, simplify_conds t tl)])
          | _ -> fail "internal error: malformed conditional"
        in

        let (t, sconds) = check_conds conds in
        (t, simplify_conds t sconds)


  (* When 'emit' is called, we have to build its arguments from the format
   * string *)
  | Call(id, el) when id="emit" || id="print" || id="fatal" -> (match el with
        [(LString s)] ->
            (SScalar TNone, SCall("__bt_emit", check_emit_fmt id ctx s))
      | _ -> fail ("'" ^ id ^ "' requires a single literal string argument")
    )
```

```
  (* 'len' is expected to be applied to arrays or strings only *)
  | Call(id, el) when id="len" -> (match el with
        [e] ->
          let (t,e') = check_expr ctx e in
          let fname = match t with
              SScalar TString | SArray _ -> "__bt_len"
            | _ -> fail (id ^ " can't be applied to type: "
                         ^ string_of_stype t)
          in
          (size_t, SCall(fname, [(t,e')]))
      | _ -> fail ("len requires a single array or string argument")
    )

  | Call(id, el) ->
        let (_,type_,_,_) = find_elem ctx.functions id in
        (type_, SCall(id, List.map (check_expr ctx) el))

and check_type ctx = function
    ScalarType t -> SScalar t
  | ArrayType(t, Some e) ->
        let st, se = check_expr ctx e in
        let st = match st with
            SScalar TAInt -> size_t
          | SScalar TInt _ -> st
          | _ -> fail ("array size can't be of type " ^ string_of_stype st)
        in
        SArray(t, Some (st, se))
  | ArrayType(t, None) -> SArray(t, None)

and check_var ctx v =
    let Var(id, t, e) = v in
    let (t, e) = match (t, e) with
        (* Both type and initial value, check that types are compatible *)
        (Some t, Some e) ->
            let t = check_type ctx t in
            let (st, se) = check_expr ctx e in
            let _ = check_type_compat t st in
            (t, (st, se))

        (* Only type was declared, add automatic input reading *)
      | (Some t, None) ->
            let t = check_type ctx t in
            (t, (t, SCall("__bt_read", [])))

        (* Only value was declared, coerce type *)
      | (None, Some e) ->
            let (st, se) = check_expr ctx e in
            (match st with
                SScalar TAInt | SScalar TAFloat ->
                    fail ("can't determine type of variable " ^ id)
              | _ ->
                    (st, (st, se))
```

```
            )
      | (None, None) ->
            fail ("internal error: variable " ^ id
                  ^ " has no type and no value")
    in
    let sv = (id, t, Some e) in
    ({ ctx with variables = add_var ctx.variables sv }, sv)

(* Returns a new context and a semantically checked block item *)
and check_stmt ctx = function
    LVar v -> let (ctx, sv) = check_var ctx v in (ctx, SLVar sv)
  | Expr e -> (ctx, SExpr(check_expr ctx e))
  | Return e -> (ctx, SReturn(check_expr ctx e))
  | While(pred, block) ->
        let pred' = match (check_expr ctx pred) with
            (SScalar TBool, _) as pred' -> pred'
          | (_,sx) -> fail ("Non-boolean expression in while predicate: "
                            ^ (string_of_sx sx))
        in

        let (_, block') = check_block ctx block in
        (ctx, SWhile(pred', block'))

  | For(idx, item, e, block) ->
        let (t, e') = check_expr ctx e in
        let idx_t = size_t in
        let item_t = match t with
            SArray(TString,_) -> fail "Can't iterate over array of strings"
          | SArray(pt,_) -> SScalar pt
          | SScalar TString -> char_t
          | _ -> fail ("Can't iterate over expression of type "
                       ^ string_of_stype t)
        in
        let idx_svar = (idx, idx_t, Some (idx_t, SLInt(0))) in
        let item_svar = (item, item_t, None) in
        let lvars = List.fold_left (
            fun vars sv -> add_var vars sv
        ) ctx.variables [idx_svar; item_svar] in
        let lctx = { ctx with variables = lvars } in
        let (_, block') = check_block lctx block in
        (ctx, SFor(idx_svar, item_svar, (t, e'), block'))


(* Returns the type of the block (type of its last item) and a list of
 * semantically-checked block items *)
and check_block ctx = function
    [] -> (SScalar TNone, [])
  | [item] ->
        let (_, item) = check_stmt ctx item in
        (match item with
            SLVar _ | SReturn _ | SWhile _ | SFor _ -> SScalar TNone
          | SExpr(t, _) -> t)
```

```
        , [item]
  | hd::tl ->
        let (ctx, item) = check_stmt ctx hd in
        let (t, block) = check_block ctx tl in
        (t, item::block)

(* Look for duplicates in a list of names *)
let check_dup kind where names =
    let sorted = List.sort (compare) names in
    let rec dups = function
        [] -> ()
      | (a::b::_) when a = b ->
            fail ("duplicate " ^ kind ^ " in " ^ where ^ ": " ^ a)
      | _ :: t -> dups t
    in dups sorted

(* Check bindings for duplicates and None types.
 * Returns a new context with the parameters as local variables and a list of
 * semantically checked parameters *)
let rec add_params ctx params = match params with
    [] -> ctx
  | Param(id, type_)::tl ->
        let sv = (id, check_type ctx type_, None) in
        let ctx = { ctx with variables = add_var ctx.variables sv } in
        add_params ctx tl

let check_params ctx params where =
    let _ = check_dup "parameter" where
        (List.map (function Param(id,_) -> id) params)
    in
    let _ = List.iter (function
        Param(id,ScalarType TNone) ->
            fail ("illegal " ^ id ^ " : None in " ^ where)
      | Param(id,ArrayType(TNone,_)) ->
            fail ("illegal " ^ id ^ " : None[] in " ^ where)
      | _ -> ()
    ) params in

    (add_params ctx params,
       List.map  (function  Param(id,  type_)  ->  (id,  check_type  ctx  type_))
params)

(* Check global program declarations: Global variable and function
 * Returns a modified context and a semantically checked program declaration.
 *)
let check_pdecl ctx = function
    Func(id, type_, params, body) ->
        let (lctx, sp) = check_params ctx params id in
        let stype = check_type ctx type_ in
        (* Add itself to its local context: allow recursion *)
        let lctx = {
            lctx with functions = add_func ctx.functions (id,stype,sp,[])
```

```
        } in
        let (_, sbody) = check_block lctx body in
        let sf = (id, stype, sp, sbody) in
        ({ ctx with functions = add_func ctx.functions sf }, SFunc sf)
  | GVar(v) ->
        let (ctx, sv) = check_var ctx v in
        (ctx, SGVar(sv))

let rec check_pdecls ctx = function
    [] -> []
  | hd::tl -> let (ctx, d) = check_pdecl ctx hd in d::(check_pdecls ctx tl)

let coerce_func (sf:sfunc) =
    let (id, ftype, sparams, body) = sf in

    let coerce_fail e t ty =
        fail ("can't coerce expression " ^ (string_of_sx e) ^ " from type "
              ^ (string_of_stype t) ^ " to type " ^ (string_of_stype ty))
    in

    let rec coerce_sexpr ty (t,e) = match (ty, t) with
        (t1, t2) when t1=t2 -> (t, e)
      | (SScalar (TInt _), SScalar TAInt)
      | (SScalar (TFloat _), SScalar TAFloat) ->
            (match e with
                SLInt _ | SLFloat _ ->
                    (ty, e)
              | SBinop(se1, op, se2) ->
                    (ty, SBinop (coerce_sexpr ty se1, op,
                                 coerce_sexpr ty se2))
              | SUnop(op, se) ->
                    (ty, SUnop (op, coerce_sexpr ty se))
              | SIf(pred, then_, else_) ->
                    (ty, SIf(pred, coerce_sblock ty then_,
                             coerce_sblock ty else_))
              | _ -> coerce_fail e t ty
            )
      | (SArray(et, _), _) ->
            (match e with
                SLArray el -> (ty,
                    SLArray(List.map (coerce_sexpr (SScalar et)) el)
                )
              | _ -> coerce_fail e t ty
            )
      | _ -> (t, e)

    and coerce_sstmt ty = function
           (* A variable declaration statement has None type. Its value
expression
         * must be coerced to the variable's type. For example, in:
         *
         *     var x: int8 = 5 + 5;
```

```
         *
         * the expression 5 + 5 must have type int8 *)
        SLVar(id, t, Some se) ->
            SLVar(id, t, Some (coerce_sexpr t se))

          (* A return statement has None type. Its returned expression must
have
         * the type that the enclosing function expects *)
      | SReturn se -> SReturn(coerce_sexpr ftype se)

        (* The last SExpr in a block is the block's value, so its type must
         * match the expected type ty. *)
      | SExpr se -> SExpr(coerce_sexpr ty se)

        (* TODO: should never reach this; SLVars should have a value defined
         * after semantic analysis *)
      | _ as i -> i

    and coerce_sblock ty = function
        [] -> []
        (* The last item in a block confers the block its type *)
      | [item] -> [coerce_sstmt ty item]
      | hd::tl ->
            let hd =
               match hd with SExpr _ -> hd | _ -> coerce_sstmt ty hd
            in
            hd::(coerce_sblock ty tl)
    in
    SFunc(id, ftype, sparams, coerce_sblock (SScalar TNone) body)

let rec coerce_sprog = function
    [] -> []
  | hd::tl -> (match hd with
        SFunc(sf) -> coerce_func sf
      | _ -> hd
    )::(coerce_sprog tl)

(* Semantic checking of the AST. Returns an SAST if successful,
 * throws an exception if something is wrong.
 *
 * Check each global declaration, then the declarations in main.
 *)

let check prog =
    let Program(pdecls, main) = prog in

    (* Add built-in functions and main to list of program declarations *)
    let built_in_funcs = [
        Func("emit", ScalarType TNone, [], []);
        Func("len", ScalarType(TInt(true,64)), [], []);
    ]
    and main_func =
```

```
    Func("main", ScalarType(TInt(false,32)), [], main @ [Return(LInt 0)])
in

let pdecls = built_in_funcs @ pdecls @ [main_func] in

(* Build global maps *)
let ctx = {
    variables = StringMap.empty;
    functions = StringMap.empty;
} in

coerce_sprog (check_pdecls ctx pdecls)
```

9.7. emit.ml

```
open Str

(* Ad-hoc parsing of the string passed to 'emit' *)

type token =
    STR of string
  | VAR of string

let parse_emit_fmt s =
    let s = Str.global_replace (Str.regexp "%") "%%" s in
    let id = Str.regexp "{[a-z_][a-zA-Z0-9_]*}" in

    let rec tokens = function
        [] -> []
      | Text(s)::tl ->
            STR(s)::(tokens tl)
      | Delim(s)::tl ->
            VAR(Str.global_replace (Str.regexp "{\\|}") "" s)::(tokens tl)
    in

    tokens (Str.full_split id s)
```

9.8. codegen.ml

```
(* Code generation: translate a semantically checked AST into LLVM IR *)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

type ctx = {
    vars : L.llvalue StringMap.t list;  (* stack of vars *)
    funcs : L.llvalue StringMap.t;      (* functions *)
    templs : L.llvalue StringMap.t;     (* templates *)
    cur_func : L.llvalue option;        (* current function *)
}

let size_t = size_t

let string_of_ctx name ctx =
    let string_of_var k lv =
        "var " ^ k ^ ": " ^ (L.string_of_llvalue lv)
    and string_of_func k lv =
        "func " ^ k ^ ": " ^ (L.string_of_llvalue lv)
    in

    let string_of_vars m i =
        StringMap.fold (fun k v r -> i^(string_of_var k v)^"\n"^r) m ""
    and string_of_funcs m i =
        StringMap.fold (fun k v r -> i^(string_of_func k v)^"\n"^r) m ""
    in

    let string_of_stack l i =
        List.fold_left
            (fun r v -> i ^ "[" ^ (string_of_vars v i) ^ "]\n" ^ r)
        "" l
    in

      name ^ " ctx {\n"
    ^ "  vars: {\n" ^ (string_of_stack ctx.vars "    ") ^ "  }\n"
    ^ "  funcs: {\n" ^ (string_of_funcs ctx.funcs "  ") ^ "  }\n"
    ^ "}"

(* Lookup a var in a chain of StringMaps *)
let rec chain_lookup k = function
    [] -> raise Not_found
  | m::tl -> try StringMap.find k m with Not_found -> chain_lookup k tl

let lookup_var k ctx =
    try chain_lookup k ctx.vars
```

```
    with Not_found -> raise (Failure ("variable " ^ k ^ " not found"))

(* Adds a var to the top var map of the context ctx *)
let add_var ctx (id:string) (lv:L.llvalue) =
    match ctx.vars with
        [] -> raise (Failure "internal error: add_var on non-existent map")
      | hd::tl ->
            let ctx = { ctx with vars = (StringMap.add id lv hd)::tl } in
            ctx

let translate prog =
    let context = L.global_context () in

    (* LLVM compilation module *)
    let the_module = L.create_module context "BitTwiddler" in

    (* Get types from context *)
    let i1_t   = L.i1_type context
    and i8_t   = L.i8_type context
    and i16_t  = L.i16_type context
    and i32_t  = L.i32_type context
    and i64_t  = L.i64_type context
    and f32_t  = L.float_type context
    and f64_t  = L.double_type context
    and void_t = L.void_type context in

    let __bt_arr_t = L.struct_type context [|
        i64_t; i64_t; i8_t; L.pointer_type i8_t; L.pointer_type i8_t
    |] in

    let __bt_str_t = L.struct_type context [|
        i64_t; L.pointer_type __bt_arr_t; L.pointer_type i8_t
    |] in


    (* Get LLVM type from BitTwiddler type *)
    let ltype_of_type = function
        SScalar t -> (match t with
            A.TInt(_,w) -> L.integer_type context w
          | A.TFloat(32) -> f32_t
          | A.TFloat(64) -> f64_t
          | A.TBool -> i1_t
          | A.TNone -> void_t
          | A.TString -> L.pointer_type __bt_str_t
           | _ -> raise (Failure ("type not implemented " ^ A.string_of_ptype
t))
        )
      | SArray _ -> L.pointer_type __bt_arr_t
    in

    (* Create an alloca instruction in the entry block of the function *)
    let create_entry_block_alloca the_function id type_ =
```

```ocaml
    let builder = L.builder_at context (
        L.instr_begin (L.entry_block the_function)
    ) in
    L.build_alloca (ltype_of_type type_) id builder
in

(* Add a terminal to a block *)
let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
        Some _ -> ()
      | None   -> ignore (instr builder)
in

(* Compare to zero *)
let build_is_nonzero v builder =
    let zero = L.const_int (L.type_of v) 0 in
    L.build_icmp L.Icmp.Ne v zero "tmp" builder
in

(* Runtime functions *)
let __bt_emit : L.llvalue =
    let __bt_emit_t =
        L.var_arg_function_type void_t [| i32_t; L.pointer_type i8_t |]
    in
    L.declare_function "__bt_emit" __bt_emit_t the_module
in

let __bt_read n t =
    let ftype = L.function_type t [| |] in
    L.declare_function ("__bt_read_" ^ n) ftype the_module
in

let __bt_read_i8  = __bt_read "i8"  i8_t in
let __bt_read_i16 = __bt_read "i16" i16_t in
let __bt_read_i32 = __bt_read "i32" i32_t in
let __bt_read_i64 = __bt_read "i64" i64_t in
let __bt_read_f32 = __bt_read "f32" f32_t in
let __bt_read_f64 = __bt_read "f64" f64_t in

let __bt_read_str =
    let ftype = L.function_type (L.pointer_type __bt_str_t) [| |] in
    L.declare_function "__bt_str_read" ftype the_module
in

let __bt_read_arr =
    let ftype = L.function_type (L.pointer_type __bt_arr_t) [|
        i64_t; i64_t
    |] in
    L.declare_function "__bt_arr_read" ftype the_module
in

let __bt_arr_new =
```

```
    let __bt_arr_new_t =
        L.function_type (L.pointer_type __bt_arr_t) [|
            i64_t; i64_t; L.pointer_type i8_t
        |]
    in
    L.declare_function "__bt_arr_new" __bt_arr_new_t the_module
in

let __bt_str_new =
    let __bt_str_new_t =
        L.function_type (L.pointer_type __bt_str_t) [|
            L.pointer_type i8_t
        |]
    in
    L.declare_function "__bt_str_new" __bt_str_new_t the_module
in

let __bt_str_concat =
    let __bt_str_concat_t =
        L.function_type (L.pointer_type __bt_str_t) [|
            L.pointer_type __bt_str_t; L.pointer_type __bt_str_t;
        |]
    in
    L.declare_function "__bt_str_concat" __bt_str_concat_t the_module
in

(* Expression builder *)
let rec build_expr ctx builder = function
    (t, SLInt i) -> (builder, L.const_int (ltype_of_type t) i)
  | (t, SLFloat f) -> (builder, L.const_float (ltype_of_type t) f)
  | (_, SLBool b) -> (builder, L.const_int i1_t (if b then 1 else 0))

  | (_, SLString s) ->
        let gptr = L.build_global_stringptr s "" builder in
        let s =
            L.build_call __bt_str_new [| gptr |] "__bt_str_new" builder
        in
        (builder, s)

  | (SArray(t,_), SLArray el) ->
        let lt = ltype_of_type (SScalar t) in
        let n = List.length el in

        (* builds list of built exprs *)
        let rec to_lv builder = function
            [] -> []
          | hd::tl ->
                let (builder, le) = build_expr ctx builder hd in
                le::(to_lv builder tl)
        in

        let v = L.define_global "lit_arr" (L.const_array
```

```
            (L.array_type lt n) (Array.of_list (to_lv builder el))
          ) the_module in

          let v_ptr = L.const_pointercast v (L.pointer_type i8_t) in

          let a =
              L.build_call __bt_arr_new [|
                  L.const_int i64_t n; L.size_of lt; v_ptr
              |] "__bt_arr_new" builder
          in
          (builder, a)

    | (_, SId id) ->
          (builder, L.build_load (lookup_var id ctx) id builder)

      (* Assignment *)
    | (SScalar _, SBinop ((_, SId id), A.Assign, e)) ->
          let (builder, e') = build_expr ctx builder e in
          ignore(L.build_store e' (lookup_var id ctx) builder);
          (builder, e')

      (* Array subscript a[i]*)
    | (t, SBinop (a, A.Subscr, i)) ->
          let (builder, i') = build_expr ctx builder i in
          let (builder, a') = build_expr ctx builder a in
          let a' = match fst a with
              SArray _ -> a'
            | SScalar A.TString ->
                    let arr_ptr = L.build_struct_gep a' 1 "arr_ptr" builder
in
                  L.build_load arr_ptr "arr" builder
            | t -> raise (Failure ("Operator " ^ A.string_of_op A.Subscr
                                  ^ " not implemented for type "
                                  ^ string_of_stype t))
          in

          let data_ptr = L.build_struct_gep a' 3 "data_ptr" builder in
          let data = L.build_load data_ptr "data" builder in
          let el_ptr_t = L.pointer_type (match t with
              SScalar A.TString -> i8_t
            | _ -> ltype_of_type t)
          in
          let data_cast =
              L.build_pointercast data el_ptr_t "data_cast" builder
          in
          let el_ptr = L.build_gep data_cast [| i' |] "el_ptr" builder in
          (builder, L.build_load el_ptr "el" builder)

      (* Binary operation on integers *)
    | (_, SBinop ((SScalar A.TInt (u,_),_) as e1, op, e2)) ->
          let (_,e1') = build_expr ctx builder e1
          and (_,e2') = build_expr ctx builder e2 in
```

```
            let r = (match op with
                A.Plus     -> L.build_add
              | A.Minus    -> L.build_sub
              | A.Times    -> L.build_mul
              | A.Div      -> if u then L.build_udiv else L.build_sdiv
              | A.Rem      -> if u then L.build_urem else L.build_srem
              | A.LShift   -> L.build_shl
              | A.RShift   -> if u then L.build_lshr else L.build_ashr
              | A.BwOr     -> L.build_or
              | A.BwAnd    -> L.build_and
                   | A.Lt        -> L.build_icmp (if u then L.Icmp.Ult else
L.Icmp.Slt)
                   | A.LtEq      -> L.build_icmp (if u then L.Icmp.Ule else
L.Icmp.Sle)
              | A.Eq       -> L.build_icmp L.Icmp.Eq
              | A.NEq      -> L.build_icmp L.Icmp.Ne
                   | A.GtEq      -> L.build_icmp (if u then L.Icmp.Uge else
L.Icmp.Sge)
                   | A.Gt        -> L.build_icmp (if u then L.Icmp.Ugt else
L.Icmp.Sgt)
              | _ -> raise (Failure ("internal error: operation "
                                    ^ A.string_of_op op
                                    ^ " not implemented for integers"))
            ) e1' e2' "tmp" builder in
            (builder, match op with
                A.And | A.Or -> build_is_nonzero r builder
              | _ -> r)

        (* Binary operation on floats *)
      | (_, SBinop ((SScalar A.TFloat _, _) as e1, op, e2)) ->
            let (_,e1') = build_expr ctx builder e1
            and (_,e2') = build_expr ctx builder e2 in
            let r = (match op with
                  A.Plus     -> L.build_fadd
                | A.Minus    -> L.build_fsub
                | A.Times    -> L.build_fmul
                | A.Div      -> L.build_fdiv
                | A.Lt       -> L.build_fcmp L.Fcmp.Olt
                | A.LtEq     -> L.build_fcmp L.Fcmp.Ole
                | A.Eq       -> L.build_fcmp L.Fcmp.Oeq
                | A.NEq      -> L.build_fcmp L.Fcmp.One
                | A.GtEq     -> L.build_fcmp L.Fcmp.Oge
                | A.Gt       -> L.build_fcmp L.Fcmp.Ogt
                | _ -> raise (Failure ("internal error: operation "
                                    ^ A.string_of_op op
                                    ^ " not implemented for floats"))
            ) e1' e2' "tmp" builder in
            (builder, r)

        (* Binary operation on bools *)
      | (_, SBinop ((SScalar A.TBool,_) as e1, op, e2)) ->
```

```
        let (_,e1') = build_expr ctx builder e1
        and (_,e2') = build_expr ctx builder e2 in
        let r = (match op with
              A.Or   -> L.build_or
            | A.And  -> L.build_and
            | _ -> raise (Failure ("internal error: operation "
                                   ^ A.string_of_op op
                                   ^ " not implemented for bools"))
        ) e1' e2' "tmp" builder in
        (builder, r)

    (* Binary operation on strings *)
  | (_, SBinop ((SScalar A.TString,_) as e1, A.Plus, e2)) ->
        let args' = List.map (build_expr ctx builder) [e1; e2] in
        let args' = Array.of_list (List.map snd args') in
        (builder, L.build_call __bt_str_concat args'
                              "__bt_str_concat" builder)

    (* Unary operation on integer *)
  | (SScalar (A.TInt _), SUnop (uop, e)) ->
        let (builder, e') = build_expr ctx builder e in
        (builder, match uop with
            A.BwNot -> L.build_not e' "tmp" builder
          | A.Neg -> L.build_neg e' "tmp" builder
          | _ -> raise (Failure ("internal error: operation "
                                 ^ A.string_of_uop uop
                                 ^ " not implemented for integers")))

    (* Unary operation on float *)
  | (SScalar (A.TFloat _), SUnop (A.Neg, e)) ->
        let (builder, e') = build_expr ctx builder e in
        (builder, L.build_fneg e' "tmp" builder)

    (* Unary operation on bool *)
  | (SScalar A.TBool, SUnop (A.Not, e)) ->
        let (builder, e') = build_expr ctx builder e in
        (builder, L.build_not e' "not" builder)

    (* Conditional block (it's an expression) *)
  | (t, SIf (pred, then_, else_)) ->
        let the_function = match ctx.cur_func with
            Some f -> f
          | None -> raise (Failure "internal error: no current function")
        in

        (* The result of the block expression will be stored here *)
        let block_res = match t with
            SScalar A.TNone -> None
          | SScalar _ ->
                  Some (create_entry_block_alloca the_function "blres" t)
          | SArray _ -> raise (Failure "arrays not implemented yet")
        in
```

```
            (* Build predicate *)
            let (_,pred') = build_expr ctx builder pred in
            let merge_bb = L.append_block context "merge" the_function in
              let build_br_merge = L.build_br merge_bb in (* partial function
*)

            (* Build then block *)
            let then_bb = L.append_block context "then" the_function in
            let then_builder = (L.builder_at_end context then_bb) in
             let then_builder = (build_block ctx then_builder block_res then_)
in
            add_terminal then_builder build_br_merge;

            (* Build else block *)
            let else_bb = L.append_block context "else" the_function in
            let else_builder = (L.builder_at_end context else_bb) in
             let else_builder = (build_block ctx else_builder block_res else_)
in
            add_terminal else_builder build_br_merge;

            let _ = L.build_cond_br pred' then_bb else_bb builder in
            let _ = L.move_block_after else_bb merge_bb in
            let builder = L.builder_at_end context merge_bb in
            (builder, match block_res with
                Some v -> L.build_load v "res" builder
              | None -> L.undef void_t)

      | (_, SCall("__bt_emit", args)) ->
            let args' = List.map (build_expr_s ctx builder) args in
            let args' = Array.of_list (List.map snd args') in
            (builder, L.build_call __bt_emit args' "" builder)

      | (_, SCall("__bt_len", [e])) ->
            let (builder, e') = build_expr ctx builder e in
            let n = L.build_struct_gep e' 0 "n" builder in
            (builder, L.build_load n "len" builder)

      | (SScalar t, SCall("__bt_read", _)) ->
            let f = match t with
                A.TInt(_,8)  -> __bt_read_i8
              | A.TInt(_,16) -> __bt_read_i16
              | A.TInt(_,32) -> __bt_read_i32
              | A.TInt(_,64) -> __bt_read_i64
              | A.TFloat 32  -> __bt_read_f32
              | A.TFloat 64  -> __bt_read_f64
              | A.TString    -> __bt_read_str
              | _ -> raise (Failure ("automatic reading of scalar type "
                                ^ A.string_of_ptype t
                                ^ " not implemented"))
            in
```

```
        (builder, L.build_call f [| |] "__bt_read" builder)

    | (SArray(t, Some n), SCall("__bt_read", _)) ->
        let builder, n' = build_expr ctx builder n in
        let elsz = L.const_int (ltype_of_type size_t) (
            match t with
                A.TInt(_,w) | A.TFloat w -> w/8
              | _ -> raise (Failure ("automatic reading of array type "
                                     ^ A.string_of_ptype t
                                     ^ " not implemented"))
        ) in
        (builder, L.build_call __bt_read_arr [|n'; elsz|]
                               "__bt_read_arr" builder)

    | (t, SCall(fname, args)) ->
        let args' = List.map (build_expr ctx builder) args in
        let args' = Array.of_list (List.map snd args') in

        let call =
            let lf =
                try StringMap.find fname ctx.funcs
                with Not_found ->
                    raise (Failure ("function " ^ fname ^ " not found"))
            in
            let result = match t with
                SScalar A.TNone -> ""
              | _ -> fname ^ "_result"
            in
            L.build_call lf args' result builder
        in
        (builder, call)

     | _ as e -> raise (Failure ("expr not implemented: " ^ string_of_sexpr
e))

  (* Build an expression, flattening strings *)
  and build_expr_s ctx builder = function
      (SScalar A.TString, _) as e ->
        let (builder, e') = build_expr ctx builder e in
        let arr_ptr = L.build_struct_gep e' 1 "arr_ptr" builder in
        let arr = L.build_load arr_ptr "arr" builder in
        let data_ptr = L.build_struct_gep arr 3 "data_ptr" builder in
        let data = L.build_load data_ptr "data" builder in
        (builder, data)
    | _ as e -> build_expr ctx builder e

  (*
   * Build an sstmt
   *)
  and build_stmt ctx builder = function
      SLVar(id, type_, e) ->
        (match ctx.cur_func with
```

```
                (* Allocate var on the stack and add it to the context *)
                Some f ->
                    let lv = create_entry_block_alloca f id type_ in
                    let ctx = add_var ctx id lv in
                    let builder = match e with
                        Some e ->
                            let (builder, e') = build_expr ctx builder e in
                            ignore(L.build_store e' lv builder);
                            builder
                      | None ->
                            builder
                    in
                    (None, ctx, builder)
              | None ->
                    raise (Failure "internal error: local var without stack")
            )

    | SExpr e ->
            let (builder, e') = build_expr ctx builder e in
            let lval = match e with
                (SScalar A.TNone, _) -> None
              | _ -> Some e'
            in
            (lval, ctx, builder)

    | SReturn e ->
              let (builder, e') = build_expr ctx builder e in
              ignore(L.build_ret e' builder);
              (None, ctx, builder)

    | SWhile(pred, body) ->
            let the_function = match ctx.cur_func with
                Some f -> f
              | None -> raise (Failure "internal error: no current function")
            in

            (* Build predicate *)
            let pred_bb = L.append_block context "while" the_function in
            let _ = L.build_br pred_bb builder in

            (* Build body *)
            let body_bb = L.append_block context "while_body" the_function in
            let body_builder = (L.builder_at_end context body_bb) in
            let body_builder = (build_block ctx body_builder None body) in
            add_terminal body_builder (L.build_br pred_bb);

            let pred_builder = L.builder_at_end context pred_bb in
            let (_,bool_val) = build_expr ctx pred_builder pred in

            let merge_bb = L.append_block context "merge" the_function in
            let _ = L.build_cond_br bool_val body_bb merge_bb pred_builder in
            (None, ctx, L.builder_at_end context merge_bb)
```

```
    (* A for statement is compiled, conceptually, to:
     *
     *     var idx:uint64 = 0;
     *     var item:item_t;
     *     var n:uint64 = len(e);
     *
     *     while idx < n {
     *         item = e[idx];
     *         ... block ...
     *         idx = idx + 1;
     *     }
     *)
  | SFor(idx_sv, item_sv, e, block) ->
        (* Declare and initialize relevant, new variables *)
        let (idx, _, _) = idx_sv in
        let (item, item_t, _) = item_sv in
        let len_sv =
            ("__len", size_t, Some (size_t, SCall("__bt_len", [e])))
        in

        (* Read item from array: item = e[idx] *)
        let pre_block:sstmt = SExpr(SScalar A.TNone, SBinop(
            (item_t, SId item),
            A.Assign,
            (item_t, SBinop(e, A.Subscr, (size_t, SId idx)))
        )) in

        (* Increment index: idx = idx + 1*)
        let post_block :sstmt = SExpr(SScalar A.TNone, SBinop(
            (size_t, SId idx),
            A.Assign,
            (size_t, SBinop((size_t, SId idx),
                            A.Plus,
                            (size_t, SLInt 1)))
        )) in

        (* Build while loop: while(idx < len) { ... } *)
        let while_ = SWhile(
            (SScalar A.TBool, SBinop(
                (size_t, SId idx),
                A.Lt,
                (size_t, SId "__len"))),
           [pre_block] @ block @ [post_block]
        ) in

        (* Generate the code *)
        let (_, _, builder) = List.fold_left (
            fun (_, lctx, builder) stmt -> build_stmt lctx builder stmt
        ) (None, ctx, builder)
        [SLVar idx_sv; SLVar item_sv; SLVar len_sv; while_] in
```

```
            (* Return old context (don't keep newly created vars) *)
            (None, ctx, builder)

    (*
     * Build a block
     *)
    and build_block ctx builder res = function
        [] ->
            builder
      | [item] ->
            (* Store the value of the last item in res *)
            let (lval, _, builder) = build_stmt ctx builder item in
            let _ = match (res, lval) with
                (Some r, Some v) -> ignore(L.build_store v r builder)
              | (None, _) -> ()
               | (Some _, None) -> raise (Failure ("Block expected to end with
"
                                                  ^ "an expression"))
            in
            builder
      | hd::tl ->
            let (_, ctx, builder) = build_stmt ctx builder hd in
            build_block ctx builder res tl
    in

    (*
     * Build a sprogram_decl
     *)

    (* Helper: list of parameters -> array of ltypes *)
    let ltypes_of_params params = Array.of_list (
        List.map (fun (_, t) -> ltype_of_type t) params
    ) in

    (* Helper: 'LLVM return' from function type *)
    let ret_of_type t = match t with
        SScalar pt -> (match pt with
            A.TInt(_,_) -> L.build_ret (L.const_int (ltype_of_type t) 0)
          | A.TFloat(_) -> L.build_ret (L.const_float (ltype_of_type t) 0.0)
          | A.TNone -> L.build_ret_void
                  |  _  ->  raise  (Failure  ("ret  type  not  implemented  "  ^
A.string_of_ptype pt))
        )
       | _ -> raise (Failure ("ret type not implemented " ^ string_of_stype
t))
    in

    (* Build a global variable *)
    let build_gvar ctx v =
        let (id, type_, _) = v in
        let ltype = ltype_of_type type_ in
```

```
        let init = match type_ with
            SScalar(A.TInt(_,_)) -> L.const_int ltype 0
          | SScalar(A.TFloat(_)) -> L.const_float ltype 0.0
          | _ -> raise (Failure ("gvar init " ^ (string_of_stype type_)
                                    ^ "not implemented"))
        in
        let the_var = L.define_global id init the_module in
        add_var ctx id the_var
    in

    (* Build a function *)
    let build_func ctx f =
        let (id, type_, params, body) = f in
        let params_ltypes = ltypes_of_params params in
        let ftype = L.function_type (ltype_of_type type_) params_ltypes in
        let the_function = L.define_function id ftype the_module in
          let builder = L.builder_at_end context (L.entry_block the_function)
in

        (* Add a parameter to the stack *)
        let add_param ctx id type_ =
            let lv = create_entry_block_alloca the_function id type_ in
            add_var ctx id lv
        in

        let rec add_params ctx = function
            [] -> ctx
          | (id,type_)::tl ->
                let ctx = add_param ctx id type_ in
                add_params ctx tl
        in

        let add_terminal builder instr =
            match L.block_terminator (L.insertion_block builder) with
                Some _ -> ()
              | None   -> ignore (instr builder)
        in

        (* Allocate parameters on the stack and assign passed in values *)
        let lctx = add_params ctx params in
        let _ = List.iter2
            (fun (id,_) value ->
                ignore(L.build_store value (lookup_var id lctx) builder))
            params
            (Array.to_list (L.params the_function))
        in

        (* Create function's local context *)
        let lctx = { lctx with
            funcs = StringMap.add id the_function lctx.funcs;
            cur_func = Some the_function;
            vars = StringMap.empty::lctx.vars;
```

```
    } in

    (* Build the body of the function *)
    let builder = build_block lctx builder None body in
    let _ = add_terminal builder (ret_of_type type_) in

    (* Returns a context with this function added to it *)
    { ctx with funcs = StringMap.add id the_function ctx.funcs }
in

(* Build a program declaration *)
let build_pdecl ctx = function
    SGVar(v) -> build_gvar ctx v
  | SFunc(f) -> build_func ctx f
in

(* Build all program declarations *)
let rec build_pdecls ctx = function
    [] -> ()
  | hd::tl ->
        let ctx = build_pdecl ctx hd in
        build_pdecls ctx tl
in

let global_ctx = {
    vars = [StringMap.empty];
    funcs = StringMap.empty;
    templs = StringMap.empty;
    cur_func = None;
} in

build_pdecls global_ctx prog;
the_module
```

9.9. runtime.c

```c
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <stdbool.h>

/*
 *
 * BitTwiddler runtime types and pools
 *
 */
struct __bt_arr {
    size_t n;               // Number of elements in the array
    size_t elsz;            // Size of each element
    bool owns_data;         // Does this array own the data in *data?
    void *data;             // Backing data
    struct __bt_arr *next;  // Next array in the linked list
};


struct __bt_str {
    size_t n;               // Number of characters, excluding NUL
    struct __bt_arr *arr;   // Backing array
    struct __bt_str *next;  // Next string in the linked list
};

static struct __bt_arr *__bt_arr_ll = NULL;
static struct __bt_str *__bt_str_ll = NULL;

/*
 *
 * Forward declarations
 *
 */
void __bt_read(void *target, size_t n);

/*
 *
 * Debug functions
 *
 */
void __bt_dbg_print_arr(struct __bt_arr *a) {
    fprintf(stderr, "[arr n=%lu elsz=%lu owns=%d data=%p]\n",
            a->n, a->elsz, a->owns_data, a->data);
}
```

```
void __bt_dbg_print_str(struct __bt_str *s) {
    fprintf(stderr, "[str len=%lu arr=[%p", s->n, s->arr);

    if (s->arr)
          fprintf(stderr, " n=%lu elsz=%lu] %p '%s']\n", s->arr->n, s->arr-
>elsz,
                s->arr->data, (char*)s->arr->data);
    else
        fprintf(stderr, "]\n");
}

void __bt_dbg_print_arr_pool(void) {
    fprintf(stderr, "Array Pool {\n");
    struct __bt_arr *a = __bt_arr_ll;
    while (a) {
        __bt_dbg_print_arr(a);
        a = a->next;
    }
    fprintf(stderr, "}\n");
}

void __bt_dbg_print_str_pool(void) {
    fprintf(stderr, "String Pool {\n");
    struct __bt_str *s = __bt_str_ll;

    while (s) {
        __bt_dbg_print_str(s);
        s = s->next;
    }
    fprintf(stderr, "}\n");
}

/*
 *
 * BitTwiddler arrays
 *
 */

void __bt_arr_freeall(void) {
    struct __bt_arr *next, *p = __bt_arr_ll;

    while (p) {
        next = p->next;

        if (p->owns_data)
            free(p->data);

        free(p);
        p = next;
    }
}
```

```c
// Allocates, zero-initializes and returns a new BitTwiddler array
struct __bt_arr *__bt_arr_new(size_t n, size_t elsz, void *data)
{
    // Free all arrays at program exit (register this function when the first
    // array is created).
    if (!__bt_arr_ll)
        atexit(__bt_arr_freeall);

    // Allocate space for a new array structure
    struct __bt_arr *a = calloc(1, sizeof(*a));

    if (!a) {
            fprintf(stderr, "Runtime Error: %s failed to allocate new array
node\n",
                __func__);
        exit(1);
    }

    // Allocate space for data
    a->n = n;
    a->elsz = elsz;
    a->owns_data = data == NULL;
    a->data = a->owns_data ? calloc(n, elsz) : data;

    if (!a->data) {
        fprintf(stderr, "Runtime Error: %s failed to allocate new array\n",
            __func__);
        exit(1);
    }

    // Put it in the pool
    a->next = __bt_arr_ll;
    __bt_arr_ll = a;

    return a;
}

// Resizes an array
void __bt_arr_resize(struct __bt_arr *arr, size_t new_n) {
    char *data = realloc(arr->data, arr->elsz*new_n);

    if (!data) {
            fprintf(stderr, "Runtime Error: %s failed to resize array\n",
__func__);
        exit(1);
    }

    arr->data = data;
    arr->n = new_n;
}
```

```
// Reads a BitTwiddler array from the standard input
struct __bt_arr *__bt_arr_read(size_t n, size_t elsz) {
    struct __bt_arr *a = __bt_arr_new(n, elsz, NULL);
    __bt_read(a->data, n*elsz);
    return a;
}

/*
 *
 *  BitTwiddler strings
 *
 */

// Frees all BitTwiddler strings, runs at program exit
// Backing arrays will be freed by __bt_arr_freeall
void __bt_str_freeall(void) {
    struct __bt_str *next, *p = __bt_str_ll;

    while (p) {
        next = p->next;
        free(p);
        p = next;
    }
}

// Allocates, initializes and returns a new BitTwiddler string
struct __bt_str *__bt_str_new(const char *val) {
     // Free all strings at program exit (register this function when the
first
    // string is created).
    if (!__bt_str_ll)
        atexit(__bt_str_freeall);

    // Allocate space for a new string structure
    struct __bt_str *s = calloc(1, sizeof(*s));

    if (!s) {
        fprintf(stderr, "Runtime Error: %s failed to allocate new string\n",
                __func__);
        exit(1);
    }

    // Put it in the pool
    s->next = __bt_str_ll;
    __bt_str_ll = s;

    if (val) {
        size_t n = strlen(val);
        s->n = n;
        s->arr = __bt_arr_new(n + 1, 1, (void*) val);
    }
```

```
    return s;
}

// Concatenates two BitTwiddler strings into a third, new string
struct __bt_str *__bt_str_concat(struct __bt_str *s1, struct __bt_str *s2) {
    struct __bt_str *s = __bt_str_new(NULL);

    s->n = s1->n + s2->n;
    s->arr = __bt_arr_new(s->n + 1, sizeof(char), NULL);

    strcpy(s->arr->data, s1->arr->data);
    strcat(s->arr->data, s2->arr->data);
    return s;
}

// Reads a BitTwiddler string from the standard input (NUL-terminated)
struct __bt_str *__bt_str_read(void) {

    // Start with a small buffer
    size_t sz = 32;
    char *buf = calloc(sz, sizeof(*buf));
    size_t n = 0;

    if (!buf) {
        fprintf(stderr, "Runtime Error: %s failed to allocate buffer for"
                        " string from stdin\n", __func__);
        exit(1);
    }

    for(;;) {
        int c = fgetc(stdin);

        if (c == EOF) {
            fprintf(stderr, "Runtime Error: %s EOF while reading string\n",
                    __func__);
            exit(1);
        }

        // Found end of string
        if (!c)
            break;

        buf[n++] = (char) c;

        // Buffer size needs to increase
        if (n == sz - 1) {
            sz *= 2;
            buf = realloc(buf, sz);
            if (!buf) {
                fprintf(stderr, "Runtime Error: %s failed to allocate buffer"
                                " for string from stdin\n", __func__);
                exit(1);
```

```
            }
        }
    }

    buf[n] = '\0';

    return __bt_str_new(buf);
}

/*
 *
 * BitTwiddler reading from stdin
 *
 */
void __bt_read(void *target, size_t n) {
    size_t r = fread(target, 1, n, stdin);

    if (r == n)
        return;

    const char *err = ".";
    if (feof(stdin))
        err = ": reached EOF";

     fprintf(stderr, "Failed to read %lu bytes from standard input%s\n", n,
err);
    fprintf(stderr, "Aborting...\n");
    exit(1);
}

int8_t  __bt_read_i8(void)  { int8_t v;  __bt_read((void*)&v, 1); return v; }
int16_t __bt_read_i16(void) { int16_t v; __bt_read((void*)&v, 2); return v; }
int32_t __bt_read_i32(void) { int32_t v; __bt_read((void*)&v, 4); return v; }
int64_t __bt_read_i64(void) { int64_t v; __bt_read((void*)&v, 8); return v; }
float   __bt_read_f32(void) { float v;   __bt_read((void*)&v, 4); return v; }
double  __bt_read_f64(void) { double v;  __bt_read((void*)&v, 8); return v; }



/*
 *
 * BitTwiddler printing function
 *
 */

enum __bt_emit_kind {
    __BT_EMIT_EMIT,
    __BT_EMIT_PRINT,
    __BT_EMIT_FATAL
};

void __bt_emit(int kind, const char *fmt, ...) {
```

```
    FILE * stream = kind == __BT_EMIT_EMIT ? stdout : stderr;

    va_list args;
    va_start(args, fmt);
    vfprintf(stream, fmt, args);
    va_end(args);

    if (kind == __BT_EMIT_FATAL) {
        exit(1);
    }
}
```

9.10. Makefile

```
.PHONY: all test clean

all: bittwiddler.native runtime.o gen_bin_data

gen_bin_data: gen_bin_data.c
	$(CC) -o $@ $<

test: all testall.sh
	./testall.sh

bittwiddler.native:
	opam config exec -- \
	ocamlbuild -use-ocamlfind bittwiddler.native

clean:
	ocamlbuild -clean
	-rm -f *diff *.ll *.err *.s *.out *.exe testall.log runtime.o gen_bin_data
```

9.11. _tags

```
# Include the llvm and llvm.analysis packages while compiling
true: package(llvm), package(llvm.analysis), package(str)

# Enable almost all compiler warnings
true : warn(+a-4)

true: debug

"runtime.o": not_hygienic
```

9.12. compile.sh

```
#!/bin/sh

# Path to the LLVM interpreter
LLI="lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the bittwiddler compiler.  Usually "./bittwiddler.native"
BITTWIDDLER="./bittwiddler.native"

# Set time limit for all operations
ulimit -t 30

$BITTWIDDLER $1 | $LLC -relocation-model=pic > $1.s
$CC -o $1.exe $1.s runtime.o

rm $1.s
```

9.13. gen_bin_data.c

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int fd = STDOUT_FILENO;

int main(int argc, char **argv) {
    argc--; argv++;

    while(argc) {
        char *t = *(argv++);
        char *d = *(argv++);
        argc -= 2;

        if (!strcmp(t, "s")) {
            write(fd, (const void*)d, strlen(d) + 1);
        }
#define CASE(N,T,S,F)\
        else if (!strcmp(t,N)){T v = F(d);write(fd, (const void*)&v, S);}
        CASE("i8",  int8_t,   1, atoi)
        CASE("i16", int16_t,  2, atoi)
        CASE("i32", int32_t,  4, atoi)
        CASE("i64", int64_t,  8, atoi)
        CASE("u8",  uint8_t,  1, atoi)
        CASE("u16", uint16_t, 2, atoi)
        CASE("u32", uint32_t, 4, atoi)
        CASE("u64", uint64_t, 8, atoi)
        CASE("f32", float,    4, atof)
        CASE("f64", double,   8, atof)
#undef CASE
        else {
            printf("unmatched\n");
        }
    }
    return 0;
}
```

9.14. testall.sh

```
#!/bin/sh

# Regression testing script for BitTwiddler
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the bittwiddler compiler.  Usually "./bittwiddler.native"
#  Try "_build/bittwiddler.native" if ocamlbuild was unable to create a
symbolic link.
BITTWIDDLER="./bittwiddler.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.bt files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
```

```
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.bt//'`
    reffile=`echo $1 | sed 's/.bt$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

           generatedfiles="$generatedfiles  ${basename}.ll  ${basename}.s  $
{basename}.exe ${basename}.out" &&
    Run "$BITTWIDDLER" "$1" ">" "${basename}.ll" &&
     Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s"
&&
```

```
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "runtime.o" &&
    if [ ! -f "${reffile}.in" ]; then
        Run "./${basename}.exe" ">" "${basename}.out"
    else
        Run "./${basename}.exe" ">" "${basename}.out" "<" "${reffile}.in"
    fi
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.bt//'`
    reffile=`echo $1 | sed 's/.bt$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$BITTWIDDLER" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
```

```
}

while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
   echo "Check your LLVM installation and/or modify the LLI variable in
testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.bt tests/fail-*.bt"
fi

for file in $files
do
    case $file in
  *test-*)
      Check $file 2>> $globallog
      ;;
  *fail-*)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
done

exit $globalerror
```

10. Test code listing

10.1. fail-001-dup-global.bt

```
var x : int32 = 0;
var x : int64 = 1;

main {
}
```

10.2. fail-001-dup-global.err

```
Fatal error: exception Failure("duplicate variable declaration: x")
```

10.3. fail-002-dup-function.bt

```
func test (a:int32):None {
}

func test (b:int64):None {
}

main {}
```

10.4. fail-002-dup-function.err

```
Fatal error: exception Failure("duplicate function declaration: test")
```

10.5. fail-003-lit-array-empty.bt

```
main {
    [];
}
```

10.6. fail-003-lit-array-empty.err

```
Fatal error: exception Failure("Can't determine type of empty literal array")
```

10.7. fail-004-lit-array-mixed-types.bt

```
main {
    [1, "2", 3, 4];
}
```

10.8. fail-004-lit-array-mixed-types.err

```
Fatal error: exception Failure("Array literal has mixed or invalid types")
```

10.9. fail-005-expr-incompatible-types-1.bt

```
main {
    1 + 1.0;
}
```

10.10. fail-005-expr-incompatible-types-1.err

```
Fatal error: exception Failure("Incompatible types int and float")
```

10.11. fail-006-expr-incompatible-types-2.bt

```
main {
    1 + [2];
}
```

10.12. fail-006-expr-incompatible-types-2.err

```
Fatal error: exception Failure("Incompatible types int and int[(1):uint64]")
```

10.13. fail-007-expr-incompatible-types-3.bt

```
main {
    [1.0, 2.0] + [3];
}
```

10.14. fail-007-expr-incompatible-types-3.err

```
Fatal error: exception Failure("Incompatible types float[(2):uint64] and int[(1):uint64]")
```

10.15. fail-008-expr-op-not-defined.bt

```
main {
    1.0 % 2.0;
}
```

10.16. fail-008-expr-op-not-defined.err

```
Fatal error: exception Failure("Operator % not defined for types float64 and float64")
```

10.17. fail-009-undeclared-ident.bt

```
main {
   var x = "x";

   y = 1;
}
```

10.18. fail-009-undeclared-ident.err

```
Fatal error: exception Failure("undeclared identifier y")
```

10.19. fail-010-cant-emit.bt

```
main {
    var x:int8[] = [1, 2, 3];

    emit("{x}\n");
}
```

10.20. fail-010-cant-emit.err

```
Fatal error: exception Failure("don't know how to emit x:int8[]")
```

10.21. fail-011-non-bool-if-predicate.bt

```
main {
    if 1 {
        emit("1\n");
    };
}
```

10.22. fail-011-non-bool-if-predicate.err

```
Fatal error: exception Failure("Non-boolean expression in conditional: 1")
```

10.23. fail-012-non-bool-while-predicate.bt

```
main {
    while 1 {}
}
```

10.24. fail-012-non-bool-while-predicate.err

```
Fatal error: exception Failure("Non-boolean expression in while predicate:
1")
```

10.25. fail-013-conditional-diff-types.bt

```
main {
    if true {
        1;
    } else {
        "1";
    };
}
```

10.26. fail-013-conditional-diff-types.err

```
Fatal error: exception Failure("conditional blocks have different types")
```

10.27. fail-014-emit-arg.bt

```
main {
    var x = "{x}";

    emit(x);
}
```

10.28. fail-014-emit-arg.err

```
Fatal error: exception Failure("'emit' requires a single literal string argument")
```

10.29. fail-015-len-arg.bt

```
main {
    var x:int16 = -5;

    len();
}
```

10.30. fail-015-len-arg.err

```
Fatal error: exception Failure("len requires a single array or string argument")
```

10.31. fail-016-len-arg-2.bt

```
main {
    var x:int16 = -5;

    len(x);
}
```

10.32. fail-016-len-arg-2.err

```
Fatal error: exception Failure("len can't be applied to type: int16")
```

10.33. fail-017-array-size.bt

```
main {
    var x:int16[1.0] = [1];
}
```

10.34. fail-017-array-size.err

```
Fatal error: exception Failure("array size can't be of type float")
```

10.35. fail-018-for-arg.bt

```
main {
    var x:uint32 = 10;
    for i, e in x {}
}
```

10.36. fail-018-for-arg.err

```
Fatal error: exception Failure("Can't iterate over expression of type
uint32")
```

10.37. fail-019-for-arg-2.bt

```
main {
    var x = ["a", "b", "c"];
    for i, e in x {}
}
```

10.38. fail-019-for-arg-2.err

```
Fatal error: exception Failure("Can't iterate over array of strings")
```

10.39. fail-020-illegal-param.bt

```
func test(a:None):None {}
main{}
```

10.40. fail-020-illegal-param.err

```
Fatal error: exception Failure("illegal a : None in test")
```

10.41. test-001-hello.bt

```
main {
    emit("Hello, world!");
}
```

10.42. test-001-hello.out

```
Hello, world!
```

10.43. test-002-emit.bt

```
main {
    var a : int32 = 1;
    var b : int32 = 2;
    var c : int32 = 3;
    emit("{a} + {b} = {c}");
}
```

10.44. test-002-emit.out

```
1 + 2 = 3
```

10.45. test-003-emit-str.bt

```
main {
    var s : string = "Hello, world!";
    emit("{s}");
}
```

10.46. test-003-emit-str.out

```
Hello, world!
```

10.47. test-100-binops-unops.bt

```
main {
    var a_i32 : int32 = 3;
    var b_i32 : int32 = 2;
    var c_i32 : int32 = 0;
    var c_bool : bool = false;

    emit("Binary operations on integers\n");
    c_i32 = a_i32 + b_i32;
    emit("(int32) {a_i32} + {b_i32} = {c_i32}\n");
    c_i32 = a_i32 - b_i32;
    emit("(int32) {a_i32} - {b_i32} = {c_i32}\n");
    c_i32 = a_i32 * b_i32;
    emit("(int32) {a_i32} * {b_i32} = {c_i32}\n");
    c_i32 = a_i32 / b_i32;
    emit("(int32) {a_i32} / {b_i32} = {c_i32}\n");
    c_i32 = a_i32 % b_i32;
    emit("(int32) {a_i32} % {b_i32} = {c_i32}\n");
    c_i32 = a_i32 << b_i32;
    emit("(int32) {a_i32} << {b_i32} = {c_i32}\n");
    c_i32 = a_i32 >> b_i32;
    emit("(int32) {a_i32} >> {b_i32} = {c_i32}\n");
    c_i32 = a_i32 | b_i32;
    emit("(int32) {a_i32} | {b_i32} = {c_i32}\n");
    c_i32 = a_i32 & b_i32;
    emit("(int32) {a_i32} & {b_i32} = {c_i32}\n");
    c_bool = (a_i32 != 0) or (b_i32 != 0);
    emit("(int32) {a_i32} or {b_i32} = {c_bool}\n");
    c_bool = (a_i32 != 0) and (b_i32 != 0);
    emit("(int32) {a_i32} and {b_i32} = {c_bool}\n");
    c_bool = a_i32 < b_i32;
    emit("(int32) {a_i32} < {b_i32} = {c_bool}\n");
    c_bool = a_i32 > b_i32;
    emit("(int32) {a_i32} > {b_i32} = {c_bool}\n");
    c_bool = a_i32 <= b_i32;
    emit("(int32) {a_i32} <= {b_i32} = {c_bool}\n");
    c_bool = a_i32 >= b_i32;
    emit("(int32) {a_i32} >= {b_i32} = {c_bool}\n");
    c_bool = a_i32 == b_i32;
    emit("(int32) {a_i32} == {b_i32} = {c_bool}\n");
    c_bool = a_i32 != b_i32;
    emit("(int32) {a_i32} != {b_i32} = {c_bool}\n");

    emit("\nUnary operations on integers\n");
    c_i32 = ~a_i32;
    emit("(int32) ~{a_i32} = {c_i32}\n");
    c_bool = not (a_i32!=0);
    emit("(int32) not {a_i32} = {c_bool}\n");
    c_i32 = -a_i32;
    emit("(int32) -{a_i32} = {c_i32}\n");
```

```
    var a_f64 : float64 = 3.0;
    var b_f64 : float64 = 2.0;
    var c_f64 : float64 = 0.0;

    emit("\nBinary operations on floats\n");
    c_f64 = a_f64 + b_f64;
    emit("(float64) {a_f64} + {b_f64} = {c_f64}\n");
    c_f64 = a_f64 - b_f64;
    emit("(float64) {a_f64} - {b_f64} = {c_f64}\n");
    c_f64 = a_f64 * b_f64;
    emit("(float64) {a_f64} * {b_f64} = {c_f64}\n");
    c_f64 = a_f64 / b_f64;
    emit("(float64) {a_f64} / {b_f64} = {c_f64}\n");
    c_bool = a_f64 < b_f64;
    emit("(float64) {a_f64} < {b_f64} = {c_bool}\n");
    c_bool = a_f64 > b_f64;
    emit("(float64) {a_f64} > {b_f64} = {c_bool}\n");
    c_bool = a_f64 <= b_f64;
    emit("(float64) {a_f64} <= {b_f64} = {c_bool}\n");
    c_bool = a_f64 >= b_f64;
    emit("(float64) {a_f64} >= {b_f64} = {c_bool}\n");
    c_bool = a_f64 == b_f64;
    emit("(float64) {a_f64} == {b_f64} = {c_bool}\n");
    c_bool = a_f64 != b_f64;
    emit("(float64) {a_f64} != {b_f64} = {c_bool}\n");

    emit("\nUnary operations on floats\n");
    c_f64 = -a_f64;
    emit("(float64) -{a_f64} = {c_f64}\n");
}
```

10.48. test-100-binops-unops.out

```
Binary operations on integers
(int32) 3 + 2 = 5
(int32) 3 - 2 = 1
(int32) 3 * 2 = 6
(int32) 3 / 2 = 1
(int32) 3 % 2 = 1
(int32) 3 << 2 = 12
(int32) 3 >> 2 = 0
(int32) 3 | 2 = 3
(int32) 3 & 2 = 2
(int32) 3 or 2 = 1
(int32) 3 and 2 = 1
(int32) 3 < 2 = 0
(int32) 3 > 2 = 1
```

```
(int32) 3 <= 2 = 0
(int32) 3 >= 2 = 1
(int32) 3 == 2 = 0
(int32) 3 != 2 = 1

Unary operations on integers
(int32) ~3 = -4
(int32) not 3 = 0
(int32) -3 = -3

Binary operations on floats
(float64) 3.000000 + 2.000000 = 5.000000
(float64) 3.000000 - 2.000000 = 1.000000
(float64) 3.000000 * 2.000000 = 6.000000
(float64) 3.000000 / 2.000000 = 1.500000
(float64) 3.000000 < 2.000000 = 0
(float64) 3.000000 > 2.000000 = 1
(float64) 3.000000 <= 2.000000 = 0
(float64) 3.000000 >= 2.000000 = 1
(float64) 3.000000 == 2.000000 = 0
(float64) 3.000000 != 2.000000 = 1

Unary operations on floats
(float64) -3.000000 = -3.000000
```

10.49. test-101-conditionals.bt

```
main {
    var x : int16 = 0;
    var y : int32 = if x == 0 {
        emit("x is 0\n");
        1;
    } else {
        emit("x is not 0\n");
        2;
    };
    emit("y={y}\n");
}
```

10.50. test-101-conditionals.out

```
x is 0
y=1
```

10.51. test-102-more-conditionals.bt

```
main {
    var x : int16 = 0;
    if x == 0 {
        emit("x is 0\n");
    } else {
        emit("x is not 0\n");
    };

    var y : int32 = -100;

    var sign_of_y : int32 = if y > 0 { 1; } elif y < 0 { -1; } else { 0; };
    emit("sign of {y} is {sign_of_y}\n");

    var forty_two : uint32 = 0x42;

    if forty_two == 0 {
        emit("forty_two==0\n");
    } elif forty_two == 42 {
        emit("forty_two==42\n");
    } elif forty_two == 0x42 {
        emit("forty_two==0x42\n");
    };
}
```

10.52. test-102-more-conditionals.out

```
x is 0
sign of -100 is -1
forty_two==0x42
```

10.53. test-103-match.bt

```
main {
    var x : int16 = 0;

    match x {
        0 -> { emit("x is 0\n"); }
        _ -> { emit("x is not 0\n"); }
    };

    var month : int32 = 12;

    # monthName's type is inferred
    var monthName = match month {
        1  -> { "January";   }
        2  -> { "February";  }
        3  -> { "March";     }
        4  -> { "April";     }
        5  -> { "May";       }
        6  -> { "June";      }
        7  -> { "July";      }
        8  -> { "August";    }
        9  -> { "September"; }
        10 -> { "October";   }
        11 -> { "November";  }
        12 -> { "December";  }
        _  -> { "(INVALID)"; }
    };

    emit("Month #{month} is \"{monthName}\"\n");
}
```

10.54. test-103-match.out

```
x is 0
Month #12 is "December"
```

10.55. test-104-while.bt

```
main {
    var x:int32 = 0;

    while x < 10 {
        emit("x={x}\n");
        x = x + 1;
    }

    emit("x={x}\n");
}
```

10.56. test-104-while.out

```
x=0
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=9
x=10
```

10.57. test-105-arrays.bt

```
main {
    var a:int8[] = [1, 2, 3, 4, 5];
    var l = len(a);

    emit("The array a has {l} elements\n");
}
```

10.58. test-105-arrays.out

```
The array a has 5 elements
```

10.59. test-106-for.bt

```
main {
    var one_to_five_u8:uint8[]    = [1, 2, 3, 4, 5];
    var one_to_five_u16:uint16[]  = [1, 2, 3, 4, 5];
    var one_to_five_u32:uint32[]  = [1, 2, 3, 4, 5];
    var one_to_five_u64:uint64[]  = [1, 2, 3, 4, 5];
    var one_to_five_i8:int8[]     = [-1, -2, -3, -4, -5];
    var one_to_five_i16:int16[]   = [-1, -2, -3, -4, -5];
    var one_to_five_i32:int32[]   = [-1, -2, -3, -4, -5];
    var one_to_five_i64:int64[]   = [-1, -2, -3, -4, -5];
    var one_to_five_f32:float32[] = [1.0, 2.0, 3.0, 4.0, 5.0];
    var one_to_five_f64:float64[] = [1.0, 2.0, 3.0, 4.0, 5.0];
    var one_to_five_str           = "12345";

    for i, v in one_to_five_u8  { emit("one_to_five_u8[{i}]:  {v}\n"); }
    for i, v in one_to_five_u16 { emit("one_to_five_u16[{i}]: {v}\n"); }
    for i, v in one_to_five_u32 { emit("one_to_five_u32[{i}]: {v}\n"); }
    for i, v in one_to_five_u64 { emit("one_to_five_u64[{i}]: {v}\n"); }
    for i, v in one_to_five_i8  { emit("one_to_five_i8[{i}]:  {v}\n"); }
    for i, v in one_to_five_i16 { emit("one_to_five_i16[{i}]: {v}\n"); }
    for i, v in one_to_five_i32 { emit("one_to_five_i32[{i}]: {v}\n"); }
    for i, v in one_to_five_i64 { emit("one_to_five_i64[{i}]: {v}\n"); }
    for i, v in one_to_five_f32 { emit("one_to_five_f32[{i}]: {v}\n"); }
    for i, v in one_to_five_f64 { emit("one_to_five_f64[{i}]: {v}\n"); }
    for i, v in one_to_five_str { emit("one_to_five_str[{i}]: {v}\n"); }
}
```

10.60. test-106-for.out

```
one_to_five_u8[0]:  1
one_to_five_u8[1]:  2
one_to_five_u8[2]:  3
one_to_five_u8[3]:  4
one_to_five_u8[4]:  5
one_to_five_u16[0]: 1
one_to_five_u16[1]: 2
one_to_five_u16[2]: 3
one_to_five_u16[3]: 4
one_to_five_u16[4]: 5
one_to_five_u32[0]: 1
one_to_five_u32[1]: 2
one_to_five_u32[2]: 3
one_to_five_u32[3]: 4
one_to_five_u32[4]: 5
one_to_five_u64[0]: 1
one_to_five_u64[1]: 2
one_to_five_u64[2]: 3
```

```
one_to_five_u64[3]: 4
one_to_five_u64[4]: 5
one_to_five_i8[0]:  255
one_to_five_i8[1]:  254
one_to_five_i8[2]:  253
one_to_five_i8[3]:  252
one_to_five_i8[4]:  251
one_to_five_i16[0]: 65535
one_to_five_i16[1]: 65534
one_to_five_i16[2]: 65533
one_to_five_i16[3]: 65532
one_to_five_i16[4]: 65531
one_to_five_i32[0]: -1
one_to_five_i32[1]: -2
one_to_five_i32[2]: -3
one_to_five_i32[3]: -4
one_to_five_i32[4]: -5
one_to_five_i64[0]: -1
one_to_five_i64[1]: -2
one_to_five_i64[2]: -3
one_to_five_i64[3]: -4
one_to_five_i64[4]: -5
one_to_five_f32[0]: 0.000000
one_to_five_f32[1]: 0.000000
one_to_five_f32[2]: 0.000000
one_to_five_f32[3]: 0.000000
one_to_five_f32[4]: 0.000000
one_to_five_f64[0]: 1.000000
one_to_five_f64[1]: 2.000000
one_to_five_f64[2]: 3.000000
one_to_five_f64[3]: 4.000000
one_to_five_f64[4]: 5.000000
one_to_five_str[0]: 49
one_to_five_str[1]: 50
one_to_five_str[2]: 51
one_to_five_str[3]: 52
one_to_five_str[4]: 53
```

10.61. test-107-more-conditionals-2.bt

```
main {
    if 2 > 1 {
        emit("2 > 1: true\n");
    } else {
        emit("2 > 1: false\n");
    };
}
```

10.62. test-107-more-conditionals-2.out

```
2 > 1: true
```

10.63. test-200-auto-inputs.bt

```
main {
    # Input data generated with:
     # ./gen_bin_data i8 -1 i16 -2 i32 -3 i64 -4 u8 5 u16 6 u32 7 u64 8 f32
9.0 f64 -10.0 s 11

    var i8  : int8;
    var i16 : int16;
    var i32 : int32;
    var i64 : int64;
    var u8  : uint8;
    var u16 : uint16;
    var u32 : uint32;
    var u64 : uint64;
    var f32 : float32;
    var f64 : float64;
    var s : string;

    emit(" i8={i8}\n");
    emit("i16={i16}\n");
    emit("i32={i32}\n");
    emit("i64={i64}\n");
    emit(" u8={u8}\n");
    emit("u16={u16}\n");
    emit("u32={u32}\n");
    emit("u64={u64}\n");
    emit("f32={f32}\n");
    emit("f64={f64}\n");
    emit("str={s}\n");
}
```

10.64. hexdump -C test-200-auto-inputs.in

```
00000000 ff fe ff fd ff ff ff fc  ff ff ff ff ff ff ff 05  |................|
00000010 06 00 07 00 00 00 08 00  00 00 00 00 00 00 00 00  |................|
00000020 10 41 00 00 00 00 00 00  24 c0 31 31 00           |.A......$.11.|
0000002d
```

10.65. test-200-auto-inputs.out

```
 i8=255
i16=65534
i32=-3
i64=-4
 u8=5
u16=6
u32=7
u64=8
f32=0.000000
f64=-10.000000
str=11
```

10.66. test-201-more-auto-inputs.bt

```
main {
    # Input data generated with
    # ./gen_bin_data i8 1 i8 2 i8 3 \
    #                i16 1 i16 2 i16 3 \
    #                i32 1 i32 2 i32 3 \
    #                i64 1 i64 2 i64 3

    var i8 : int8[3];
    var i16 : int16[3];
    var i32 : int32[3];
    var i64 : int64[3];

    var i8_0 = i8[0];
    var i8_1 = i8[1];
    var i8_2 = i8[2];

    var i16_0 = i16[0];
    var i16_1 = i16[1];
    var i16_2 = i16[2];

    var i32_0 = i32[0];
    var i32_1 = i32[1];
    var i32_2 = i32[2];

    var i64_0 = i64[0];
    var i64_1 = i64[1];
    var i64_2 = i64[2];

    emit("i8=[{i8_0},{i8_1},{i8_2}]\n");
    emit("i16=[{i16_0},{i16_1},{i16_2}]\n");
    emit("i32=[{i32_0},{i32_1},{i32_2}]\n");
    emit("i64=[{i64_0},{i64_1},{i64_2}]\n");
}
```

10.67. hexdump -C test-201-more-auto-inputs.in

```
00000000 01 02 03 01 00 02 00 03  00 01 00 00 00 02 00 00  |................|
00000010 00 03 00 00 00 01 00 00  00 00 00 00 00 02 00 00  |................|
00000020 00 00 00 00 00 03 00 00  00 00 00 00 00           |.............|
0000002d
```

10.68. test-201-more-auto-inputs.out

```
i8=[1,2,3]
i16=[1,2,3]
i32=[1,2,3]
i64=[1,2,3]
```

10.69. ./test-300-str.bt

```
main {
    var hello = "Hello, ";
    var world = "world!";

    var hello_world = hello + world;

    emit("\"{hello}\" + \"{world}\" = \"{hello_world}\"\n");

    var one = "1";
    var two = "2";
    var three = "3";

    var one_two_three = one + ":" + two + ":" + three;

    emit("{one_two_three}\n");

    var a = "127";
    var b = "0";
    var c = "1";
    var home = a + "." + b + "." + b + "." + c;

    emit("home={home}\n");
```

}

10.70. test-300-str.out

```
"Hello, " + "world!" = "Hello, world!"
1:2:3
home=127.0.0.1
```

10.71. test-900-gcd.bt

```
func gcd(a:uint64, b:uint64):uint64  {
    if b == 0 {
        return a;
    } else {
        return gcd(b, a % b);
    };
}

main {
    var a : uint64 = 10;
    var b : uint64 = 5;
    var r1 : uint64 = gcd(a, b);
    var r2 : uint64 = gcd(b, a);
    emit("gcd({a}, {b}) = {r1}\n");
    emit("gcd({b}, {a}) = {r2}\n");
}
```

10.72. test-900-gcd.out

```
gcd(10, 5) = 5
gcd(5, 10) = 5
```

10.73. test-901-read-characters.bt

```
# Input data generated with
# ./gen_bin_data u8 2 s "Barry" u16 10 s "Ann" u16 55

func read_character():None {
    var name : string;
    var level : uint16;

    emit("  \"{name}\": {\"level\": {level} }\n");
}

main {
    var num_chars : uint8;

    emit("{\n");
    var i : uint8 = 0;
    while i < num_chars {
        read_character();
        i = i + 1;
    }
    emit("}\n");
}
```

10.74. hexdump -C test-901-read-characters.in

```
00000000  02 42 61 72 72 79 00 0a  00 41 6e 6e 00 37 00     |.Barry...Ann.7.|
0000000f
```

10.75. test-901-read-characters.out

```
{
  "Barry": {"level": 10 }
  "Ann": {"level": 55 }
}
```

10.76. test-902-gcd-improved.bt

```
func gcd(a:uint64, b:uint64):uint64  {
    return match b {
        0 -> { a;             }
        _ -> { gcd(b, a % b); }
    };
}

main {
    var a : uint64;
    var b : uint64;
    var r1 = gcd(a, b);
    var r2 = gcd(b, a);
    emit("gcd({a}, {b}) = {r1}\n");
    emit("gcd({b}, {a}) = {r2}\n");
}
```

10.77. hexdump -C test-902-gcd-improved.in

```
00000000 0a 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00  |................|
00000010
```

10.78. test-902-gcd-improved.out

```
gcd(10, 5) = 5
gcd(5, 10) = 5
```