BitTwiddler

a language for binary data parsers

Project Proposal

Programming Languages and Translators

COMS W4115 – Fall 2018

Bruno Martins – bm2787

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.

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.

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;

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(‘}’);
}
```

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.

`float32, float64`	Floating point numbers, 32- or 64-bit wide.
`bit`	Single bit.
`string`	Single or several characters. Example: **`hello: string = "world"`**.
`Type`	A basic type or a template type.
`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.

Keywords

Keyword	Description
`parse`	The entry point of a program. Must be present exactly once.
`template`	Used to declare templates, akin to **`dict`** in Python, but smarter.
`_`	Means *self* inside a **`template`**, means *any* in **`match`**.
`func`	Declare a function.
`return`	Return early from a function.
`if, else`	Conditional execution.
`for, in`	Iteration over all items of an iterable.
`match`	Pattern matching (similar to Rusts's **`match`** operator).
`->`	**`match`** arm.
`:`	Type annotation.
`;`	End of statement.
`@`	Prevent embedding a field into a **`template`**.
`{ }`	Code block delimiter.
`#`	Comment.
`' "`	**`string`** delimiters.

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.

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.
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;
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.
map: Func< Array<*type2*>, Array<*type1*>, Function<*type2*, *type1*>>	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.
`join: Func<string, string, Array<string>>`	Concatenate strings in the second argument interspersed with the string in the first arg.

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');
}
```

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');
}
```