Mr. White - Erlang Interpreter for Whitespace

Erlang & Elixir Factory SF Bay Area 2017
Derek Brown
http://github.com/derek121/mrwhite

Overview

- The Whitespace language and example program
- Lexing and parsing Whitespace with Erlang
- Example Erlang code for processing Whitespace operations
- A more complex Whitespace example
- Slides and code at http://github.com/derek121/mrwhite

Whitespace

- Imperative, stack-based language
- Only significant characters are space, tab, and linefeed
- Operations consists of an instruction type followed by the command
- The interpreter maintains a stack of integers and a heap

Whitespace Data

- Numbers are represented in base 2
  - *Space*: 0
  - *Tab*: 1
  - *LF*: End of number
  - First digit indicates sign: *Space* for positive, *Tab* for negative
- Labels are *LF*-terminated lists of *Spaces* and *Tabs*

Whitespace Operations

Operations are a series of tokens, beginning with the *Instruction Modification Parameter (IMP)*

Whitespace Operations - IMPs

IMP	Meaning
Space	Stack Manipulation
Tab Space	Arithmetic
Tab Tab	Heap access
LF	Flow Control
Tab LF	I/O

Whitespace Operations - Stack Manipulation

Command	Parameters	Meaning
Space	Number	Push the number onto the stack
LF Space		Duplicate the top item on the stack
LF Tab		Swap the top two items on the stack
LF LF		Discard the top item on the stack
Tab Space	Number	Copy the Nth item of the stack (index given by the argument) onto the top of the stack
Tab LF	Number	“Slide” N items off the stack while keeping the top item

Whitespace Operations - Arithmetic

Commands operate on the top two items on the stack, and replace them with the result.

Command	Parameters	Meaning
Space Space		Addition
Space Tab		Subtraction
Space LF		Multiplication
Tab Space		Integer Division
Tab Tab		Modulo

Whitespace Operations - Heap Accesss

Store: the top two items on the stack are the value to store and the logical address at which to place it
Retrieve: the top of the stack is the address to retrieve. The value is is pushed on the stack.

Command	Parameters	Meaning
Space		Store
Tab		Retrieve

Whitespace Operations - Flow Control

Command	Parameters	Meaning
Space Space	Label	Mark a location in the program
Space Tab	Label	Call a subroutine
Space LF	Label	Jump unconditionally to a label
Tab Space	Label	Jump to a label if the top of the stack is zero
Tab Tab	Label	Jump to a label if the top of the stack is negative
Tab LF		End a subroutine and transfer control back to the caller
LF LF		End the program

Whitespace Operations - IO

Command	Parameters	Meaning
Space Space		Output the character at the top of the stack
Space Tab		Output the number at the top of the stack
Tab Space		Read a character and place it in the heap at the location given by the top of the stack
Tab Tab		Read a number and place it in the heap at the location givenby the top of the stack

A Basic Whitespace Program

A Basic Whitespace Program (For Humans)

S = Space. T = Tab. Newlines as Newlines.

```
SSSS
SSSTSSSST
SSSTTSSTST
SSSTTSTTTT
SSSTSSTSTS
SSSTSSSSS
SSSTTSTTTT
SSSTTSTTSS
SSSTTSTTSS
SSSTTSSTST
SSSTSSTSSS

SSS
T
SSS
S
TSST

S
S

SSST
SSSTSTS
T
SS
```

A Basic Whitespace Program
(For Humans Who Aren't Robots)

```
$ cat sample.wst
stack push 0
stack push 33
stack push 101
stack push 111
stack push 74
stack push 32
stack push 111
stack push 108
stack push 108
stack push 101
stack push 72

flow mark s
io output_char
stack duplicate
flow jump_zero st
flow jump s

flow mark st
stack push 10
io output_char
flow end_program
```

A Basic Whitespace Program

Output

```
1> mrwhite_from_text:run({file_in, "sample.wst"}).
```

A Basic Whitespace Program

Output

```
1> mrwhite_from_text:run({file_in, "sample.wst"}).
Hello Joe!
```

A Basic Whitespace Program

Output

```
1> mrwhite_from_text:run({file_in, "sample.wst"}).
Hello Joe!
```

Lexing and Parsing

- leex: A regular expression based lexical analyzer generator
  - E.g., translate a space, a tab, and a space to a token representing *stack copy*
- yecc: Parser generator
  - E.g., recognize *stack copy* followed by a *number* as a complete stack copy operation

Lexing

We use `leex`, analagous to the standard Unix tool `lex`

```
1> leex:file("lex_spec").
```

Processes `lex_spec.xrl`, and outputs `lex_spec.erl`

We can then call `lex_spec:string/1` with our program input.

Its output is then used in the parsing step.

Parsing

We use `yecc`, analagous to the standard Unix tool `yacc`

```
1> yecc:file("parse_spec").
```

Processes `parse_spec.xrl`, and outputs `parse_spec.erl`

We can then call `parse_spec:parse/1` with the output from the previous leex step.

Example output:

```
[{stack_push, 23}, stack_duplicate, flow_end_program]
```

leex Specification for Whitespace (1/5)

```
Definitions.

S = \s
T = \t
L = \n

STACK_PUSH          = {S}{S}
STACK_DUP           = {S}{L}{S}
STACK_COPY          = {S}{T}{S}
STACK_SWAP          = {S}{L}{T}
STACK_DISCARD       = {S}{L}{L}
STACK_SLIDE         = {S}{T}{L}

ARITH_ADD           = {T}{S}{S}{S}
ARITH_SUB           = {T}{S}{S}{T}
ARITH_MUL           = {T}{S}{S}{L}
ARITH_DIV           = {T}{S}{T}{S}
ARITH_MOD           = {T}{S}{T}{T}

HEAP_STORE          = {T}{T}{S}
HEAP_RETRIEVE       = {T}{T}{T}
```

leex Specification for Whitespace (2/5)

```
FLOW_MARK           = {L}{S}{S}
FLOW_CALL           = {L}{S}{T}
FLOW_JUMP           = {L}{S}{L}
FLOW_JUMP_ZERO      = {L}{T}{S}
FLOW_JUMP_NEGATIVE  = {L}{T}{T}
FLOW_END_SUB        = {L}{T}{L}
FLOW_END            = {L}{L}{L}

IO_OUTPUT_CHAR      = {T}{L}{S}{S}
IO_OUTPUT_NUM       = {T}{L}{S}{T}
IO_READ_CHAR        = {T}{L}{T}{S}
IO_READ_NUM         = {T}{L}{T}{T}

NUM                 = [{S}{T}]+{L}
LABEL               = [{S}{T}]+{L}
```

leex Specification for Whitespace (3/5)

```
Rules.

{STACK_PUSH}{NUM}          : {token, {stack_push,         TokenLine}, "N" ++ extract_number_or_label(3, TokenChars)}.
{STACK_DUP}                : {token, {stack_duplicate,    TokenLine}}.
{STACK_COPY}{NUM}          : {token, {stack_copy,         TokenLine}, "N" ++ extract_number_or_label(4, TokenChars)}.
{STACK_SWAP}               : {token, {stack_swap,         TokenLine}}.
{STACK_DISCARD}            : {token, {stack_discard,      TokenLine}}.
{STACK_SLIDE}{NUM}         : {token, {stack_slide,        TokenLine}, "N" ++ extract_number_or_label(4, TokenChars)}.

{ARITH_ADD}                : {token, {arith_add,          TokenLine}}.
{ARITH_SUB}                : {token, {arith_sub,          TokenLine}}.
{ARITH_MUL}                : {token, {arith_mul,          TokenLine}}.
{ARITH_DIV}                : {token, {arith_div,          TokenLine}}.
{ARITH_MOD}                : {token, {arith_mod,          TokenLine}}.

{HEAP_STORE}               : {token, {heap_store,         TokenLine}}.
{HEAP_RETRIEVE}            : {token, {heap_retrieve,      TokenLine}}.
```

leex Specification for Whitespace (4/5)

```
{FLOW_MARK}{LABEL}          : {token, {flow_mark,          TokenLine}, "L" ++ extract_number_or_label(4, TokenChars)}.
{FLOW_CALL}{LABEL}          : {token, {flow_call,          TokenLine}, "L" ++ extract_number_or_label(4, TokenChars)}.
{FLOW_JUMP}{LABEL}          : {token, {flow_jump,          TokenLine}, "L" ++ extract_number_or_label(4, TokenChars)}.
{FLOW_JUMP_ZERO}{LABEL}     : {token, {flow_jump_zero,     TokenLine}, "L" ++ extract_number_or_label(4, TokenChars)}.
{FLOW_JUMP_NEGATIVE}{LABEL} : {token, {flow_jump_negative, TokenLine}, "L" ++ extract_number_or_label(4, TokenChars)}.
{FLOW_END_SUB}              : {token, {flow_end_sub,       TokenLine}}.
{FLOW_END}                  : {token, {flow_end_program,   TokenLine}}.

{IO_OUTPUT_CHAR}            : {token, {io_output_char,     TokenLine}}.
{IO_OUTPUT_NUM}             : {token, {io_output_num,      TokenLine}}.
{IO_READ_CHAR}              : {token, {io_read_char,       TokenLine}}.
{IO_READ_NUM}               : {token, {io_read_num,        TokenLine}}.

N[{S}{T}]+{L}               : {token, {ws_number, TokenLine, tl(TokenChars)}}.
L[{S}{T}]+{L}               : {token, {ws_label,  TokenLine, tl(TokenChars)}}.
```

leex Specification for Whitespace (5/5)

```
Erlang code.

extract_number_or_label(StartIdx, TokenChars) ->
  lists:nthtail(StartIdx - 1, TokenChars).
```

yecc Specification for Whitespace (1/4)

```
Nonterminals ops elements op.

Terminals stack_push stack_duplicate stack_copy stack_swap stack_discard stack_slide
          arith_add arith_sub arith_mul arith_div arith_mod
          heap_store heap_retrieve
          flow_mark flow_call flow_jump flow_jump_zero flow_jump_negative flow_end_sub flow_end_program
          io_output_char io_output_num io_read_char io_read_num
          ws_number ws_label.
```

yecc Specification for Whitespace (2/4)

```
Rootsymbol elements.

elements -> ops : '$1'.

ops -> op : ['$1'].
ops -> op ops : ['$1'] ++ '$2'.

op -> stack_push ws_number        : {stack_push, parse_num(unwrap('$2'))}.
op -> stack_duplicate             : stack_duplicate.
op -> stack_copy ws_number        : {stack_copy, parse_num(unwrap('$2'))}.
op -> stack_swap                  : stack_swap.
op -> stack_discard               : stack_discard.
op -> stack_slide ws_number       : {stack_slide, parse_num(unwrap('$2'))}.

op -> arith_add                   : arith_add.
op -> arith_sub                   : arith_sub.
op -> arith_mul                   : arith_mul.
op -> arith_div                   : arith_div.
op -> arith_mod                   : arith_mod.

op -> heap_store                  : heap_store.
op -> heap_retrieve               : heap_retrieve.
```

yecc Specification for Whitespace (3/4)

```
op -> flow_mark ws_label          : {flow_mark, parse_label(unwrap('$2'), [])}.
op -> flow_call ws_label          : {flow_call, parse_label(unwrap('$2'), [])}.
op -> flow_jump ws_label          : {flow_jump, parse_label(unwrap('$2'), [])}.
op -> flow_jump_zero ws_label     : {flow_jump_zero, parse_label(unwrap('$2'), [])}.
op -> flow_jump_negative ws_label : {flow_jump_negative, parse_label(unwrap('$2'), [])}.
op -> flow_end_sub                : flow_end_sub.
op -> flow_end_program            : flow_end_program.

op -> io_output_char              : io_output_char.
op -> io_output_num               : io_output_num.
op -> io_read_char                : io_read_char.
op -> io_read_num                 : io_read_num.
```

yecc Specification for Whitespace (4/4)

```
Erlang code.

-define(S,  32).
-define(T,  9).
-define(LF, 10).

unwrap({_,_,V}) -> V.

parse_num([?S | Rest]) ->
  parse_num(1, Rest, []);
parse_num([?T | Rest]) ->
  parse_num(-1, Rest, []).

parse_num(Multiplier, [?LF], Acc) ->
  Rev = lists:reverse(Acc),
  Multiplier * erlang:list_to_integer(Rev, 2);
parse_num(Multiplier, [?S | Rest], Acc) ->
  parse_num(Multiplier, Rest, [$0 | Acc]);
parse_num(Multiplier, [?T | Rest], Acc) ->
  parse_num(Multiplier, Rest, [$1 | Acc]).

parse_label([?LF], Acc) ->
  lists:reverse(Acc);
parse_label([?S | Rest], Acc) ->
  parse_label(Rest, [$s | Acc]);
parse_label([?T | Rest], Acc) ->
  parse_label(Rest, [$t | Acc]).
```

Functionality of the mrwhite Application

- Execute Whitespace Programs
- Execute Whitespace-as-Text Programs
- Convert Whitespace to Whitespace-as-Text
- Convert Whitespace-as-Text to Whitespace

Execution Steps

- Extract labels (for jumps) into map: `#{Key => ListIndex}`
- Execute remaining operations
  - Maintain stack of integers (as a list)
  - Maintain heap (as a map)
  - Follow arbitrary jump/subroutine calls within the list of operations

Sample Operation Implementation: Stack Push

```
run([{stack_push, N} | Rest], Stack, Heap, Labels, NumSubCalls, AllOps) ->
  run(Rest, [N | Stack], Heap, Labels, NumSubCalls, AllOps);
```

Sample Operation Implementation: Arith Add

```
run([arith_add | Rest], Stack = [First, Second | StackRest], Heap, Labels, NumSubCalls, AllOps)
  when length(Stack) >= 2 ->
  run(Rest, [First + Second | StackRest], Heap, Labels, NumSubCalls, AllOps);
```

Sample Operation Implementation: Heap Store

```
run([heap_store | Rest], Stack = [Value, Address | StackRest], Heap, Labels, NumSubCalls, AllOps)
  when length(Stack) >= 2 ->
  run(Rest, StackRest, Heap#{Address => Value}, Labels, NumSubCalls, AllOps);
```

Sample Operation Implementation: Flow Call

```
run([{flow_call, Label} | Rest], Stack, Heap, Labels, NumSubCalls, AllOps) ->
  case maps:get(Label, Labels, undefined) of
    undefined ->
      exit({unknown_label_for_call_sub, Label});
    N         ->
      SubOps = lists:nthtail(N, AllOps),
      {sub_done, SubStack, SubHeap, Labels} = run(SubOps, Stack, Heap, Labels, NumSubCalls + 1, AllOps),
      run(Rest, SubStack, SubHeap, Labels, NumSubCalls, AllOps)
  end;
```

Sample Operation Implementation: IO Read Number

```
run([io_read_num | Rest], [Top | StackRest], Heap, Labels, NumSubCalls, AllOps) ->
  N = erlang:list_to_integer(lists:droplast(io:get_line(""))),
  run(Rest, StackRest, Heap#{Top => N}, Labels, NumSubCalls, AllOps).
```

The Canonical Demo Program

As seen in the original Whitespace specification.

Operation	Description
`stack push 1`	Put a 1 on the stack
`flow mark stssssstt`	Set a Label at this point
`stack duplicate`	Duplicate the top stack item
`io output_num`	Output the current value
`stack push 10`	Put 10 (newline) on the stack...
`io output_char`	...and output the newline
`stack push 1`	Put a 1 on the stack
`arith add Addition`	This increments our current value.
`stack duplicate`	Duplicate that value so we can test it
`stack push 11`	Push 11 onto the stack
`arith sub Subtraction`	So if we've reached the end, we have a zero on the stack.
`flow jump_zero stsssstst`	If we have a zero, jump to the end
`flow jump stssssstt`	Jump to the start
`flow mark stsssstst`	Set the end label
`stack discard`	Discard our accumulator, to be tidy
`flow end_program`	Finish

The Canonical Demo Program - Execution

```
1> mrwhite_from_text:run({file_in, "demo.wst"}).
1
2
3
4
5
6
7
8
9
10
ok
2>
```

References

Mr. White on Github:
http://github.com/derek121/mrwhite

Whitespace:
http://compsoc.dur.ac.uk/whitespace/tutorial.html
https://en.wikipedia.org/wiki/Whitespace_(programming_language)

leex and yecc:
http://erlang.org/doc/man/leex.html
http://erlang.org/doc/man/yecc.html
http://andrealeopardi.com/posts/tokenizing-and-parsing-in-elixir-using-leex-and-yecc/
https://arifishaq.wordpress.com/2014/01/22/playing-with-leex-and-yeec/
https://cameronp.svbtle.com/how-to-use-leex-and-yecc
http://relops.com/blog/2014/01/13/leex_and_yecc/


YO
MR. WHITE