P0 WebAssembly Code Generator Tests

Emil Sekerinski, McMaster University, revised February 2022

In [ ]:
```
import nbimporter; nbimporter.options["only_defs"] = False
from P0 import compileString
```

The following modified standard library provides a constant result when calling P0 `read()` rather than taking it from interactive input in order to allow automated testing. P0 `read()` is only called in the section *Input & Output*.

In [ ]:
```
def runpywasm(wasmfile):
    import pywasm
    def write(s, i): print(i)
    def writeln(s): print()
    def read(s): return 5
    vm = pywasm.load(wasmfile, {'P0lib': {'write': write, 'writeln': writeln, 'read': read}})
```

As `pywasm` does not support functions with multiple returns and bulk memory operations, but Chrome does, some tests use the web browser's implementation of WebAssembly instead:

In [ ]:
```
def runwasm(wasmfile):
    from IPython.core.display import display, Javascript
    display(Javascript("""
    const params = {
        P0lib: {
            write: i => this.append_stream({text: '' + i, name: 'stdout'}),
            writeln: () => this.append_stream({text: '\\n', name: 'stdout'}),
            read: () => window.prompt()
        }
    }
    fetch('""" + wasmfile + """') // asynchronously fetch file, return Response object
      .then(response => response.arrayBuffer()) // read the response to completion and stores it in an ArrayBuf
      .then(code => WebAssembly.compile(code)) // compile (sharable) code.wasm
      .then(module => WebAssembly.instantiate(module, params)) // create an instance with memory
    // .then(instance => instance.exports.program()); // run the main program; not needed if start function spe
    """))
```

Procedure `compileerr(s)` returns an empty string if compiling `s` with the WebAssembly code generator succeeds or the error message produced while compiling; the error message is also printed. The procedure is used here to test code generation.

In [ ]:
```
def compileerr(s):
    try: compileString(s, target = 'wat'); return ''
    except Exception as e:
        print(e); return str(e)
```

Experimental: method calls (ignore)

In [ ]:
```
# compileString("""
# type A = [2 .. 9] → integer
# var a: A
# procedure (r: A) q()
#   r[2] := 7
# program p
#   writeln() //a.q()
# """)#, 'assign.wat', target = 'wat')
```

In [ ]:
```
# !wat2wasm assign.wat
```

In [ ]:
```
# runpywasm('assign.wasm')
```

Error: "WASM: no nested procedures"

In [ ]:
```
assert "WASM: no nested procedures" in compileerr("""
program p
  procedure q()
    write(5)
  q()
""")
```

Error: "WASM: set too large"

In [ ]:
```
assert "WASM: set too large" in compileerr("""
var s: set [0..100]
program p
  writeln()
""")
```

In [ ]:
```
print(compileString("""
var s: set [0..10]
program p
  writeln()
"""))
```

The subsequent tests write the generated code to a textual `.wat` file first. That file is then read into a variable, `asm`, which is then compared with the expected code. The generated `.wat` file is then converted with `wat2wasm` to a binary `.wasm` file. That file is then executed with `runpywasm`. The output of execution is captured in the variable `out`, which is then compared with the expected output.

Assignment

In [ ]:
```
compileString("""
var a: [2 .. 9] → integer
program p
  var x, y: integer
    a[3] := 5
    x, y := a[3], 7
    x, y := y, x
    write(x); write(y) // writes 7, 5
""", 'assign.wat', target = 'wat')

with open('assign.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $program
(local $x i32)
(local $y i32)
(local $0 i32)
i32.const 4
i32.const 5
i32.store
i32.const 4
i32.load
i32.const 7
local.set $y
local.set $x
local.get $y
local.get $x
local.set $y
local.set $x
local.get $x
call $write
local.get $y
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm assign.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('assign.wasm')
```

In [ ]:
```
assert str(out) == """\
7
5
"""
```

Relational Operators

In [ ]:
```
compileString("""
procedure q(b: boolean)
  b := b = false
program p
  var x: integer
    q(x > 7)
""", 'relop.wat', target = 'wat')

with open('relop.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $q (param $b i32)
(local $0 i32)
local.get $b
i32.const 0
i32.eq
local.set $b
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $x i32)
(local $0 i32)
local.get $x
i32.const 7
i32.gt_s
call $q
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm relop.wat # validating generated code by translating to binary format
```

Input & Output

In [ ]:
```
compileString("""
program p
  var x: integer
    x ← read(); x := 3 × x
    write(x); writeln()
    write(x × 5)
""", 'io.wat', target = 'wat')

with open('io.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $x i32)
(local $0 i32)
call $read
local.set $x
i32.const 3
local.get $x
i32.mul
local.set $x
local.get $x
call $write
call $writeln
local.get $x
i32.const 5
i32.mul
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm io.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('io.wasm')
```

In [ ]:
```
assert str(out) == """\
15

75
"""
```

Parameter Passing

In [ ]:
```
compileString("""
type T = [1..10] → integer
var a: T
procedure q(b: integer, c: integer)
```

```
    write(b); write(c)
procedure r() → (d: integer)
    a[3] := 9; d := 5
program p
  var x: integer
  a[2] := 7; q(3, a[2]) // writes 3, 7
  x ← r(); write(x); write(a[3]) // writes 5, 9
""", 'params.wat', target = 'wat')

with open('params.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $q (param $b i32) (param $c i32)
(local $0 i32)
local.get $b
call $write
local.get $c
call $write
)
(func $r  (result i32)
(local $d i32)
(local $0 i32)
i32.const 8
i32.const 9
i32.store
i32.const 5
local.set $d
local.get $d
)
(global $_memsize (mut i32) i32.const 40)
(func $program
(local $x i32)
(local $0 i32)
i32.const 4
i32.const 7
i32.store
i32.const 3
i32.const 4
i32.load
call $q
call $r
local.set $x
local.get $x
call $write
i32.const 8
i32.load
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm params.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('params.wasm')
```

In [ ]:
```
assert str(out) == """\
3
7
5
9
"""
```

Multiple Result Parameters

In [ ]:
```
compileString("""
procedure swap(x0, y0: integer) → (x1, y1: integer)
    x1, y1 := y0, x0
program p
  var x, y: integer
    x, y ← swap(5, 7)
    write(x); write(y) // writes 7, 5
""", 'multipleassign.wat', target = 'wat')

with open('multipleassign.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $swap (param $x0 i32) (param $y0 i32) (result i32) (result i32)
(local $x1 i32)
(local $y1 i32)
(local $0 i32)
local.get $y0
local.get $x0
local.set $y1
local.set $x1
local.get $x1
local.get $y1
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $x i32)
(local $y i32)
(local $0 i32)
i32.const 5
i32.const 7
call $swap
local.set $y
local.set $x
local.get $x
call $write
local.get $y
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm multipleassign.wat # validating generated code by translating to binary format
```

This should print `7 5` on separate lines. As the `capture` cell magic does not work with `runwasm`, this test is not automated.

In [ ]:
```
runwasm('multipleassign.wasm')
```

Arrays and Records

In [ ]:
```
compileString("""
type A = [1 .. 7] → integer
type R = (f: integer, g: A, h: integer)
var v: A
var w: R
var x: integer
program p
  x := 9;
  w.h := 12 - 7; write(w.h) // writes 5
  v[1] := 3; write(v[x - 8]) //writes 3
  w.g[x div 3] := 9; write(w.g[3]) // writes 9
  v[x - 2] := 7; w.g[x - 3] := 7
  write(v[7]); write(w.g[6]) // writes 7, 7
""", 'arrayrec.wat', target = 'wat')

with open('arrayrec.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $x (mut i32) i32.const 0)
(global $_memsize (mut i32) i32.const 64)
(func $program
(local $0 i32)
i32.const 9
global.set $x
i32.const 60
i32.const 5
i32.store
i32.const 60
i32.load
call $write
i32.const 0
i32.const 3
i32.store
global.get $x
i32.const 8
i32.sub
i32.const 1
i32.sub
i32.const 4
i32.mul
i32.const 0
i32.add
i32.load
call $write
global.get $x
i32.const 3
i32.div_s
i32.const 1
i32.sub
i32.const 4
i32.mul
i32.const 32
i32.add
i32.const 9
i32.store
i32.const 40
i32.load
call $write
global.get $x
i32.const 2
i32.sub
i32.const 1
i32.sub
i32.const 4
i32.mul
i32.const 0
i32.add
i32.const 7
i32.store
global.get $x
i32.const 3
i32.sub
i32.const 1
i32.sub
i32.const 4
i32.mul
i32.const 32
i32.add
i32.const 7
i32.store
i32.const 24
i32.load
call $write
i32.const 52
i32.load
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm arrayrec.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('arrayrec.wasm')
```

In [ ]:
```
assert str(out) == """\
5
3
9
7
7
"""
```

Array Assignment

Following tests copy arrays and records. P0 generates `memory.copy` instructions, which are not supported by pywasm, but are supported by Chrome. For conversion of textual to binary WebAssembly, `wat2wasm` needs the `enable-bulk-memory` flag.

In [ ]:
```
compileString("""
var c: [0 .. 1] → integer
var a, b: [2 .. 9] → integer
program p
  b[2] := 3; b[3] := 5
  a := b
  write(a[2]); write(a[3]); write(a[4]) // writes 3, 5, 0
""", 'arrayassignment.wat', target = 'wat')

with open('arrayassignment.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $_memsize (mut i32) i32.const 72)
(func $program
(local $0 i32)
i32.const 40
i32.const 3
i32.store
i32.const 44
i32.const 5
i32.store
i32.const 8
i32.const 40
i32.const 32
memory.copy
i32.const 8
i32.load
call $write
i32.const 12
i32.load
call $write
i32.const 16
i32.load
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm --enable-bulk-memory arrayassignment.wat
```

This should print `3 5 0` on separate lines. As the `capture` cell magic does not work with `runwasm`, this test is not automated.

In [ ]:
```
runwasm('arrayassignment.wasm')
```

Array Value and Result Parameters

In [ ]:
```
compileString("""
type A = [2 .. 9] → integer
type B = [0 .. 1] → A
var b: B
procedure q(x: A) → (y: A)
    y := x; write(x[4]) // writes 0
program p
  b[1][2] := 3; b[1][3] := 5
  b[0] ← q(b[1])
  write(b[0][2]); write(b[0][3]) // writes 3, 5
""", 'arrayvalueresult.wat', target = 'wat')

with open('arrayvalueresult.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $q (param $x i32) (result i32)
(local $y i32)
(local $0 i32)
(local $_fp i32)
global.get $_memsize
local.set $_fp
global.get $_memsize
i32.const 32
i32.add
local.tee $y
global.set $_memsize
local.get $y
local.get $x
i32.const 32
memory.copy
i32.const 4
i32.const 2
i32.sub
i32.const 4
i32.mul
local.get $x
i32.add
i32.load
call $write
local.get $y
)
(global $_memsize (mut i32) i32.const 64)
(func $program
(local $0 i32)
i32.const 32
i32.const 3
i32.store
i32.const 36
i32.const 5
i32.store
i32.const 0
i32.const 32
call $q
i32.const 32
memory.copy
i32.const 0
i32.load
call $write
i32.const 4
i32.load
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm --enable-bulk-memory arrayvalueresult.wat
```

This should print `0 3 5` on separate lines. As the `capture` cell magic does not work with `runwasm`, this test is not automated.

In [ ]:
```
runwasm('arrayvalueresult.wasm')
```

Local Array

In [ ]:
```
compileString("""
type A = [2 .. 9] → integer
type B = [0 .. 1] → A
procedure q(x: A)
  var y: A
    y := x
    write(x[2]); write(x[3]); write(x[4]) // writes 3, 5, 0
program p
  var b: B
    b[1][2] := 3; b[1][3] := 5
    q(b[1])
""", 'localarray.wat', target = 'wat')

with open('localarray.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $q (param $x i32)
(local $y i32)
(local $0 i32)
(local $_fp i32)
global.get $_memsize
local.set $_fp
global.get $_memsize
i32.const 32
i32.add
local.tee $y
global.set $_memsize
local.get $y
local.get $x
i32.const 32
memory.copy
i32.const 2
i32.const 2
i32.sub
i32.const 4
i32.mul
local.get $x
i32.add
i32.load
call $write
i32.const 3
i32.const 2
i32.sub
i32.const 4
i32.mul
local.get $x
i32.add
i32.load
call $write
i32.const 4
i32.const 2
i32.sub
i32.const 4
i32.mul
local.get $x
i32.add
i32.load
call $write
local.get $_fp
global.set $_memsize
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $b i32)
(local $0 i32)
(local $_fp i32)
global.get $_memsize
local.set $_fp
global.get $_memsize
i32.const 64
i32.add
local.tee $b
global.set $_memsize
i32.const 1
i32.const 32
i32.mul
local.get $b
i32.add
i32.const 2
i32.const 2
i32.sub
i32.const 4
i32.mul
i32.add
i32.const 3
i32.store
i32.const 1
i32.const 32
i32.mul
local.get $b
i32.add
i32.const 3
i32.const 2
i32.sub
i32.const 4
i32.mul
i32.add
i32.const 5
i32.store
i32.const 1
i32.const 32
i32.mul
local.get $b
i32.add
call $q
local.get $_fp
global.set $_memsize
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm --enable-bulk-memory localarray.wat
```

This should print `3 5 0` on separate lines. As the `capture` cell magic does not work with `runwasm`, this test is not automated.

In [ ]:
```
runwasm('localarray.wasm')
```

Two-dimensional Array

In [ ]:
```
compileString("""
type R = boolean
type S = [1..11] → R
type T = [3..9] → S
var x: T
var y: integer
var b: boolean
program p
  x[y][5] := false
  b := x[y][y + 1]
""", 'twoD.wat', target = 'wat')

with open('twoD.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $y (mut i32) i32.const 0)
(global $b (mut i32) i32.const 0)
(global $_memsize (mut i32) i32.const 77)
(func $program
(local $0 i32)
global.get $y
i32.const 3
i32.sub
i32.const 11
i32.mul
i32.const 0
i32.add
i32.const 5
i32.const 1
i32.sub
i32.const 1
i32.mul
i32.add
i32.const 0
i32.store
global.get $y
i32.const 3
i32.sub
i32.const 11
i32.mul
i32.const 0
i32.add
global.get $y
i32.const 1
i32.add
i32.const 1
i32.sub
i32.const 1
i32.mul
i32.add
i32.load
global.set $b
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm twoD.wat # validating generated code by translating to binary format
```

Sets

In [ ]:
```
compileString("""
type S = set [1..10]
procedure elements(s: S)
  var i: integer
    writeln(); i := 0
    while i < 32 do
      if i ∈ s then write(i)
      i := i + 1
procedure difference(s: S, t: S) → (u: S)
  u := s ∩ Ct
program p
  var s: S
    s := {3}; elements(s) // writes 3
    s := s ∪ {1, 9}; elements(s) // writes 1, 3, 9
    s := Cs; elements(s) // writes 2, 4, 5, 6, 7, 8, 10
    s := s ∩ {5, 7, 9}; elements(s) // writes 5, 7
    s ← difference(s, {7, 8, 9}); elements(s) // writes 5
    writeln(); if s ⊆ {2, 5, 20} then write(#s) // writes 1
    if {2, 5} ⊆ s then write(-1) else write(-2) // writes -2

""", 'sets.wat', target = 'wat')

with open('sets.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(func $elements (param $s i32)
(local $i i32)
(local $0 i32)
call $writeln
i32.const 0
local.set $i
loop
local.get $i
i32.const 32
i32.lt_s
if
local.get $i
local.set $0
i32.const 1
local.get $0
i32.shl
local.get $s
i32.and
if
local.get $i
call $write
end
local.get $i
i32.const 1
i32.add
local.set $i
br 1
end
end
)
(func $difference (param $s i32) (param $t i32) (result i32)
(local $u i32)
(local $0 i32)
local.get $t
i32.const 0x7fe
i32.xor
local.get $s
i32.and
local.set $u
local.get $u
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $s i32)
(local $0 i32)
i32.const 3
local.set $0
i32.const 1
local.get $0
i32.shl
local.set $s
local.get $s
call $elements
i32.const 1
local.set $0
i32.const 1
local.get $0
i32.shl
i32.const 9
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
local.get $s
i32.or
local.set $s
local.get $s
call $elements
local.get $s
i32.const 0x7fe
i32.xor
local.set $s
local.get $s
call $elements
i32.const 5
local.set $0
i32.const 1
local.get $0
i32.shl
```

```
i32.const 7
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
i32.const 9
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
local.get $s
i32.and
local.set $s
local.get $s
call $elements
local.get $s
i32.const 7
local.set $0
i32.const 1
local.get $0
i32.shl
i32.const 8
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
i32.const 9
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
call $difference
local.set $s
local.get $s
call $elements
call $writeln
local.get $s
local.tee $0
local.get $0
i32.const 2
local.set $0
i32.const 1
local.get $0
i32.shl
i32.const 5
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
i32.const 20
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
i32.and
i32.eq
if
local.get $s
i32.popcnt
call $write
end
i32.const 2
local.set $0
i32.const 1
local.get $0
i32.shl
i32.const 5
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
local.tee $0
local.get $0
local.get $s
i32.and
i32.eq
```

```
if
i32.const -1
call $write
else
i32.const -2
call $write
end
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm sets.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('sets.wasm')
```

In [ ]:
```
assert str(out) == """\

3

1
3
9

2
4
5
6
7
8
10

5
7

5

1
-2
"""
```

Booleans and Conditions

In [ ]:
```
compileString("""
program p
  const five = 5
  const seven = 7
  const always = true
  const never = false
  var x, y, z: integer
  var b, t, f: boolean
    x := seven; y := 9; z := 11; t := true; f := false
    if true then write(7) else write(9)    // writes 7
    if false then write(7) else write(9)   // writes 9
    if t then write(7) else write(9)       // writes 7
    if f then write(7) else write(9)       // writes 9
    if ¬ t then write(7) else write(9)     // writes 9
    if ¬ f then write(7) else write(9)     // writes 7
    if t or t then write(7) else write(9)  // writes 7
    if t or f then write(7) else write(9)  // writes 7
    if f or t then write(7) else write(9)  // writes 7
    if f or f then write(7) else write(9)  // writes 9
    if t and t then write(7) else write(9) // writes 7
    if t and f then write(7) else write(9) // writes 9
    if f and t then write(7) else write(9) // writes 9
    if f and f then write(7) else write(9) // writes 9
    writeln()
    b := true
    if b then write(3) else write(5) // writes 3
    b := false
    if b then write(3) else write(5) // writes 5
    b := x < y
    if b then write(x) else write(y) // writes 7
    b := (x > y) or t
    if b then write(3) else write(5) // writes 3
    b := (x > y) or f
    if b then write(3) else write(5) // writes 5
    b := (x = y) or (x > y)
    if b then write(3) else write(5) // writes 5
    b := (x = y) or (x < y)
    if b then write(3) else write(5) // writes 3
    b := f and (x ≥ y)
    if b then write(3) else write(5) // writes 5
    writeln()
    while y > 3 do                   // writes 9, 8, 7, 6, 5, 4
      write(y); y := y - 1
    write(y); writeln()              // writes 3
    if ¬(x < y) and t then
      write(x)                       // writes 7
""", 'cond.wat', target = 'wat')

with open('cond.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $x i32)
(local $y i32)
(local $z i32)
(local $b i32)
(local $t i32)
(local $f i32)
(local $0 i32)
i32.const 7
local.set $x
i32.const 9
local.set $y
i32.const 11
local.set $z
i32.const 1
local.set $t
i32.const 0
local.set $f
i32.const 1
if
i32.const 7
call $write
else
i32.const 9
call $write
end
i32.const 0
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $t
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $t
i32.eqz
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
i32.eqz
if
i32.const 7
call $write
else
i32.const 9
```

```
call $write
end
local.get $t
if (result i32)
i32.const 1
else
local.get $t
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $t
if (result i32)
i32.const 1
else
local.get $f
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
if (result i32)
i32.const 1
else
local.get $t
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
if (result i32)
i32.const 1
else
local.get $f
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $t
if (result i32)
local.get $t
else
i32.const 0
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $t
if (result i32)
local.get $f
else
i32.const 0
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
if (result i32)
local.get $t
```

```
else
i32.const 0
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
local.get $f
if (result i32)
local.get $f
else
i32.const 0
end
if
i32.const 7
call $write
else
i32.const 9
call $write
end
call $writeln
i32.const 1
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
i32.const 0
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
local.get $x
local.get $y
i32.lt_s
local.set $b
local.get $b
if
local.get $x
call $write
else
local.get $y
call $write
end
local.get $x
local.get $y
i32.gt_s
if (result i32)
i32.const 1
else
local.get $t
end
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
local.get $x
local.get $y
i32.gt_s
if (result i32)
i32.const 1
else
local.get $f
end
local.set $b
local.get $b
```

```
if
i32.const 3
call $write
else
i32.const 5
call $write
end
local.get $x
local.get $y
i32.eq
if (result i32)
i32.const 1
else
local.get $x
local.get $y
i32.gt_s
end
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
local.get $x
local.get $y
i32.eq
if (result i32)
i32.const 1
else
local.get $x
local.get $y
i32.lt_s
end
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
local.get $f
if (result i32)
local.get $x
local.get $y
i32.ge_s
else
i32.const 0
end
local.set $b
local.get $b
if
i32.const 3
call $write
else
i32.const 5
call $write
end
call $writeln
loop
local.get $y
i32.const 3
i32.gt_s
if
local.get $y
call $write
local.get $y
i32.const 1
i32.sub
local.set $y
br 1
end
end
local.get $y
call $write
call $writeln
local.get $x
local.get $y
i32.lt_s
```

```
i32.eqz
if (result i32)
local.get $t
else
i32.const 0
end
if
local.get $x
call $write
end
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm cond.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('cond.wasm')
```

In [ ]:
```
assert str(out) == """\
7
9
7
9
9
7
7
7
7
9
7
9
9
9

3
5
7
3
5
5
3
5

9
8
7
6
5
4
3

7
"""
```

Constant Folding, Local & Global Variables

In [ ]:
```
compileString("""
const seven = (9 mod 3 + 5 × 3) div 2
type int = integer
var x, y: integer
procedure q()
  const sotrue = true and true
  const sofalse = false and true
  const alsotrue = false or true
  const alsofalse = false or false
  var x: int
    x := 3
    if sotrue then y := x else y := seven
    write(y) // writes 3
    if sofalse then y := x else y := seven
    write(y) // writes 7
    if alsotrue then y := x else y := seven
    write(y) // writes 3
    if alsofalse then y := x else y := seven
    write(y) // writes 7
    if ¬(true or false) then write(5) else write(9) // writes 9
program p
  x := 7; q(); write(x) // writes 7
""", 'folding.wat', target = 'wat')
```

```
with open('folding.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $x (mut i32) i32.const 0)
(global $y (mut i32) i32.const 0)
(func $q
(local $x i32)
(local $0 i32)
i32.const 3
local.set $x
i32.const 1
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 1
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
i32.const 5
call $write
else
i32.const 9
call $write
end
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $0 i32)
i32.const 7
global.set $x
call $q
global.get $x
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm folding.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('folding.wasm')
```

In [ ]:
```
assert str(out) == """\
3
7
```

```
3
7
9
7
"""
```

Procedures

In [ ]:
```
compileString("""
var g: integer          // global variable
procedure q(v: integer) // value parameter
  var l: integer        // local variable
    l := 9
    if l > v then write(l)
    else write(g)
program p
  g := 5; q(7)
""", 'proc.wat', target = 'wat')

with open('proc.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $g (mut i32) i32.const 0)
(func $q (param $v i32)
(local $l i32)
(local $0 i32)
i32.const 9
local.set $l
local.get $l
local.get $v
i32.gt_s
if
local.get $l
call $write
else
global.get $g
call $write
end
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $0 i32)
i32.const 5
global.set $g
i32.const 7
call $q
)
(memory 1)
(start $program)
)"""
```

In [ ]:
```
!wat2wasm proc.wat # validating generated code by translating to binary format
```

In [ ]:
```
%%capture out
runpywasm('proc.wasm')
```

In [ ]:
```
assert str(out) == """\
9
"""
```

Illustrating Lack of Optimization

In [ ]:
```
compileString("""
program p
  var x: integer
    x := 5
    x := x + 0
    x := 0 + x
    x := x × 1
    x := 1 × x
    x := x + 3
    x := 3 + x
""", 'opt.wat', target = 'wat')

with open('folding.wat', 'r') as f: asm = f.read()
assert asm == """\
(module
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
(global $x (mut i32) i32.const 0)
(global $y (mut i32) i32.const 0)
(func $q
(local $x i32)
(local $0 i32)
i32.const 3
local.set $x
i32.const 1
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 1
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
local.get $x
global.set $y
else
i32.const 7
global.set $y
end
global.get $y
call $write
i32.const 0
if
i32.const 5
call $write
else
i32.const 9
call $write
end
)
(global $_memsize (mut i32) i32.const 0)
(func $program
(local $0 i32)
i32.const 7
global.set $x
call $q
global.get $x
call $write
)
(memory 1)
(start $program)
)"""
```

In [ ]:

```
!wat2wasm opt.wat # validating generated code by translating to binary format
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js