CMSC 430: Introduction to Compilers

Fraud: Binding, variables, and binary operations

Announcements 03/10/2026

- Quiz 6, due Monday 03/23
- Midterm 1, March 12, take home
- You can submit the exam multiple times. Last submission will be graded.
- Assignment 4 will be released after midterm 1.

- Today:
  - Fraud

Fraud

- Fraud adds local binding and the ability to use binary operations to our target language.

```
(let ((id0 e1)) e2)
```

- This form binds the identifier id0 to value of e1 within the scope of e2. It is same as `let x = e1 in e2` in OCaml

Let: examples

```
 x                                          meaningless
(let ((x 7)) x)                               7
(let ((x 7)) 2)                               2
(let ((x 7)) (add1 x))                        8
(let ((x (add1 7))) x)                        8
(let ((x 7)) (let ((y 2)) x))                 7
(let ((x 7)) (let ((x 2)) x))                 2
(let ((x (add1 x))) x)                     meaningless
(let ((x 7)) (let ((x (add1 x))) x))          8
```

Let: examples

- **Concrete:**
  - `(let ((x 0)) 1)`
- **AST:**
  - `(Let 'x (Lit 0) (Var 'x))`

Binary operations

- Fraud adds the following binary operators
  - `+ : (+ 1 2)`
  - `- : (- 1 2)`
  - `< : (< 1 2)`
  - `= : numeric equality`

Binary operations

- AST
  - `(Prim2 '+ (Lit 1) (Lit 2))` `;(+ 1 2)`
  - `(Prim2 '- (Lit 1) (Lit 2))` `;(- 1 2)`
  - `(Let '1 (Lit 1) (Let 'y (Let 2) (Prim2 '+ (Var 'x) (Var 'x))))`
    - `; (let ((x 1) (let ((y 2)) (+ x y))`
  - …

Interpreting variables and bindings

The interpreter uses an environment to manage associations between variables and their values.

```
(define (interp e)
(with-handlers ([err? identity])
(interp-e e '())))

;; Expr Env -> Value { raises 'err }
(define (interp-e e r) ;; where r closes e
  (match e
  [(Var x) (lookup r x)]
```

Interpreting variables and bindings

The interpreter uses an environment to manage associations between variables and their values.

```
(define (interp-e e r)
  (match e
  [(Var x) (lookup r x)]
  …
```

We need run-time representation of environments, identifier names, and implementation of `lookup` and `ext` in assembly. Seems… hard.

Representing the Environment

{
  x menas 42
  y means 42
  z means #f
}

```
`((x 42)
  (y 12)
  (z #f))
```

;; type Env = (Listof (List Id Value))

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (let ((z ...))
      ;; what do you know about the
      ;; environment used to evaluate e?
      e)))
```

Observe something about environments

- The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (+ (let ((z ...))
         ;; what do you know about the
         ;; environment used to evaluate e1?
         e1)
       ;; what about e2?
       e2)))
```

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (let ((z ...))
      ;; where will y's binding
      ;; be in the environment?
      y)))
```

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (let ((z ...))
      ;; where will z's binding
      ;; be in the environment?
      z)))
```

Observe something about environments

- Suppose we get to this point in interpreting a program:

```
(interp-env (Var 'x) '((y 1) (x 99) (p 7)))
```

- The program surrounding this occurrence of `x` has to have looked like this!

```
(let ((p ...))
  ...
  (let ((x ...))
    ...
    (let ((y ...))
      ... x ...)))
```

Observe something about environments

- Suppose we know the program looks like this:

```
(let ((p ...))
  ...
  (let ((x ...))
    ...
    (let ((y ...))
      ... x ...)))
```

- The environment that will be used when x is interpreted must look like this:

```
'((y ??) (x ??) (p ??))
```

- But now we can see that `lookup` will retrieve the second element; The location of a variables binding in the environment is a static property

Generalizing the observation

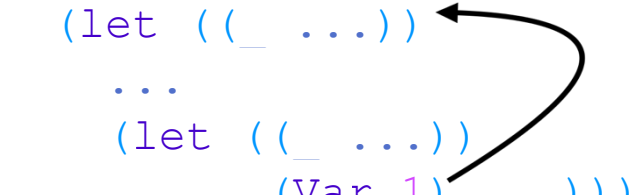
- For each variable occurrence, we can precisely calculate the location in the environment *before interpreting the program.*
-
- Lookup doesn’t need to be a linear search. We can compute the index of the value in the list.

Variable names are irrelevant

- An interpreter that uses lexical addresses

```
(let ((p ...))
  ...
  (let ((x ...))
    ...
    (let ((y ...))
      ... x ...)))
```

```
(let ((  ...))
  ...
  (let ((_ ...))
    ...
    (let ((_ ...))
      ... (Var 1) ...)))
```



- Environment can change from **`[Listof (List Id Value)]`** to **`[Listof Value]`**. **`lookup`** for **`(Var i)`** becomes **`(list-ref r i)`**. **`ext`** becomes cons.

Announcements

- A5: due 03/31, a week from today. Start early.
- M1 grades released.

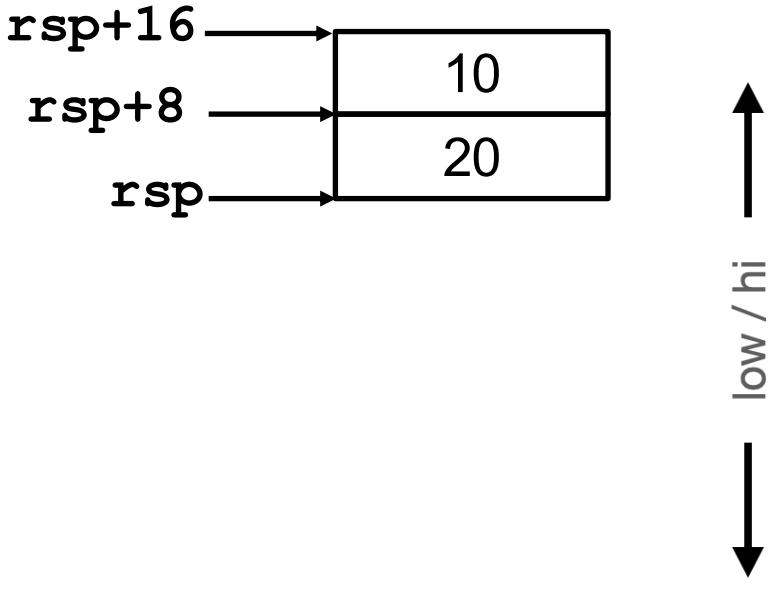
- Today:
  - Fraud
  - Hustle

An Overview of the Stack in x86

- The “stack” is just some memory allocated to your program.
- Its address is held in the **`rsp`** register.
- The stack grows “downward”, i.e. toward lower memory addresses.
- Instructions that affect the stack:
  - `(`**`Push`**` r), (`**`Pop`**` r)`
  - `(Add rsp n), (Sub rsp n)`
  - `(Call label), (Ret)`
  - `(Mov r (`**`Mem`**` rsp i)), (Mov (Mem rsp i) r)`
    where i is a multiple of 8

Stack operations

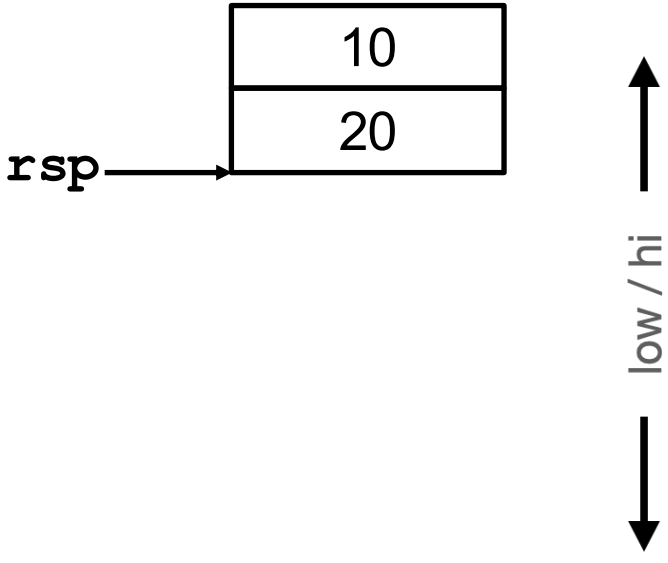
```
(Mov r9 10)
(Push r9)
(Mov r9 20)
(Push r9)
(Mov r9 (Mem rsp 8))
(Pop r9)
(Pop r9)
```



Stack operations

Popping when you don't care what on the stack

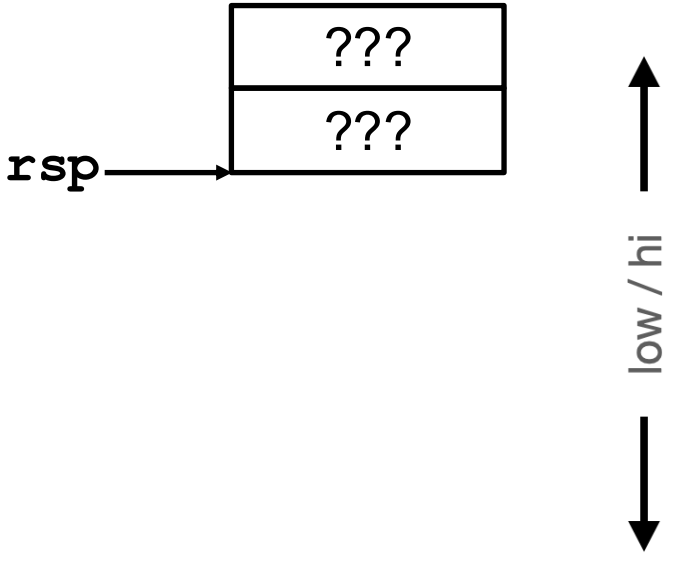
```
(Mov r9 10)
(Push r9)
(Mov r9 20)
(Push r9)
(Mov r9 (Mem rsp 8))
(Pop r9)
(Pop r9)
(Add rsp 8)
(Add rsp 8)
```



Stack operations

Pushing when you don't care what on the stack

```
(Mov r9 100)
(Sub rsp 8)
(Sub rsp 8)
```




```
(Mov r9 100)
(Sub rsp 16)
```

Various facts about the Fraud compiler

- Registers:
  - `rax` - return value
  - `rsp` - stack pointer
  - `rdi` - first param when calling run-time system
- Stack is 8-byte (64-bit) aligned
  - i.e. divisible by 8, ends in `#b000`
  - (Must align to 16-bytes to call)
- `(compile-e e c)` - leaves stack initial state
  - Length of compile time environment = Number of elements on stack at runtime

Stack-alignment in Fraud

- Stack must align to 16-bytes to call:
  - i.e. divisible by 16, ends in `#b0000`

- Pad the stack

```
Mov r15 rsp
And r15 #b1000
Sub rsp r15
Call foo
Add rsp r15
```

`r15` is 0 when `rsp` ends in `#b0000`
`r15` is 8 when `rsp` ends in `#b1000`

- Un-pad

```
(Add rsp r15)
```

Stack alignment example

Pad and un-pad the stack for write-byte:

```
(seq
  assert-byte
  pad-stack
  (Mov rdi rax)
  (Call 'write_byte)
  unpad-stack)
```

Example

```
(asm-display (compile-e (parse
        '(let ((x 10))
             (let ((y (let ((x 30)) x)))
                  (+ x y))))
        '()))
```

OCaml:

```
let x = 10 in
let y = (let x = 30 in x)
x + y
40
```

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

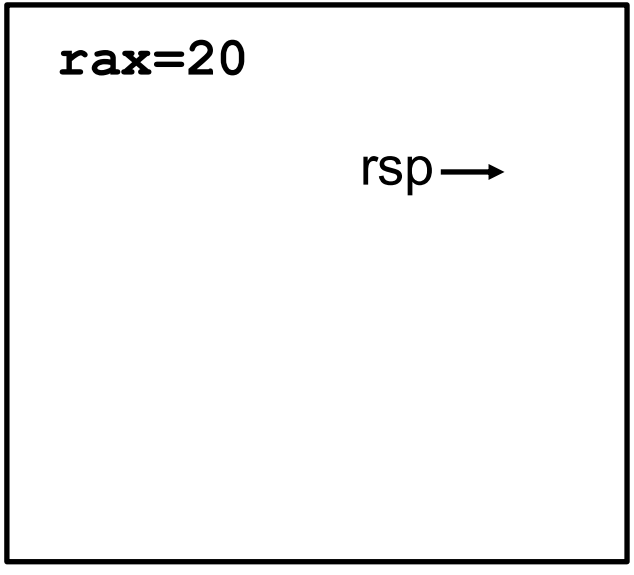
```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



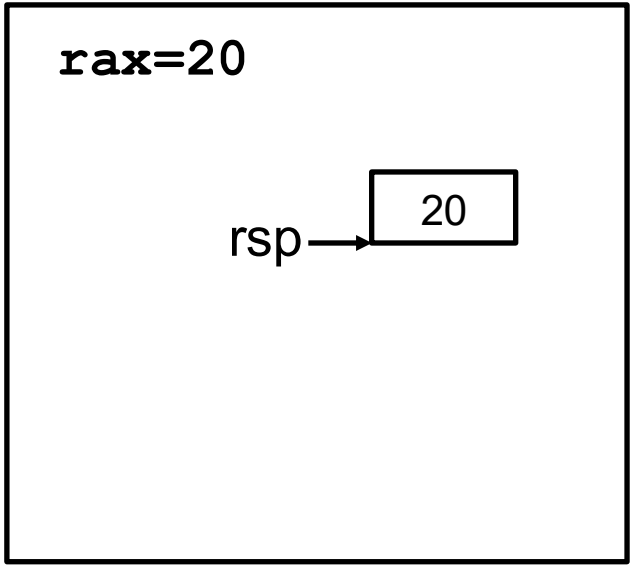

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Evaluate the expression 10

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



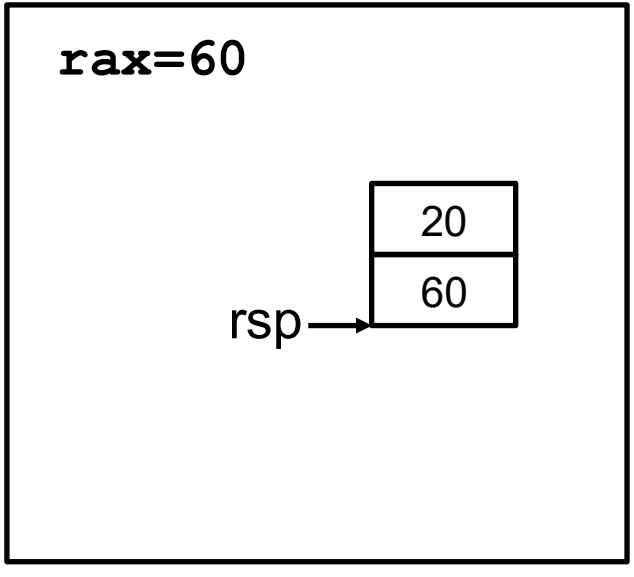

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Push the value of x to stack

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



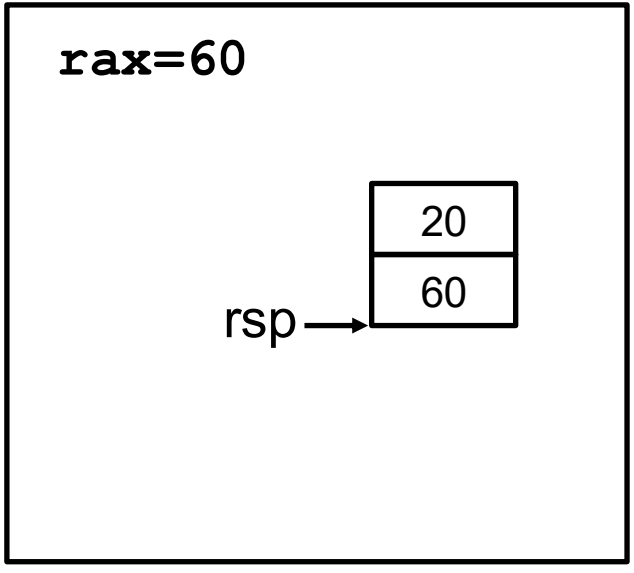

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Evaluate the expression 30 and push the value of inner x to stack

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



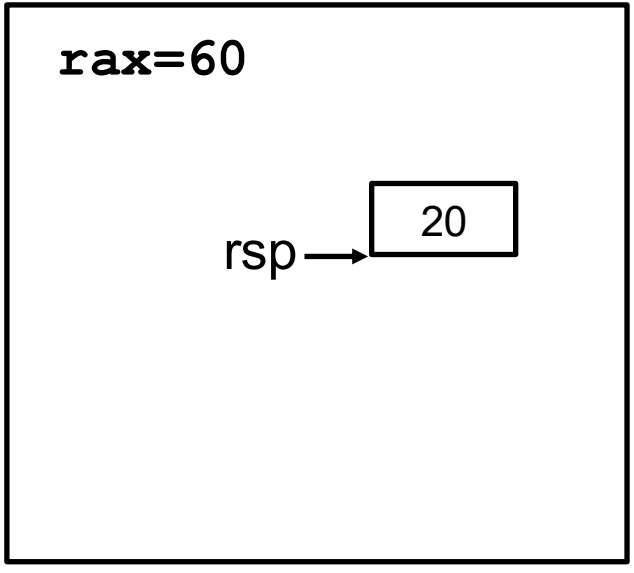

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Evaluate the inner x

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



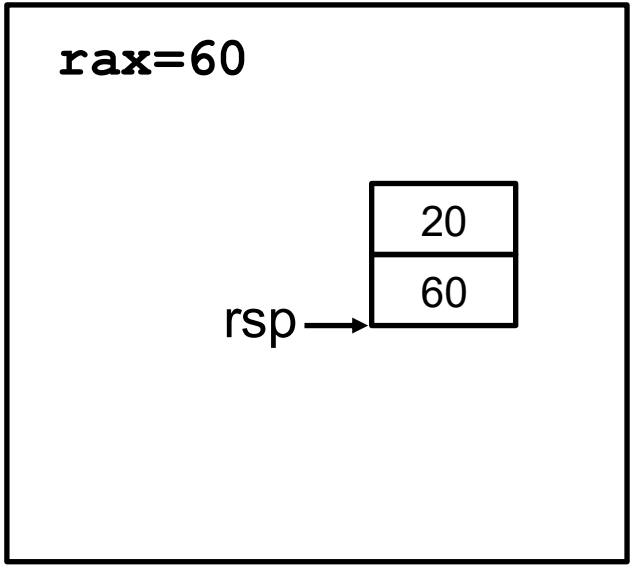

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Evaluate the inner x

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



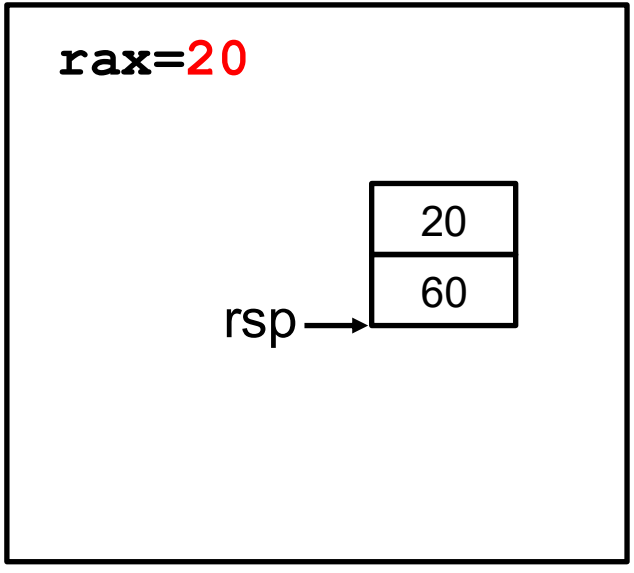

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

push the value of inner y to stack

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



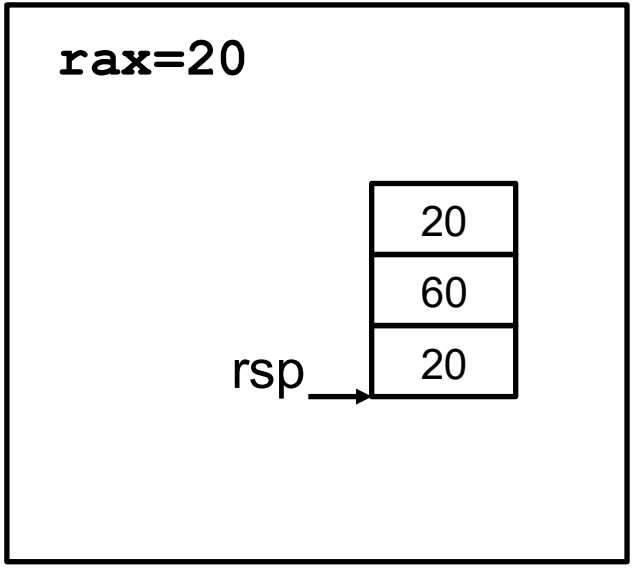

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Evaluate x in (+ x y)

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



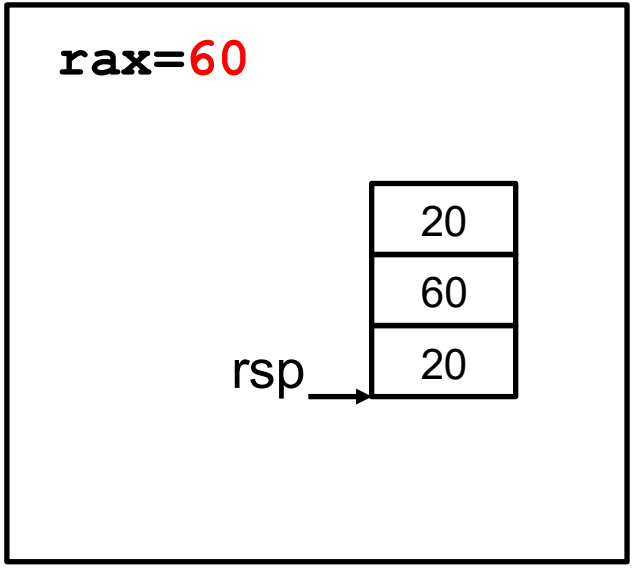

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Push x, the 1st argument of +

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



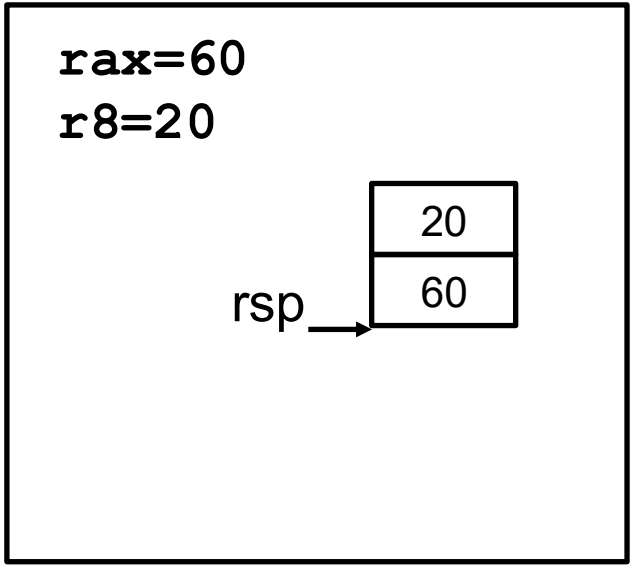

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

read y, the second argument of (+ x y)

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



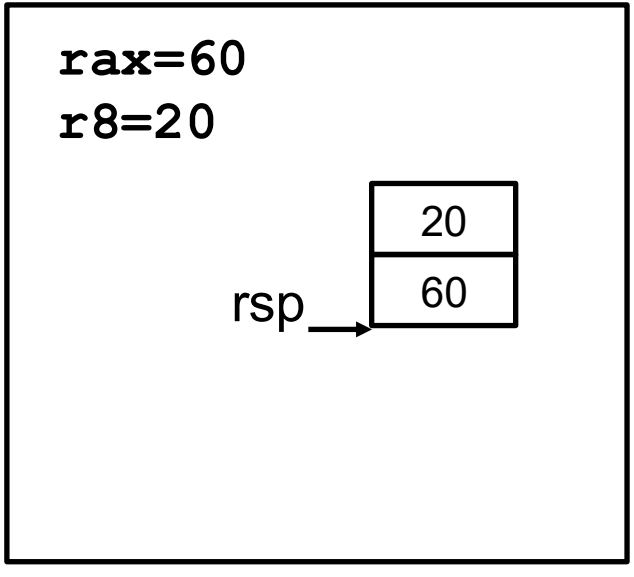

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

read the 1st argument of + to a register)

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



Assert integer: x

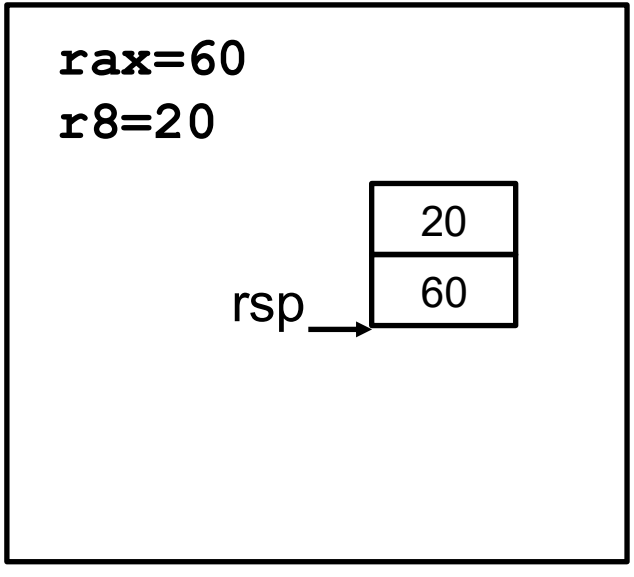
```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

assert 1st argument is an integer

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



Assert integer; y

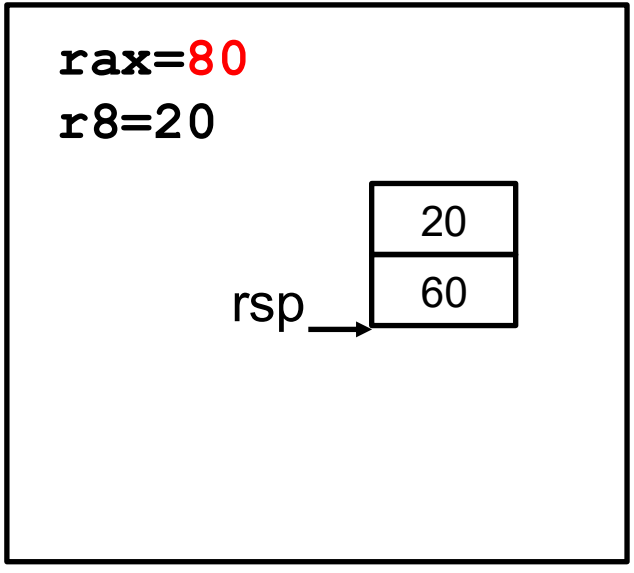
```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

assert 2nd argument is an integer

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



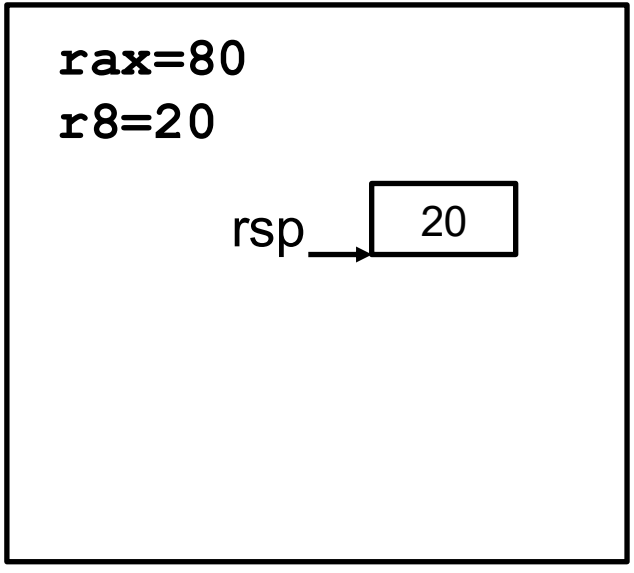
(+ x y)

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8;
add rsp, 8
```

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



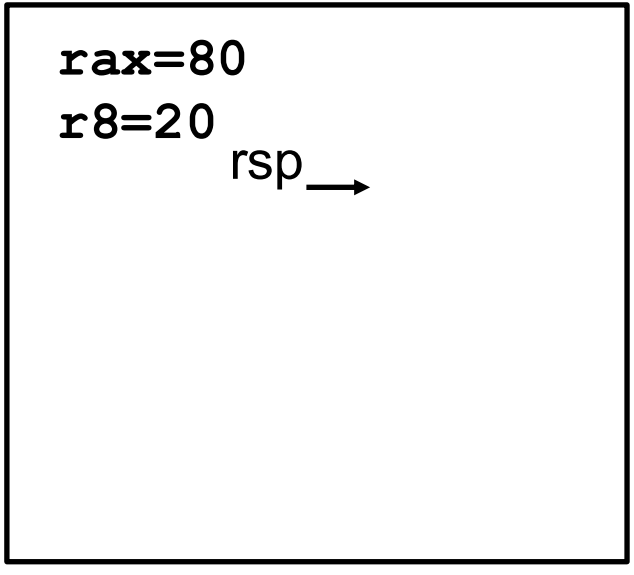

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8
add rsp, 8
```

Remove y from stack

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```



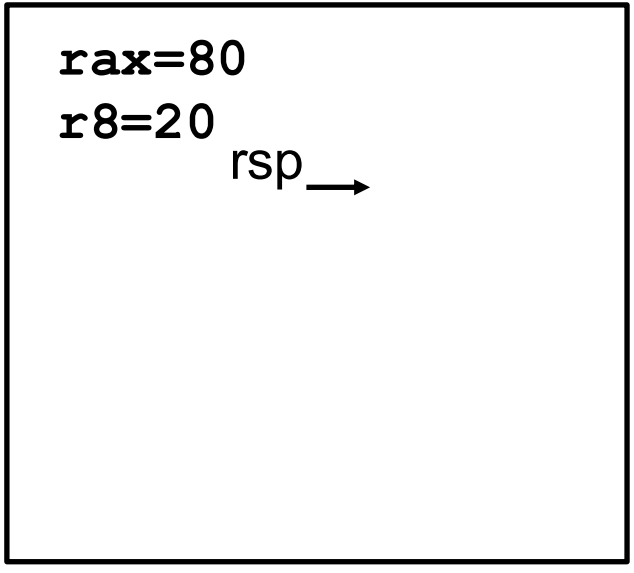

```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8
add rsp, 8
```

Remove x from stack

Example

```
(let ((x 10))
   (let ((y (let ((x 30)) x)))
(+ x y))))
```

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8
push rax
mov rax, [rsp + 8]
push rax
mov rax, [rsp + 8]
pop r8
```




```
mov r9, r8
and r9, 1
cmp r9, 0
jne _err
mov r9, rax
and r9, 1
cmp r9, 0
jne _err
add rax, r8
add rsp, 8
add rsp, 8
```

Return value: rax=80

Example: Summary

```
mov rax, 20
push rax
mov rax, 60
push rax
mov rax, [rsp + 0]
add rsp, 8 ; pop x
push rax ; push y value 30
mov rax, [rsp + 8] ; read x, the 1st argument for (+ x y)
push rax; push the 1st operand of +
mov rax, [rsp + 8]; read y, the second argument of (+ x y)
pop r8;  (read the 1st argument of + to a register)
mov r9, r8. ;assert integer for 1st argument
and r9, 1
cmp r9, 0
jne _err
mov r9, rax; assert integer for 2nd argument
and r9, 1
cmp r9, 0
jne _err
add rax, r8; (+ x y)
add rsp, 8
add rsp, 8
```