Simplifying
let, variables, operators, and if expressions
let \( x = 10 + 12 \) in
let \( y = 2 + x \) in
  if \( x > 23 \) then 3 else 4
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
### Simplification

<table>
<thead>
<tr>
<th>Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>let x = 22 in</td>
</tr>
<tr>
<td>let y = 2 + x in</td>
</tr>
<tr>
<td>if x &gt; 23 then 3 else 4</td>
</tr>
</tbody>
</table>

| Stack |

| Heap |

let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
Simplification

Workspace

let y = 2 + x in
  if x > 23 then 3 else 4

Stack

Heap

x  22
x is not a value: so look it up in the stack
let y = 2 + 22 in
   if x > 23 then 3 else 4
let \( y = 2 + 22 \) in
    if \( x > 23 \) then 3 else 4
Simplification

Workspace

let y = 24 in
  if x > 23 then 3 else 4

Stack

Heap

CIS120
\begin{align*}
\text{Workspace} & : \quad \text{let } y = 24 \text{ in } \\
& \quad \text{if } x > 23 \text{ then } 3 \text{ else } 4 \\
\text{Stack} & : \quad \begin{array}{c|c}
\text{x} & 22 \\
\end{array} \\
\text{Heap} & : \\
\end{align*}
Simplification

Workspace

\[
\text{if } x > 23 \text{ then } 3 \text{ else } 4
\]

Stack

<table>
<thead>
<tr>
<th>x</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>24</td>
</tr>
</tbody>
</table>

Heap
Simplification

Workspace

\[
\text{if } x > 23 \text{ then } 3 \text{ else } 4
\]

Stack

<table>
<thead>
<tr>
<th>x</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>24</td>
</tr>
</tbody>
</table>

Heap
Simplification

Workspace

if 22 > 23 then 3 else 4

Stack

x 22
y 24

Heap
Simplification

Workspace

if 22 > 23 then 3 else 4

Stack

x | 22
y | 24

Heap
if false then 3 else 4

Workspace

Stack
x 22

y 24

Heap
Simplification

Workspace

if false then 3 else 4

Stack

<table>
<thead>
<tr>
<th>x</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>24</td>
</tr>
</tbody>
</table>

Heap
## Simplification

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>x 22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y 24</td>
<td></td>
</tr>
</tbody>
</table>

DONE!
Simplification Rules

- A let-expression “let \( x = e \) in body” is ready if the expression \( e \) is a value
  - it is simplified by adding a binding of \( x \) to \( e \) at the end of the stack and leaving body in the workspace

- A variable is always ready
  - it is simplified by replacing it with its value from the stack, where binding lookup goes in order from most recent to least recent

- A primitive operator (like \(+\)) is ready if both of its arguments are values
  - it is simplified by replacing it with the result of the operation

- An “if” expression is ready if the test is true or false
  - if it is true, it is simplified by replacing it with the then branch
  - if it is false, it is simplified by replacing it with the else branch

CIS120
Simplifying lists and datatypes using the heap
For uniformity, we’ll pretend lists are declared like this:

```ocaml
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```
For uniformity, we’ll pretend lists are declared like this:

```ocaml
type 'a list =
| Nil
| Cons of 'a * 'a list
```
Simplification

Workspace

Cons (1, Cons (2, Cons (3, Nil)))

Stack

Heap
Simplification

Workspace

Cons (1, Cons (2, Cons (3, Nil)))

Stack

Heap

Nil
Simplification

Workspace

Stack

Heap

Cons (1, Cons (2, Cons (3, Nil)))
Simplification

Workspace: Cons (1, Cons (2, Nil))

Stack: Cons 3

Heap: Nil
Simplification

Workspace: 
Cons (1, )

Stack:
- Cons 3
- Cons 2

Heap:
- Nil
Simplification

Workspace

Stack

Heap

Cons 1

Cons 2

Cons 3

Nil

DONE!
Simplifying Datatypes

• A datatype constructor (like `Nil` or `Cons`) is ready if all its arguments are values
  – It is simplified by:
    • creating a new heap cell labeled with the constructor and containing the argument values*
    • replacing the constructor expression in the workspace by a reference to this heap cell

*Note: in OCaml, using a datatype constructor causes some space to be automatically allocated on the heap. Other languages have different mechanisms for accomplishing this: for example, the keyword ‘new’ in Java works similarly (as we’ll see in a few weeks).
Simplifying functions
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
Function Simplification

let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```ocaml
let add1 : int -> int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```
## Function Simplification

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let add1 : int -&gt; int = fun (x:int) -&gt; x + 1 in add1 (add1 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
let add1 = in
  add1 (add1 0)

fun (x:int) -> x + 1
let add1 = in
    add1 (add1 0)

fun (x:int) -> x + 1
Function Simplification

Workspace

```
add1 (add1 0)
```

Stack

```
add1
```

Heap

```
fun (x:int) -> x + 1
```
Function Simplification

Workspace

add1 (add1 0)

Stack

Heap

fun (x:int) -> x + 1
Function Simplification

Workspace

add1 (0)

Stack

add1

Heap

fun (x:int) -> x + 1
Function Simplification

Workspace: add1 (___ 0)

Stack: add1

Heap: fun (x:int) -> x + 1
Do the Call, Saving the Workspace

Workspace

\[ x + 1 \]

Stack

add1

Heap

fun (x:int) -> x + 1

add1 (___)

\[ x \]
\[ 0 \]

Note the saved workspace and pushed function argument.

- compare with the workspace on the previous slide.
- the name ‘x’ comes from the name in the heap

The new workspace is the *body* of the function
Function Simplification

Workspace

\[ x + 1 \]

Stack

\[ \text{add1} \]

Heap

\[ \text{fun} \ (x : \text{int}) \rightarrow x + 1 \]

\[ \text{add1} (\_\_\_) \]

[ ]

[0]
Function Simplification

Workspace

0+1

Stack

add1

Heap

fun (x:int) -> x + 1

add1 (___)

x 0
Function Simplification

Workspace

0+1

Stack

add1

Heap

fun (x:int) -> x + 1

add1 (____)

x 0
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

add1 (___)

x 0

1

POP!
See how the ASM *restored* the saved workspace, replacing its `hole` with the value computed into the old workspace. (Compare with previous slide.)
Function Simplification

```
add1 1

fun (x:int) -> x + 1
```
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

1
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

1
Function Simplification

Workspace

\[ x + 1 \]

Stack

add1

Heap

fun (x:int) -> x + 1

---

\[ x \]

\[ 1 \]
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

x + 1

____

x 1
Function Simplification

Workspace

1+1

Stack

add1

Heap

fun (x:int) -> x + 1

x 1
Function Simplification

Workspace: 1+1

Stack: add1

Heap: fun (x:int) -> x + 1

CIS120
```haskell
fun (x:int) -> x + 1

add1

add1 1

x 1

2

POP!
```
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

2

DONE!
Simplifying Functions

• A function definition “let rec f (x₁:t₁)...(xₙ:tₙ) = e in body” is always ready.
  – It is simplified by replacing it with “let f = fun (x₁:t₁)...(xₙ:tₙ) = e in body”

• A function “fun (x₁:t₁)...(xₙ:tₙ) = e” is always ready.
  – It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.

• A function call is ready if the function and its arguments are all values
  – it is simplified by
    • saving the current workspace contents on the stack
    • adding bindings for the function’s parameter variables (to the actual argument values) to the end of the stack
    • copying the function’s body to the workspace
Function Completion

When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren’t any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.