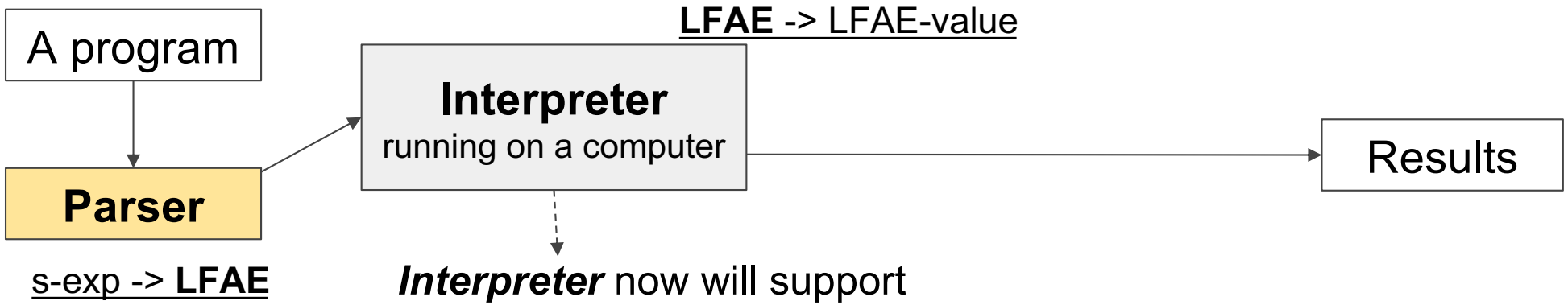
ITP30011
Recursion

Lecture 15
JC

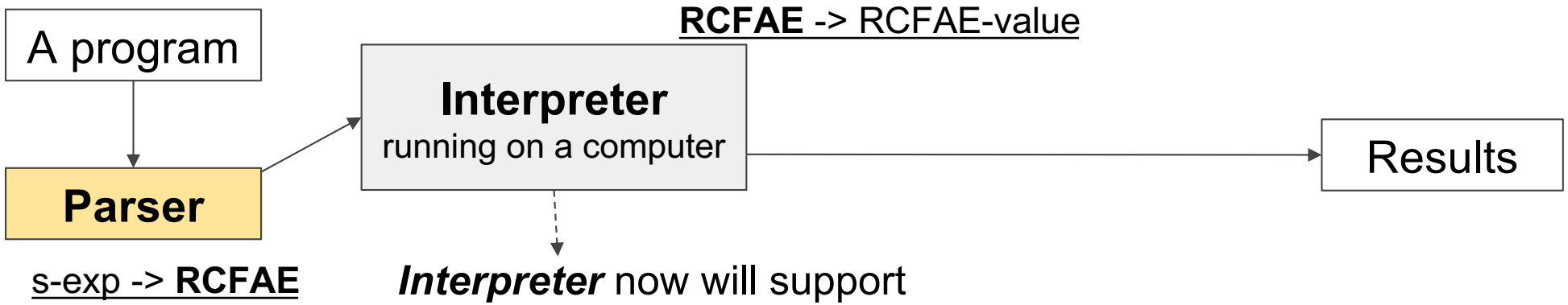
Big Picture (modeling languages: substitution)



Interpreter now will support

(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions
(5) Laziness

Big Picture (modeling languages: substitution)



Today, we are going to do logical thinking for recursion.

This will be helpful for implementing recursion for our language in the next lecture.

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

Our language now support multiplication for some famous recursive examples ;)

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

For recursive function, we need to support a conditional expression.
if (0==<RCFAE>)
    <RCFAE>
else
    <RCFAE>

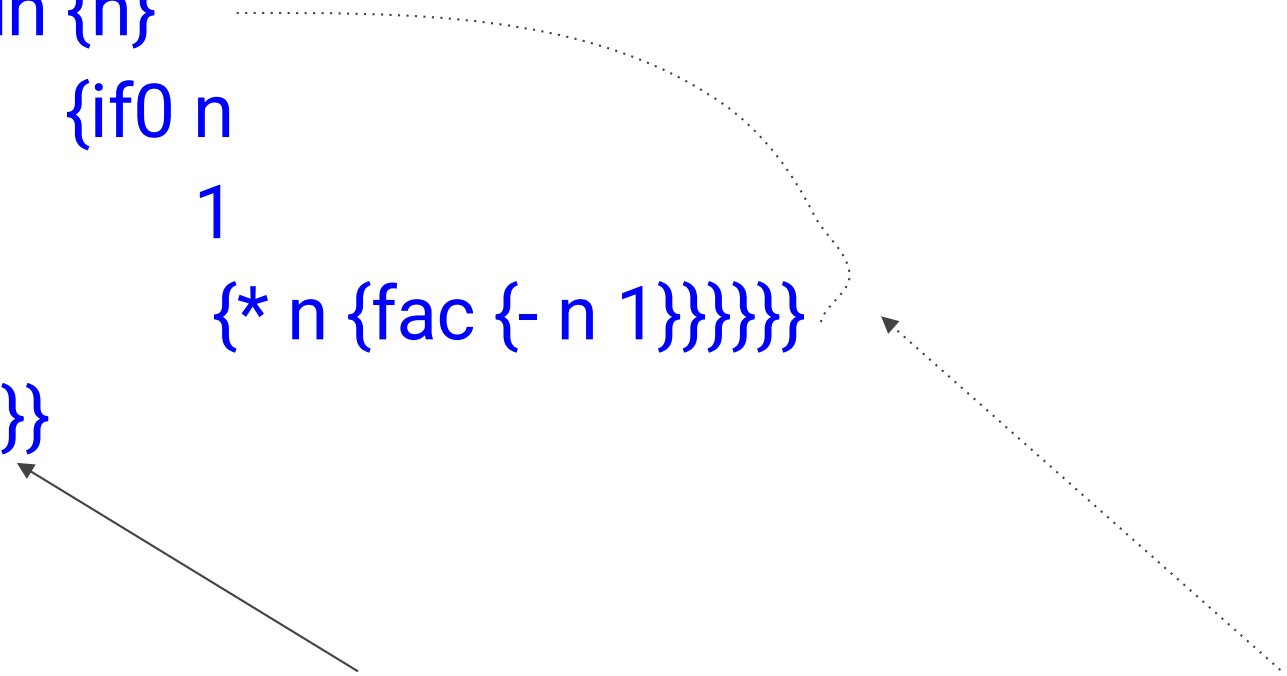
RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

Syntax for defining a recursive function and its call

Factorial - our language looks like this for recursion

```
{rec {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
     {fac 10}}
⇒ 3628800
```



rec binds both in the body expression and in the binding expression.

Factorial

```
{rec {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
     {fac 10}}
```

How about just using 'with'??

Factorial - Let's think this using 'with'?

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

Factorial - Let's think this using 'with'?

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

⇒ free identifier 'fac'!

Doesn't work: with does not support recursive definitions

Factorial - Let's think this using 'with'?

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

⇒ free identifier 'fac'!

Doesn't work: with does not support recursive definitions.

Factorial - Let's think this using 'with'?

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

⇒ free identifier 'fac'!

Doesn't work: with does not support recursive definitions. Still, at the point what we call fac, obviously we have a binding id *fac* for fac…

Factorial - Let's think this using 'with'?

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

⇒ free identifier 'fac'!

Doesn't work: with does not support recursive definitions. Still, at the point what we call fac, obviously we have a binding id *fac* for fac…

… so pass {fac 10} as *an argument*!

Factorial (Assume that a function can have multiple parameters.)

```
{with {facX {fun {facY n}
                 {if0 n
                      1
                      {* n {facY facY {- n 1}}}}}}
      {facX facX 10}}
```

Factorial

```
{with {facX {fun {facY n}
                 {if0 n
                      1
                      {* n {facY facY {- n 1}}}}}}
      {facX facX 10}}
```

Wrap this to fac back...

Factorial

```
{with {fac
         {fun {n}
              {with {facX {fun {facY n}
                               {if0 n
                                    1
                                    {* n {facY facY {- n 1}}}}}}
                    {facX facX n}}}}
       {fac 10}}
```

Factorial

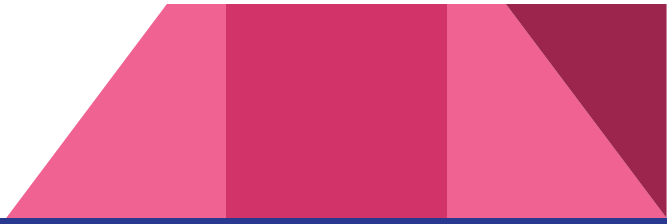
```
{with {fac
         {fun {n}
              {with {facX {fun {facY n}
                               {if0 n
                                    1
                                    {* n {facY facY {- n 1}}}}}}
                    {facX facX n}}}}
       {fac 10}}
```

But the language we implement has *only single-argument* functions...

From Multi-Arg. to Single-Arg.

```
{with {f {fun {x y z} {+ z {+ y x}}}}
      {f 1 2 3 }}
```

⇒ Rewrite this using a function only with one parameter?



From Multi-Arg. to Single-Arg.

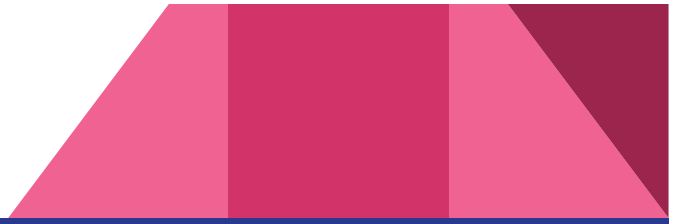
```
{with {f {fun {x y z} {+ z {+ y x}}}}
      {f 1 2 3 }}
```

⇒

```
{with {f {fun {x}
              {fun {y}
                   {fun {z}
                        {+ z {+ y x}}}}}}
      {{{f 1} 2} 3}}
```

Factorial

```
{with {fac
          {fun {n}
               {with {facX
                          {fun {facY n}
                               {if0 n
                                    1
                                    {* n {facY facY {- n 1}}}}}}
                     {facX facX n}}}}
     {fac 10}}
```



Factorial

```
{with {fac
          {fun {n}
               {with {facX
                          {fun {facY}
                               {fun {n}
                                    {if0 n
                                    1
                                    {* n {{facY facY} {- n 1}}}}}}}
                     {{facX facX} n}}}}
     {fac 10}}
```

Factorial

```
{with {fac
          {fun {n}
               {with {facX
                          {fun {facY}
                               {fun {n}
                                    {if0 n
                                    1
                                    {* n {{facY facY} {- n 1}}}}}}}
                     {{facX facX} n}}}}
      {fac 10}}
```

Simplify: {fun {n} {with {f ...} {{f f} n}}}

⇒ {with {f ...} {f f}} by "η reduction"

η reduction (eta reduction)

- If two functions lead to the same result, they are the same functions.
- {fun {n} {{fun {x} x} n}}
  {{fun {n} {{fun {x} x} n}} 2} Result: 2
- {fun {x} x}
  {{fun {x} x} 2} Result: 2
- {fun {n} {e n}} ⇒ e where n is not free in e.
  {fun {n} {{fun {x} {+ n x}} n}}
  {fun {x} {+ n x}} η reduction is not possible as n is free in {fun {x} n}.

η reduction (eta reduction)

- If two functions lead to the same result, they are the same functions.
- {fun {n} {{fun {x} x} n}}
  {{fun {n} {{fun {x} x} n}} 2} Result: 2
- {fun {x} x}
  {{fun {x} x} 2} Result: 2
- {fun {n} {e n}} ⇒ e where n is not free in e.
  {fun {n} {{fun {x} {+ n x}} n}}
  {fun {x} {+ n x}}

η reduction is not possible as n is free in {fun {x} {+ n x}.

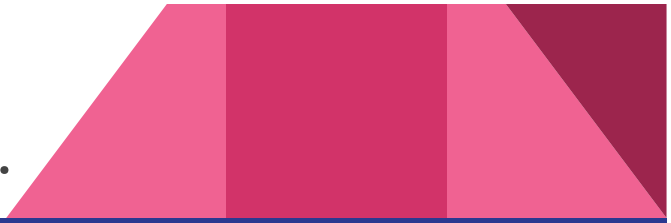
Free id: we cannot do η reduction for this code.

Factorial

```
{with {fac
           {fun {n}
                {with {facX
                           {fun {facY}
                                {fun {n}
                                     {if0 n
                                     1
                                     {* n {{facY facY} {- n 1}}}}}}}
                      {{facX facX} n}}}}
      {fac 10}}
```

"η reduction"

{fun {n} {e n}} ⇒ e where n is not free in e.



Factorial

```
{with {fac
        {fun {n}
          {with {facX
                  {fun {facY}
                    {fun {n}
                      {if0 n
                           1
                           {* n {{facY facY} {- n 1}}}}}}}
                {{facX facX} n}}}}
       {fac 10}}
```

This 'n' is a binding id.

"η reduction"

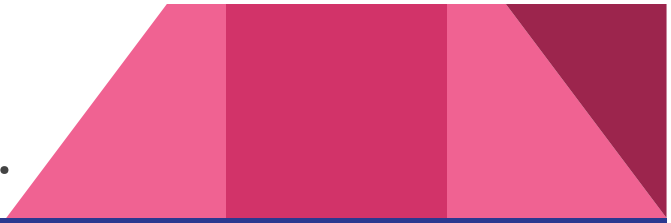
{fun {n} {e n}} ⇒ e where n is not free in e.

Factorial

```
{with {fac
         {fun {n}
              {with {facX
                        {fun {facY}
                             {fun {n}
                                  {if0 n
                                  1
                                  {* n {{facY facY} {- n 1}}}}}}}
                    {{facX facX} n}}}}
         {fac 10}}
```

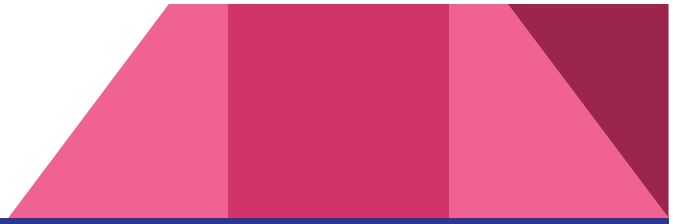
"η reduction"

{fun {n} {e n}} ⇒ e where n is not free in e.



Factorial

```
{with {fac
          {with {facX
                    {fun {facY}
                         {fun {n}
                              {if0 n
                                   1
                                   {* n {{facY facY} {- n 1}}}}}}}}
                {facX facX}}}
      {fac 10}}
```



Factorial

```
{with {fac
          {with {facX
                    {fun {facY}     ; Almost original fac
                         {fun {n}
                              {if0 n
                                   1
                                   {* n {{facY facY} {- n 1}}}}}}}
                {facX facX}}}
      {fac 10}}
```

Factorial - Original

```
{with {fac
        {fun {n}
             {if0 n
                  1
                  {* n {fac {- n 1}}}}}}}

  {fac 10}}
```

Factorial

```
{with {fac
        {with {facX
                {fun {facY}     ; Almost original fac
                  {fun {n}
                    {if0 n
                         1
                         {* n {{facY facY} {- n 1}}}}}}}
              {facX facX}}}
  {fac 10}}
```

Make this to be substituted by fac.

More like original: introduce a local binding for {facY facY}...

Factorial

```
{with {fac
          {with {facX
                    {fun {facY}
                         {with {fac {facY facY}}
                               ; Exactly like original fac
                               {fun {n}
                               {if0 n
                                    1
                                    {* n {fac {- n 1}}}}}}}}
                    {facX facX}}}
       {fac 10}}
```

Factorial

```
{with {fac
          {with {facX
                    {fun {facY}
                         {with {fac {facY facY}}
                               ; Exactly like original fac
                               {fun {n}
                               {if0 n
                                    1
                                    {* n {fac {- n 1}}}}}}}}
                    {facX facX}}}
       {fac 10}}
```

Opps! - this is an infinite loop

We used to evaluate {facY facY} only when n is non-zero.

Delay {facY facY}...

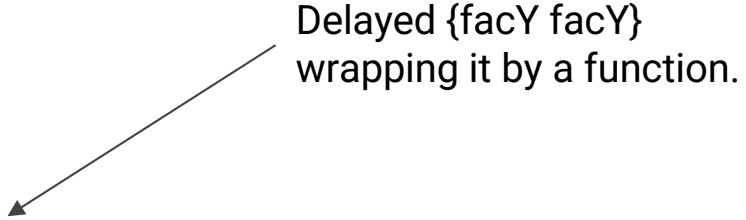
Can you improve our language to support 'if0'?
This code can run after delaying {facY facY}

(But we will implement a complete and general interpreter in the next class)

We may apply same logic for other recursive examples.

Factorial

```
{with {fac
          {with {facX
                    {fun {facY}
                         {with {fac {fun {x} {{facY facY} x}}}
                               ; Exactly like original fac
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}}
                    {facX facX}}}
      {fac 10}}
```

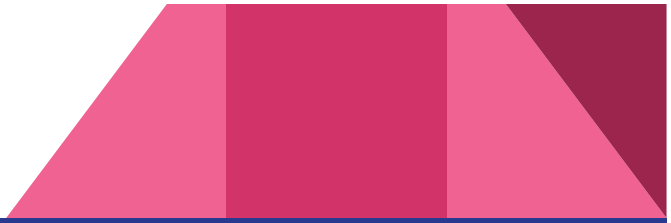


Factorial

```
{with {fac
           {with {facX
                      {fun {facY}
                           {with {fac {fun {x} {{facY facY} x}}}
                                 ; Exactly like original fac
                                  {fun {n}
                                       {if0 n
                                            1
                                             {* n {fac {- n 1}}}}}}}}
                      {facX facX}}}
      {fac 10}}
```

Now, what about fib, sum, etc.?

Abstract over the fac-specific part...

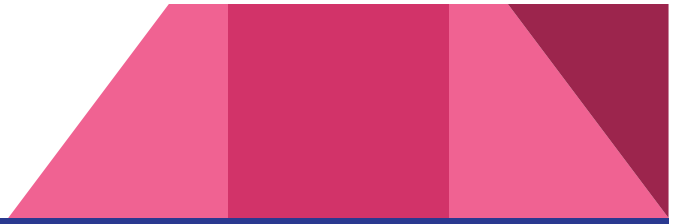


Make-Recursive and Factorial

```
{with {mk-rec {fun {body-proc}
                {with {fX {fun {fY}
                            {with {f {fun {x}
                                       {{fY fY} x}}}
                                  {body-proc f}}}}
                      {fX fX}}}}
      {with {fac {mk-rec
                  {fun {fac}
                    ; Exactly like original fac
                    {fun {n}
                      {if0 n
                           1
                           {* n {fac {- n 1}}}}}}}}
            {fac 10}}}
```

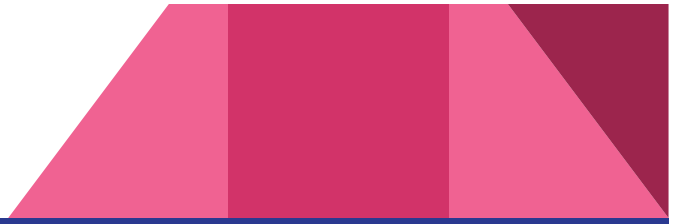
Fibonacci

```
{with {fib {mk-rec
                    {fun {fib}
                         ; Usual fib
                         {fun {n}
                              {if {or {= n 0} {= n 1}}
                                  1
                                  {+ {fib {- n 1}}
                                     {fib {- n 2}}}}}}}}
      {fib 5}}
```



Sum

```
{with {sum {mk-rec
                    {fun {sum}
                         ; Usual sum
                         {fun {l}
                              {if {empty? l}
                                  0
                                   {+ {first l}
                                      {sum {rest l}}}}}}}}}
      {sum '{1 2 3 4}}}
```



Do you want to use recursion in such a complicated way?

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

Syntax for defining a recursive function and its call

Factorial

```
{rec {fac {fun {n}
                {if0 n
                     1
                      {* n {fac {- n 1}}}}}}
     {fac 10}}
```

Topics we cover and schedule (tentative)

- Racket tutorials (L2,3)
- Modeling languages (L4,5)
- Interpreting arithmetic (L5)
- Language principles
  - Substitution (L6-7)
  - Function (L8)
  - Deferring Substitution (L9)
  - First-class Functions (L10-L12)
  - Laziness (L13,14)
  - Recursion (L15,16)
  - Mutable data structures (L17,18,19,20)
  - Variables (L21,22)
  - Continuations (L23-26)
- Guest Video Lecture (L27)

TODO

Read Chapter 9. Implementing Recursion

JC

jcnam@handong.edu

https://lifove.github.io

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or modified/created by JC based on the main text book.