But first, some clean up.

Our semantics:

\[
\begin{align*}
  e_1 &\to e'_1 \\
  e_1 e_2 &\to e'_1 e_2 \\
  v e_2 &\to v e'_2 \\
  A(e) &\to A(e') \\
  B(e) &\to B(e') \\
  (e_1, e_2) &\to (e'_1, e_2) \\
  (v_1, e_2) &\to (v_1, e'_2) \\
  e.1 &\to e'.1 \\
  e.2 &\to e'.2 \\
  e &\to e' \\
  \text{match } e \text{ with } Ax. e_1 | By. e_2 &\to \text{match } e' \text{ with } Ax. e_1 | By. e_2 \\
  \lambda x. e &\to e[v/x] \\
  (v_1, v_2).1 &\to v_1 \\
  (v_1, v_2).2 &\to v_2 \\
  \text{match } A(v) \text{ with } A x. e_1 | By. e_2 \to e_1[v/x] \\
  \text{match } B(v) \text{ with } Ay. e_1 | Bx. e_2 \to e_2[v/x]
\end{align*}
\]
But first, some clean up.

Our semantics:
Boring rules to grind sub-expressions down:

\[
\begin{align*}
 e_1 &\to e'_1 \\
 e_2 &\to e'_2 \\
 v &\to v \\
 e &\to e' \\
 A(e) &\to A(e') \\
 B(e) &\to B(e') \\
 (e_1, e_2) &\to (e'_1, e'_2) \\
 (v_1, e_2) &\to (v_1, e'_2) \\
 e.1 &\to e'.1 \\
 e.2 &\to e'.2
\end{align*}
\]

match \( e \) with \( Ax. e_1 \mid By. e_2 \) \to match \( e' \) with \( Ax. e_1 \mid By. e_2 \)

\[
\begin{align*}
 (\lambda x. e) v &\to e[v/x] \\
 (v_1, v_2).1 &\to v_1 \\
 (v_1, v_2).2 &\to v_2
\end{align*}
\]

match \( A(v) \) with \( Ax. e_1 \mid By. e_2 \) \to \( e_1[v/x] \)

match \( B(v) \) with \( Ay. e_1 \mid Bx. e_2 \) \to \( e_2[v/x] \)

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

\[
E ::= [\cdot] \mid E, e \mid v, E \mid (E, e) \mid (v, E) \mid (E.1) \mid (E.2) \\
| A(E) \mid B(E) \mid (match \ E \ with \ Ax. e_1 \mid By. e_2)
\]

But first, some clean up.

Our semantics:
Boring rules to grind sub-expressions down:

\[
\begin{align*}
 e_1 &\to e'_1 \\
 e_2 &\to e'_2 \\
 v &\to v \\
 e &\to e' \\
 A(e) &\to A(e') \\
 B(e) &\to B(e') \\
 (e_1, e_2) &\to (e'_1, e'_2) \\
 (v_1, e_2) &\to (v_1, e'_2) \\
 e.1 &\to e'.1 \\
 e.2 &\to e'.2
\end{align*}
\]

match \( e \) with \( Ax. e_1 \mid By. e_2 \) \to match \( e' \) with \( Ax. e_1 \mid By. e_2 \)

Interesting rules that actually do work:

\[
\begin{align*}
 (\lambda x. e) v &\to e[v/x] \\
 (v_1, v_2).1 &\to v_1 \\
 (v_1, v_2).2 &\to v_2
\end{align*}
\]

match \( A(v) \) with \( Ax. e_1 \mid By. e_2 \) \to \( e_1[v/x] \)

match \( B(v) \) with \( Ay. e_1 \mid Bx. e_2 \) \to \( e_2[v/x] \)

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

\[
E ::= [\cdot] \mid E, e \mid v, E \mid (E, e) \mid (v, E) \mid (E.1) \mid (E.2) \\
| A(E) \mid B(E) \mid (match \ E \ with \ Ax. e_1 \mid By. e_2)
\]

How many \([\cdot]\) ("holes") can an evaluation context have?
We can do better: Separate concerns

*Evaluation contexts* define where interesting work can happen:

\[
E ::= \[ \cdot \] | E e | v E | (E, e) | (v, E) | E.1 | E.2 \\
| A(E) | B(E) | (match E with A x. e_1 | By. e_2)
\]

How many \[ \cdot \] ("holes") can an evaluation context have? *Only one.*

\[E[e]\] just means to “fill the hole” in \(E\) with \(e\):

\[
(\[\cdot\].1)[(1, 2)] = (1, 2).1
\]
We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

\[ E ::= []; E e; v E; (E,e); (v,E); E.1; E.2 \]
| A(E) | B(E) | (match E with A.x. e1 | By. e2) \]

How many \([\cdot]\) ("holes") can an evaluation context have? **Only one.**

\(E[e]\) just means to "fill the hole" in \(E\) with \(e\):

\(([\cdot],1)(1,2)] = (1,2).1\)
\n\(([\cdot], \lambda x.x)[1] = (1, \lambda x.x)\)
\n\(([\cdot] x y)[\lambda a. \lambda b a] = (\lambda a. \lambda b a) x y\)

---

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

\[ E ::= []; E e; v E; (E,e); (v,E); E.1; E.2 \]
| A(E) | B(E) | (match E with A.x. e1 | By. e2) \]

\(E[e]\) just means to "fill the hole" in \(E\) with \(e\).

Now we can cleanly separate our semantics:

\[ e \rightarrow e' \text{ with } 1 \text{ rule:} \]
\[ e \oreg e' \rightarrow E[e'] \]
Evaluation with evaluation contexts

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

\[ E ::= [\cdot] | E \ e | v \ E | (E,e) | (v,E) | E.1 | E.2 \]

\[ | A(E) | B(E) | \text{(match } E \text{ with } Ax. \ e_1 | By. \ e_2) \]

\[ E[e] \text{ just means to “fill the hole” in } E \text{ with } e. \]

Now we can cleanly separate our semantics:

\[ e \rightarrow e' \text{ with 1 rule: } \frac{e \overset{p}{\rightarrow} e'}{E[e] \rightarrow E[e']} \]

\[ e \overset{p}{\rightarrow} e' \text{ does all the “interesting work”:} \]

- \((\lambda x. \ e) \overset{p}{\rightarrow} \ e[v/x] \)
- \((v_1, v_2).1 \overset{p}{\rightarrow} v_1 \)
- \((v_1, v_2).2 \overset{p}{\rightarrow} v_2 \)

\[ \text{match } A(v) \text{ with } Ax. \ e_1 | By. \ e_2 \overset{p}{\rightarrow} e_1[v/x] \]

\[ \text{match } B(v) \text{ with } Ay. \ e_1 | Bx. \ e_2 \overset{p}{\rightarrow} e_2[v/x] \]
Evaluation with evaluation contexts

\[ E ::= [\cdot] \mid E \ e \mid v \ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \]
\[ \mid A(E) \mid B(E) \mid (\text{match } E \text{ with } A x. \ e_1 \mid B y. \ e_2) \]

Evaluation relies on decomposition (unstapling the correct subtree)

- Given \( e \), find \( E, e_a, e_a' \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{p} e_a' \)

Many possible eval contexts may match a given \( e \) ...

\[
(\cdot)(1, (1, (1, (1, 1)))) = (1, (1, (1, (1, 1))))
\]
\[
((1, \cdot))(1, (1, (1, 1))) = (1, (1, (1, (1, 1))))
\]
\[
((1, 1, \cdot))(1, (1, 1)) = (1, (1, (1, (1, 1))))
\]
\[
((1, 1, (1, \cdot)))(1, 1) = (1, (1, (1, (1, 1))))
\]
\[
((1, 1, (1, (1, \cdot))))[1] = (1, (1, (1, (1, 1))))
\]

Evaluation with evaluation contexts

\[ E ::= [\cdot] \mid E \ e \mid v \ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \]
\[ \mid A(E) \mid B(E) \mid (\text{match } E \text{ with } A x. \ e_1 \mid B y. \ e_2) \]

Evaluation relies on decomposition (unstapling the correct subtree)

- Given \( e \), find \( E, e_a, e_a' \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{p} e_a' \)

Unique Decomposition Theorem: at most one decomposition of \( e \)

Evaluation with evaluation contexts

\[ E ::= [\cdot] \mid E \ e \mid v \ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \]
\[ \mid A(E) \mid B(E) \mid (\text{match } E \text{ with } A x. \ e_1 \mid B y. \ e_2) \]

Evaluation relies on decomposition (unstapling the correct subtree)

- Given \( e \), find \( E, e_a, e_a' \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{p} e_a' \)

Unique Decomposition Theorem: at most one decomposition of \( e \)

- \( E \) carefully picks leftmost non-value sub-expression
Evaluation with evaluation contexts

\[ E ::= [·] | E e | v E | (E, e) | (v, E) | E.1 | E.2 \\
| A(E) | B(E) | (match E with Ax.e1 | By.e2) \]

Evaluation relies on decomposition (unstapling the correct subtree)
- Given \( e \), find \( E, e_a, e'_a \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{p} e'_a \)

Unique Decomposition Theorem: at most one decomposition of \( e \)
- \( E \) carefully picks leftmost non-value sub-expression
- Hence eval is deterministic: at most one primitive step applies

Progress Theorem (restated): If \( e \) is well-typed, then there is a decomposition or \( e \) is a value

Evaluation Contexts: So what?

Small-step semantics (old) and evaluation-context semantics (new) are very similar:
- Totally equivalent step sequence
  - (made both left-to-right call-by-value)
- Just rearranged things to be more concise: Each boring rule became a form of \( E \)
Continuations

Now that we have defined $E$ explicitly in our *metalanguage*, what if we also put it on our *language*

- From metalanguage to language is called *reification*

First-class continuations:

$$e ::= \ldots | \text{letcc } x. \ e | \text{throw } e \ e | \text{cont } E$$
Continuations

Now that we have defined $E$ explicitly in our metalanguage, what if we also put it on our language

- From metalanguage to language is called \textit{reification}

First-class continuations:

$$
e ::= \ldots | \text{letcc } x. e | \text{throw } e e | \text{cont } E
\varepsilon ::= \ldots | \text{cont } E
E ::= \ldots | \text{throw } E e | \text{throw } \varepsilon E
$$

$$E[\text{letcc } x. e] \rightarrow E[(\lambda x. e)(\text{cont } E)]$$

Continuations

Now that we have defined $E$ explicitly in our metalanguage, what if we also put it on our language

- From metalanguage to language is called \textit{reification}

First-class continuations:

$$
e ::= \ldots | \text{letcc } x. e | \text{throw } e e | \text{cont } E
\varepsilon ::= \ldots | \text{cont } E
E ::= \ldots | \text{throw } E e | \text{throw } \varepsilon E
$$

$$E[\text{letcc } x. e] \rightarrow E[(\lambda x. e)(\text{cont } E)] \quad E[\text{throw } (\text{cont } E')] \varepsilon \rightarrow E'[\varepsilon]$$
Continuations

Now that we have defined $E$ explicitly in our metalanguage, what if we also put it on our language

- From metalanguage to language is called reification

First-class continuations:

$$
e ::= \ldots | \text{letcc } x. e | \text{throw } v E | \text{cont } E$$

$$v ::= \ldots | \text{cont } E$$

$$E ::= \ldots | \text{throw } v E$$

$E[\text{letcc } x. e] \rightarrow E[(\lambda x. e)(\text{cont } E)]$

$E[\text{throw } (\text{cont } E') v] \rightarrow E'[v]$

- New operational rules for $\rightarrow$ not $\xrightarrow{P}$ because “the $E$ matters”
- $\text{letcc } x. e$ grabs the current evaluation context (“the stack”) (not in source programs: “saved stack (value)”)
- $\text{throw } (\text{cont } E') v$ restores old context: “jump somewhere” (restores old context: “jump somewhere”)

Continuations

Now that we have defined $E$ explicitly in our metalanguage, what if we also put it on our language

- From metalanguage to language is called reification

First-class continuations:

$$
e ::= \ldots | \text{letcc } x. e | \text{throw } v E | \text{cont } E$$

$$v ::= \ldots | \text{cont } E$$

$$E ::= \ldots | \text{throw } v E$$

$E[\text{letcc } x. e] \rightarrow E[(\lambda x. e)(\text{cont } E)]$

$E[\text{throw } (\text{cont } E') v] \rightarrow E'[v]$

- New operational rules for $\rightarrow$ not $\xrightarrow{P}$ because “the $E$ matters”
- $\text{letcc } x. e$ grabs the current evaluation context (“the stack”) (not in source programs: “saved stack (value)”)
- $\text{throw } (\text{cont } E') v$ restores old context: “jump somewhere” (restores old context: “jump somewhere”)
Examples (exceptions-like)

1 + (letcc k. 2 + 3) →* 6

1 + (letcc k. 2 + (throw k 3)) →* 4

1 + (letcc k. (throw k (2 + 3))) →* 6

1 + (letcc k. (throw k (throw k (throw k 2)))) →* 3

Examples (exceptions-like)

1 + (letcc k. 2 + 3) →* 6

1 + (letcc k. 2 + (throw k 3)) →* 4

1 + (letcc k. (throw k (2 + 3))) →* 6

1 + (letcc k. (throw k (throw k (throw k 2)))) →* 3
Another view

If you’re confused, think call stacks:

▶ What if your favorite language had operations for:
  ▶ Store current stack in \( x \)
  ▶ Replace current stack with stack in \( x \)

▶ “Resume the stack’s hole” with something different or when mutable state is different
  ▶ Else you are sure to have an infinite loop since you will later resume the stack again

Example (“time travel”)

SML/NJ has continuations. This runs and binds 10 to \( z \):

```sml
open SMLofNJ.Cont
val g : int cont option ref = ref NONE
val x = ref true (* avoids infinite loop *)
val y = ref (1 + 2 + (callcc (fn k => ((g := SOME k); 3))))
val z = if !x then (x := false; throw (valOf (!g)) 7) else !y
```

Is this useful?

First-class continuations are a **single** construct sufficient for:

▶ Exceptions

▶ Cooperative threads (including coroutines)
  ▶ “yield” captures the continuation (the “how to resume me”) and gives it to the scheduler (implemented in the language), which then throws to another thread’s “how to resume me”

▶ Other crazy things
  ▶ Often called the “goto of functional programming” — incredibly powerful, but nonstandard uses are usually inscrutable
  ▶ Key point is that we can “jump back in” unlike boring-old exceptions

Where are we

Done:

▶ Redefined our operational semantics using evaluation contexts
▶ That made it easy to define first-class continuations
▶ Example uses of continuations

---

(Elements of the text are extracted or generated to ensure readability.)
Where are we

Done:
▶ Redefined our operational semantics using evaluation contexts
▶ That made it easy to define first-class continuations
▶ Example uses of continuations

Now: How the heck do we implement this?

Rather than adding a powerful primitive, we can achieve the same effect via a whole-program translation into a sublanguage (source-to-source transformation)
▶ Every function takes extra arg: continuation says what’s next
▶ Never “return” — instead call current continuation w/ result
▶ Every expression becomes a continuation-accepting function
Where are we

Done:
▶ Redefined our operational semantics using evaluation contexts
▶ That made it easy to define first-class continuations
▶ Example uses of continuations

Now: How the heck do we implement this?

Rather than adding a powerful primitive, we can achieve the same effect via a whole-program translation into a sublanguage (source-to-source transformation)
▶ Every function takes extra arg: continuation says what’s next
▶ Never “return” — instead call current continuation w/ result
▶ Every expression becomes a continuation-accepting function
▶ Will be able to reintroduce letcc and throw “for free”

CPS examples

Invariant: every function takes continuation as extra argument

let mult’ ...

CPS examples

Invariant: every function takes continuation as extra argument

let mult’ x y k = ...
CPS examples

Invariant: every function takes continuation as extra argument

let mult' x y k = k (x * y)

let add' x y k = k (x + y)

let sub' x y k = k (x - y)

let eq' x y k = k (x = y)

let rec fact' n k = ...
CPS examples

OK, now you convert:

```latex
let fact n =
  aux n 1

let rec aux n acc =
  if n = 0 then acc
  else
    aux (n - 1) (n * acc)
```

The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

```latex
\[ e ::= x \mid \lambda x. e \mid e \mid e + e \]
\[ v ::= x \mid \lambda x. e \mid c \]
\]

\[ \text{CPS}_E(v) = \]

\text{CPS}_E(v) =
The CPS transformation (one way to do it)
A metafunction from expressions to expressions

Example source language (other features similar):

\[
\begin{align*}
  e &::= x | \lambda x. e | e \ e | c | e + e \\
  v &::= x | \lambda x. e | c
\end{align*}
\]

\[
\begin{align*}
  \text{CPS}_E(v) &= \lambda k. k \ \text{CPS}_V(v) \\
  \text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \ \lambda x_1. \ \text{CPS}_E(e_2) \ \lambda x_2. \ k \ (x_1+x_2)
\end{align*}
\]
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[
\begin{align*}
e & ::= x \mid \lambda x. e \mid e \, e \mid c \mid e + e \\
v & ::= x \mid \lambda x. e \mid c
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_E(v) &= \lambda k. \text{CPS}_V(v) \\
\text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \, \lambda x_1. \text{CPS}_E(e_2) \, \lambda x_2. k \, (x_1 + x_2) \\
\text{CPS}_E(e_1 \, e_2) &= \lambda k. \text{CPS}_E(e_1) \, \lambda f. \text{CPS}_E(e_2) \, \lambda x. f \, x \, k
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_V(c) &= c
\end{align*}
\]
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[
\begin{align*}
e & ::= x \mid \lambda x. e \mid e \ e \mid c \mid e + e \\
v & ::= x \mid \lambda x. c
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_E(v) &= \lambda k. \text{CPS}_V(v) \\
\text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. k (x_1 + x_2) \\
\text{CPS}_E(e_1 \ e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f \ x \ k
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_V(c) &= c \\
\text{CPS}_V(x) &= x \\
\text{CPS}_V(\lambda x. e) &= \lambda x. \lambda k. \text{CPS}_E(e) \ k
\end{align*}
\]

The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[
\begin{align*}
e & ::= x \mid \lambda x. e \mid e \ e \mid c \mid e + e \\
v & ::= x \mid \lambda x. c
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_E(v) &= \lambda k. \text{CPS}_V(v) \\
\text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. k (x_1 + x_2) \\
\text{CPS}_E(e_1 \ e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f \ x \ k
\end{align*}
\]

\[
\begin{align*}
\text{CPS}_V(c) &= c \\
\text{CPS}_V(x) &= x \\
\text{CPS}_V(\lambda x. e) &= \lambda x. \lambda k. \text{CPS}_E(e) \ k
\end{align*}
\]

To run the whole program \( e \), do

\[ \text{CPS}_E(e) \ (\lambda x. x) \]
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[ \begin{align*}
\text{e} & ::= x \mid \lambda x. \text{e} \mid \text{e e} \mid c \mid \text{e + e} \\
\text{v} & ::= x \mid \lambda x. \text{e} \mid c \\
\end{align*} \]

\[ \begin{align*}
\text{CPS}_E(v) &= \lambda k. \text{CPS}_V(v) \\
\text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. k (x_1 + x_2) \\
\text{CPS}_E(e_1 e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f x k \\
\text{CPS}_V(c) &= c \\
\text{CPS}_V(x) &= x \\
\text{CPS}_V(\lambda x. e) &= \lambda x. \lambda k. \text{CPS}_E(e) k \\
\end{align*} \]

To run the whole program \( e \), do \( \text{CPS}_E(e) (\lambda x. x) \)

Result of the CPS transformation

- Correctness: \( e \) is equivalent to \( \text{CPS}_E(e) \lambda x. x \)
- If whole program has type \( \tau_P \) and \( e \) has type \( \tau \), then \( \text{CPS}_E(e) \) has type \( (\tau \rightarrow \tau_P) \rightarrow \tau_P \)
- Fixes evaluation order: \( \text{CPS}_E(e) \) will evaluate \( e \) in left-to-right call-by-value
  - Other similar transformations encode other evaluation orders
  - Every intermediate computation is bound to a variable (helpful for compiler writers)

Result of the CPS transformation

- Correctness: \( e \) is equivalent to \( \text{CPS}_E(e) \lambda x. x \)
- If whole program has type \( \tau_P \) and \( e \) has type \( \tau \), then \( \text{CPS}_E(e) \) has type \( (\tau \rightarrow \tau_P) \rightarrow \tau_P \)

Fixes evaluation order: \( \text{CPS}_E(e) \) will evaluate \( e \) in left-to-right call-by-value

- Other similar transformations encode other evaluation orders
- Every intermediate computation is bound to a variable (helpful for compiler writers)
Result of the CPS transformation

- Correctness: $e$ is equivalent to $\text{CPS}_E(e) \ \lambda x. \ x$
- If whole program has type $\tau_P$ and $e$ has type $\tau$, then $\text{CPS}_E(e)$ has type $(\tau \rightarrow \tau_P) \rightarrow \tau_P$
- Fixes evaluation order: $\text{CPS}_E(e)$ will evaluate $e$ in left-to-right call-by-value
  - Other similar transformations encode other evaluation orders
  - Every intermediate computation is bound to a variable (helpful for compiler writers)
- For all $e$, evaluation of $\text{CPS}_E(e)$ stays in this sublanguage:

\[
e ::= v \mid v \cdot (v \cdot v) \mid (v + v) \\
v ::= x \mid \lambda x. \ e \mid c
\]

Encoding first-class continuations

If you apply the CPS transform, then you can add letcc and throw “for free” right in the source language

\[
\text{CPS}_E(\text{letcc } k \cdot e) = \text{CPS}_E(e)
\]

Result of the CPS transformation

- Correctness: $e$ is equivalent to $\text{CPS}_E(e) \ \lambda x. \ x$
- If whole program has type $\tau_P$ and $e$ has type $\tau$, then $\text{CPS}_E(e)$ has type $(\tau \rightarrow \tau_P) \rightarrow \tau_P$
- Fixes evaluation order: $\text{CPS}_E(e)$ will evaluate $e$ in left-to-right call-by-value
  - Other similar transformations encode other evaluation orders
  - Every intermediate computation is bound to a variable (helpful for compiler writers)
- For all $e$, evaluation of $\text{CPS}_E(e)$ stays in this sublanguage:

\[
e ::= v \mid v \cdot (v \cdot v) \mid (v + v) \\
v ::= x \mid \lambda x. \ e \mid c
\]

- Hence no need for a call-stack: every call is a tail-call
  - Now the program is maintaining the evaluation context via a closure that has the next “link” in its environment that has the next "link" in its environment, etc.
Encoding first-class continuations
If you apply the CPS transform, then you can add \texttt{letcc} and \texttt{throw} “for free” right in the source language

\[
\begin{align*}
\text{CPS}_E(\text{letcc } k. e) &= \lambda k. \text{CPS}_E(e) \ k \\
\text{CPS}_E(\text{throw } e_1 e_2) &= \lambda k. \text{CPS}_E(e_1) \ \lambda x_1. \text{CPS}_E(e_2) \ \lambda x_2. x_1 x_2 \\
\end{align*}
\]

or just \(x_1\)

\texttt{letcc} gets passed the current continuation just as it needs

\texttt{throw} ignores the current continuation just as it should

You can also manually program in this style (fully or partially)

\textgreater{} Has other uses as a programming idiom too...
Encoding first-class continuations

If you apply the CPS transform, then you can add letcc and throw “for free” right in the source language

\[
\begin{align*}
\text{CPS}_E(\text{letcc } k. e) &= \lambda k. \text{CPS}_E(e) k \\
\text{CPS}_E(\text{throw } e_1 e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. x_1 x_2 \\
&\text{or just } x_1
\end{align*}
\]

- letcc gets passed the current continuation just as it needs
- throw ignores the current continuation just as it should

You can also manually program in this style (fully or partially)
- Has other uses as a programming idiom too...

A useful advanced programming idiom

- A first-class continuation can “reify session state” in a client-server interaction
  - If the continuation is passed to the client, which returns it later, then the server can be stateless
  - Suggests CPS for web programming
  - Better: tools that do the CPS transformation for you
    - Gives you a “prompt-client” primitive without server-side state

- Because CPS uses only tail calls, it avoids deep call stacks when traversing recursive data structures
  - See lec13code.ml for this and related idioms

In short, “thinking in terms of CPS” is a powerful technique few programmers have