CS107: Intermediate Representations and CPS

Guannan Wei
guannan.wei@tufts.edu

February 5, 2026

Spring 2026

Tufts University

Logistics & Recap

- Project 1 and 2 are graded
- Visit my office hours if you have questions about your grades

Logistics & Recap

- Project 1 and 2 are graded
- Visit my office hours if you have questions about your grades
- **What have we learned so far?**

A Taste Of MiniScala

A MiniScala function to compute x to the power of y:

```
def pow(x: Int,  y: Int): Int =
  if (y == 0) 1                  // pow(x, 0) == 1
  else if (even(y)) {
    val t = pow(x, y / 2)        // pow(x, 2z) = pow(pow(x, z), 2)
    t * t
  } else {
    x * pow(x, y - 1)            // pow(x, z + 1) = x * pow(x, z)
  }
```

A Taste Of MiniScala

Say “Hello”:

```
val arr = new Array[Int](5);
arr(0) = 'H'; arr(1) = 'e'; arr(2) = 'l'; arr(3) = 'l'; arr(4) = 'o';

var i = 0;
while (i < 5) {
  putchar(arr(i));
  i = i + 1
};

0
```

A Taste Of MiniScala

Our implementation of MiniScala is already quite powerful.

- We can do essentially everything we can do in C
- Missing features (structs, strings): can be implemented as arrays

What are still missing?

A Taste Of MiniScala

Our implementation of MiniScala is already quite powerful.

- We can do essentially everything we can do in C
- Missing features (structs, strings): can be implemented as arrays

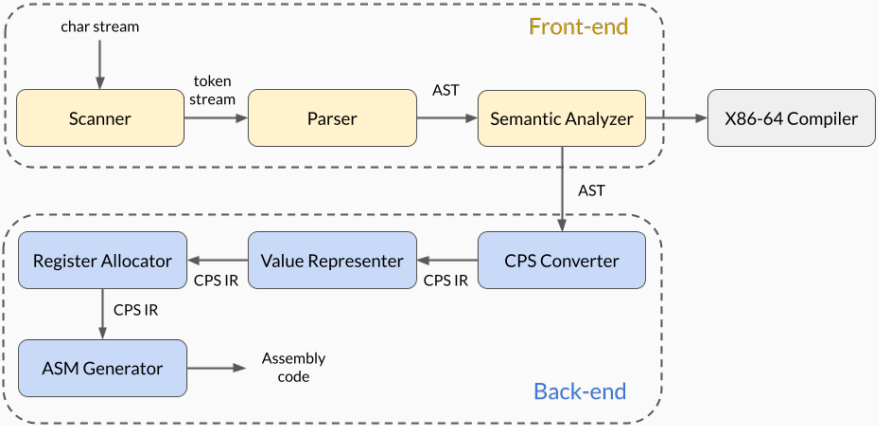
What are still missing?

- Higher level features: nested first-class functions, objects
- Code quality: optimizations

Intermediate Representations

- Intermediate representations (IR) or intermediate languages are the data structures used by the compiler to represent the program being compiled.
- Choosing a good IR is crucial, as many analyses and transformations (e.g. optimizations) are substantially easier to perform on some IRs than on others.
- Some non-trivial compilers actually use several IRs during the compilation process, and they tend to become more low-level as the code approaches its final form.

Compiler Architecture



We also have interpreters for the AST representation, the CPS IR, and the ASM language.

Intermediate Representations

The revised MiniScala compiler manipulates a total of four languages:

1) MiniScala the source language – the surface language in which programs are written,
2) MiniScala/AST – the parsed and desugared ASTs of MiniScala,
3) **MiniScala/CPS** – the main intermediate language, on which optimizations are performed,
4) ASM is the assembly language of the target (virtual) machine.

Intermediate Representations

The revised MiniScala compiler manipulates a total of four languages:

1) MiniScala the source language – the surface language in which programs are written,
2) MiniScala/AST – the parsed and desugared ASTs of MiniScala,
3) **MiniScala/CPS** – the main intermediate language, on which optimizations are performed,
4) ASM is the assembly language of the target (virtual) machine.

- There are interpreters for the last three languages, which are useful to check that a program behaves in the same way as it is undergoes transformation.
- These interpreters also serve as the semantics for their languages.

IR #1: MiniScala/CPS - A Functional IR

A functional IR is an intermediate representation that is close to a (very) simple functional programming language. Typical functional IRs have the following characteristics:

- all primitive operations (e.g. arithmetic operations) are performed on atomic values (variables or constants),
- the result of these operations is always named,
- variables cannot be re-assigned.

IR #1: MiniScala/CPS - A Functional IR

A functional IR is an intermediate representation that is close to a (very) simple functional programming language. Typical functional IRs have the following characteristics:

- all primitive operations (e.g. arithmetic operations) are performed on atomic values (variables or constants),
- the result of these operations is always named,
- variables cannot be re-assigned.

We will be using a CPS-based functional IR called MiniScala/CPS.

Some of these characteristics are shared with more mainstream IRs, like SSA used in GCC and LLVM.

Continuation-Passing Style (CPS)

RABBIT:

A Compiler for SCHEME

(A Dialect of LISP)

A Study in

Compiler Optimization

Based on Viewing
LAMBDA as RENAME
and
PROCEDURE CALL as GOTO

using the techniques of

Macro Definition of Control and Environment Structures
Source-to-Source Transformation
Procedure Integration
and
Tail-Recursion

Guy Lewis Steele Jr.

Massachusetts Institute of Technology

May 1978

Continuation-Passing Style (CPS)

A subset of SCHEME (rather than triples, for example) serves as the representation intermedieate between the optimized SCHEME code and the final output code; code is expressed in this subset in the so'called constinuation-passinq style. As a subset of SCHEME, it enjoys the same theoretical properties; one could even apply the same optimizer used on the input code to the intermediate code. However, the subset is so chosen that all temporary quantities are made manifest as variables, and no control stack is needed to evaluate it. As a result, this apparently applicative representation admits an imperative interpretation which permits easy transcription to fianl imperative machine code. These qualities suggest that an applicative language like SCHEME is a better candidate for an UNCOL than the more imperative candidates proposed to date.

Continuation-Passing Style (CPS)

What is continuation-passing style (CPS)?

A programming style or program representation where control is made explicit by passing “the rest of the computation” as an extra argument to functions.

Continuation-Passing Style (CPS)

What is continuation-passing style (CPS)?

A programming style or program representation where control is made explicit by passing "the rest of the computation" as an extra argument to functions.

Why compiling to CPS?

- CPS makes control flow explicit, which simplifies many analyses and transformations.
- Uniform representation of control flow: function calls, returns, jumps, exceptions, etc. are all represented uniformly as continuation invocations.

Uses of Continuations

Continuations are used for several purposes in our compiler for MiniScala/CPS:

1) They represent code blocks which can be “jumped to” from several locations.
2) They represent the code to execute after a function call.
3) Control transfer is represented as invoking continuations.

Every function gets a continuation as argument, which it must invoke with its return value.

Code Example

To illustrate the differences between the various IRs, we will use a program fragment to compute and print the greatest common divisor (GCD) of 2016 and 714.

The MiniScala version of that fragment could be:

```
def gcd(x: Int, y: Int): Unit =
  if (y == 0)
    printInt(x)
  else
    gcd(y, y % x);

gcd(2016, 714)
```

GCD in MiniScala/CPS

The MiniScala/CPS version of the GCD program fragment looks as follows:

```
// source code
def gcd(x: Int, y: Int): Unit =
  if (y == 0)
    printInt(x)
  else
    gcd(y, x % y);

gcd(2016, 714)
```

```
// CPS IR
def_f gcd(c, x, y) = {
  def_c ct() = { printInt(c, x) };
  def_c cf() = {
    val_p t = x % y;
    gcd(c, y, t);
  }
  val_l z = 0;
  if (y == z) ct() else cf()
};
def_c ce(x) = { val_l z = 0; halt(z) };
val_l x = 2016; val_l y = 714;
gcd(ce, x, y)
```

Local Continuations

A crucial notion in MiniScala/CPS is that of **local continuation**.

A local continuation is similar to a (local) function but with the following restrictions:

- continuations are *not* "first class citizens": they cannot be stored in variables or passed as arbitrary arguments. The only exception being the return continuation (described later),
- continuations never return, and must therefore be invoked in tail position only.

These restrictions enable continuations to be compiled much more efficiently than normal functions. This is the only reason why continuations exist as a separate construct.

MiniScala/CPS Grammar

- The grammar of MiniScala/CPS is defined as follows:

```
N ::= name
L ::= integer, character, boolean or unit literal
T ::= val_l N = L; T
    | val_p N = P(N, ...); T
    | def_c N(N, ...) = { T }; T
    | def_f N(N, ...) = { T }; T
    | N(N, ...)
    | if (N C N) N() else N()
    | halt(N)
P ::= '+' | '-' | '*' | '/' | '%' | ...
C ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
```

MiniScala/CPS Local Bindings

- Bind the name `n` to the literal value `l` in expression `e`:

  `val`$_{\ell}$ `n = l; e`

  The literal value can be an integer, a character, a boolean or the unit value.

MiniScala/CPS Local Bindings

- Bind the name `n` to the literal value `l` in expression `e`:

  $\texttt{val}_l$ `n = l; e`

  The literal value can be an integer, a character, a boolean or the unit value.

- Bind the name `n` to the result of the application of primitive `p` to the value of $\texttt{n}_1$, ... in expression `e`:

  $\texttt{val}_p$ `n = p(`$\texttt{n}_1$`, ...); e`

  The primitive `p` cannot be a logical (i.e. boolean) primitive, as such primitives are only meant to be used in conditional expressions - see later.

MiniScala/CPS Functions

- Define functions $\texttt{f}_1$, … with arguments $\texttt{n}_{1,1}$, … and return continuation $\texttt{c}_1$, … in expression `e`.

  $\texttt{def}_f\ \texttt{f}_1\texttt{(c}_1\texttt{, n}_{1,1}\texttt{, ...) = \{ e}_1\texttt{ \}; def}_f\texttt{f}_2\texttt{ = ...; ...; e}$

  The functions can be mutually recursive. The return continuation $\texttt{c}_1$ takes a single argument: the return value. Applying it is interpreted as returning from the function.

MiniScala/CPS Functions

- Define functions $\texttt{f}_1$, … with arguments $\texttt{n}_{1,1}$, … and return continuation $\texttt{c}_1$, … in expression `e`.

  $\texttt{def}_f$ $\texttt{f}_1\texttt{(c}_1\texttt{, n}_{1,1}\texttt{, ...) = \{ e}_1\texttt{ \}; def}_f\texttt{f}_2\texttt{ = ...; ...; e}$

  The functions can be mutually recursive. The return continuation $\texttt{c}_1$ takes a single argument: the return value. Applying it is interpreted as returning from the function.

- Apply function `f` to return continuation `c` and arguments $\texttt{n}_1$, …

  $\texttt{f(c, n}_1\texttt{, ...)}$

  The name `c` must either be bound by an enclosing $\texttt{def}_c$ or be the name of the return continuation of the current function.

MiniScala/CPS Local Continuations

- Define local continuations c_1 with arguments n_1, … and body e_1 in e:

  def$_c$ c_1(n$_1$, ...) = { e_1 }; e

  Interpretation: like a local function that never returns. e could contain other continuation definitions too.

MiniScala/CPS Local Continuations

- Define local continuations c_1 with arguments n_1, … and body e_1 in e:

  def$_c$ $c_1(n_1, \ldots) = \{ e_1 \}; e$

  Interpretation: like a local function that never returns. e could contain other continuation definitions too.

- Apply continuation c to the value of n_1, …

  $c(n_1, \ldots)$

  The name c must either be bound by an enclosing **def**$_c$ or be the name of the return continuation of the current function.

  - If c is a local continuation, $c(\ldots)$ can be seen as a jump with arguments.
  - If c refers to the current return continuation, $c(\ldots)$ can be seen as a return from the current function, with the given return value.

MiniScala/CPS Control Constructs

- Test whether the condition `p` is true for the value of $\texttt{n}_1$ and $\texttt{n}_2$, then apply continuation `ct` if it is, or `cf` if it isn't.

  `if` ($\texttt{n}_1$ `p` $\texttt{n}_2$) `ct()` **`else`** `cf()`

  Both continuation `ct` and `cf` must have no parameter. The primitive `p` must be a logical primitive.

  Note: **`if`** is a branching form of continuation invocation for parameterless continuations. It is therefore a conditional version of `c()`.

MiniScala/CPS Control Constructs

- Test whether the condition `p` is true for the value of n_1 and n_2, then apply continuation `ct` if it is, or `cf` if it isn't.

  `if` (n_1 `p` n_2) `ct() else cf()`

  Both continuation `ct` and `cf` must have no parameter. The primitive `p` must be a logical primitive.

  Note: **`if`** is a branching form of continuation invocation for parameterless continuations. It is therefore a conditional version of `c()`.

- Halts program execution:

  `halt``(n)`

  Exit with the value bound to `n` (which must be an integer).

MiniScala/CPS Syntactic Sugar

To make MiniScala/CPS programs easier to read and write, we allow to use the postfix and infix notations for the primitive operations.

```
val_l t = 1;
val_p x = -t + t
```

is equivalent to the actual representation:

```
val_l t = 1;
val_p mt = -t;
val_p x = +(mt, t)
```

Scopes of Continuation

The scoping rules of MiniScala/CPS are mostly the "obvious ones". The only exception is the rule for continuation variables, which are not visible in nested functions!

For example, in the following code:

```
def_c c0(r) = { printInt(ce, r); }
def_f f(c1, x) = {
  val_p t = x + x;
  c1(t)
}
```

c_0 is not visible in the body of `f`!

This guarantees that continuations are truly local to the function that defines them, and can therefore be compiled efficiently.

GCD in MiniScala/CPS

The MiniScala/CPS version of the GCD program fragment looks as follows:

```
// source code
def gcd(x: Int, y: Int): Unit =
  if (y == 0)
    printInt(x)
  else
    gcd(y, x % y);

gcd(2016, 714)
```

```
// CPS IR
def_f gcd(c, x, y) = {
  def_c ct() = { printInt(c, x) };
  def_c cf() = {
    val_p t = x % y;
    gcd(c, y, t);
  }
  val_l z = 0;
  if (y == z) ct() else cf()
};
def_c ce(x) = { val_l z = 0; halt(z) };
val_l x = 2016; val_l y = 714;
gcd(ce, x, y)
```

Translating MiniScala to MiniScala/CPS

The translation from MiniScala/AST to MiniScala/CPS is specified as a function denoted by $[\![\cdot]\!]$ taking two arguments:

1) `T`, the MiniScala/AST term to be translated,
2) `C`, the context, a MiniScala/CPS term containing a hole into which a name bound to the value of the translated term has to be plugged.

Translating MiniScala to MiniScala/CPS

The translation from MiniScala/AST to MiniScala/CPS is specified as a function denoted by $[\![\cdot]\!]$ taking two arguments:

1) `T`, the MiniScala/AST term to be translated,
2) `C`, the context, a MiniScala/CPS term containing a hole into which a name bound to the value of the translated term has to be plugged.

This function is written in a "mixfix" notation, as follows:

$[\![$`T`$]\!]$ `C`

The translation function must return a MiniScala/CPS term.

Translation Context

The translation context is a MiniScala/CPS term representing the partial translation of the MiniScala expression surrounding the one being translated.

- This term contains a single hole, written □, representing the currently unknown *name* that will be bound to the value of the expression being translated.
- The hole of a context `C` must eventually be filled with some name `n`, written as `C[n]`.

  For example, `f(□)[m]` denotes plugging name `m` into the context `f(□)`, resulting in the term `f(m)`.

Context Representation

Translation rules often build contexts that include other contexts. One (fictional) example to translate application `e1(e2)` could be:

$[\![\texttt{e}_1]\!]$ `(`$[\![\texttt{e}_2]\!]$ `□(□))`

Context Representation

Translation rules often build contexts that include other contexts. One (fictional) example to translate application `e1(e2)` could be:

$[\![\texttt{e}_1]\!]\ ([\![\texttt{e}_2]\!]\ \square(\square))$

In such situations, using the anonymous hole ($\square$) is ambiguous. To resolve the ambiguity, we *represent contexts as meta-functions* taking a single (named) argument. The above is therefore written as:

$[\![\texttt{e}_1]\!]\ \lambda \texttt{v}_1(\texttt{v}_1)([\![\texttt{e}_2]\!]\ \lambda \texttt{v}_2(\texttt{v}_1(\texttt{v}_2)))$

where `v1`/`v2` is the variable in the meta-variable representing the translation result of `e1`/`e2`.

Context Representation

Translation rules often build contexts that include other contexts. One (fictional) example to translate application `e1(e2)` could be:

$[\![\texttt{e}_1]\!]\ (\texttt{[}\![\texttt{e}_2]\!]\ \square(\square))$

In such situations, using the anonymous hole ($\square$) is ambiguous. To resolve the ambiguity, we *represent contexts as meta-functions* taking a single (named) argument. The above is therefore written as:

$[\![\texttt{e}_1]\!]\ \lambda \texttt{v}_1(\texttt{[}\![\texttt{e}_2]\!]\ \lambda \texttt{v}_2(\texttt{v}_1(\texttt{v}_2)))$

where `v1`/`v2` is the variable in the meta-variable representing the translation result of `e1`/`e2`.

With such a representation of contexts, filling the hole of context `C` with name `n` is performed by *meta-function application*, written `C[n]`.

The Translation In Scala

In Scala (our meta-language of the MiniScala project), the translation function $[\![\cdot]\!]$ is defined as a function with the following signature:

```
def MiniScalaToCPS(t: MSTree, c: Symbol => CPSTree): CPSTree
```

In the body of that function, plugging the context `c` with a name (i.e. a Symbol) bound to a Scala value named `n` is performed by Scala function application:

```
c(n)
```

To clarify the presentation, MiniScala terms appear in orange, MiniScala/CPS terms in purple, and meta-terms in black.

MiniScala/AST to MiniScala/CPS Translation (1)

Note: in the following expressions, all underlined names are fresh.

$[\![\texttt{n}]\!]\ \texttt{C} = \texttt{C[n]}$
where n is an identifier for immutable variable

$[\![\texttt{l}]\!]\ \texttt{C} = \texttt{val}_{\ell}\ \underline{\texttt{n}}\ \texttt{= l; C[n]}$
where l is a literal

MiniScala/AST to MiniScala/CPS Translation (1)

Note: in the following expressions, all underlined names are fresh.

$[\![\texttt{n}]\!]\ \texttt{C} = \texttt{C[n]}$
where n is an identifier for immutable variable

$[\![\texttt{l}]\!]\ \texttt{C} = \texttt{val}_l\ \underline{\texttt{n}}\ \texttt{= l; C[n]}$
where l is a literal

$[\![\ \texttt{val}\ \texttt{n}_1\ \texttt{=}\ \texttt{e}_1\texttt{;}\ \texttt{e}\ ]\!]\ \texttt{C} =$
$[\![\texttt{e}_1]\!](\lambda \texttt{v}\ (\texttt{val}_p\ \texttt{n}_1\ \texttt{= id(v);}\ [\![\texttt{e}]\!]\ \texttt{C}))$

MiniScala/AST to MiniScala/CPS Translation (1)

Note: in the following expressions, all underlined names are fresh.

$[\![\texttt{n}]\!]\ \texttt{C} = \texttt{C[n]}$
where n is an identifier for immutable variable

$[\![\texttt{l}]\!]\ \texttt{C} = \texttt{val}_l\ \underline{\texttt{n}}\ \texttt{= l; C[n]}$
where l is a literal

$[\![\ \texttt{val}\ \texttt{n}_1\ \texttt{=}\ \texttt{e}_1\texttt{; e}\ ]\!]\ \texttt{C} =$
$[\![\texttt{e}_1]\!](\lambda \texttt{v}\ (\texttt{val}_p\ \texttt{n}_1\ \texttt{= id(v);}\ [\![\texttt{e}]\!]\ \texttt{C}))$

$[\![\ \texttt{var}\ \texttt{n}_1\ \texttt{=}\ \texttt{e}_1\texttt{; e}\ ]\!]\ \texttt{C} =$
$\texttt{val}_l\ \underline{\texttt{s}}\ \texttt{= 1;}$
$\texttt{val}_p\ \texttt{n}_1\ \texttt{= block-alloc-var(s);}$
$\texttt{val}_l\ \underline{\texttt{z}}\ \texttt{= 0;}$
$[\![\texttt{e}_1]\!](\lambda \texttt{v}\ (\texttt{val}_p\ \underline{\texttt{d}}\ \texttt{= block-set(n}_1\texttt{, z, v);}\ [\![\texttt{e}]\!]\ \texttt{C}))$

MiniScala/AST to MiniScala/CPS Translation (2)

```
⟦ n1 = e1 ⟧ C =
  val_l z = 0;
  ⟦e1⟧(λv (val_p d = block-set(n1, z, v); C[v] ))

⟦n⟧ C =
  val_l z = 0;
  val_p v = block-get(n, z); C[v]
  where n is an identifier for mutable variable
```

MiniScala/AST to MiniScala/CPS Translation (2)

$$[\![\ n_1 = e_1\ ]\!]\ C =$$
$$\quad val_l\ z = 0;$$
$$\quad [\![e_1]\!](\lambda v\ (val_p\ d = block\text{-}set(n_1, z, v);\ C[v]\ ))$$

$$[\![n]\!]\ C =$$
$$\quad val_l\ z = 0;$$
$$\quad val_p\ v = block\text{-}get(n, z);\ C[v]$$
$$\quad \textit{where } n \textit{ is an identifier for mutable variable}$$

$$[\![\ def\ f_1(n_{1,1}: \_, \ldots) = e_1;\ def\ \ldots\ ;\ e]\!]\ C =$$
$$\quad def_f\ f_1(c, n_{1,1}, \ldots) = \{$$
$$\quad\quad [\![e_1]\!](\lambda v(c(v)))$$
$$\quad \};$$
$$\quad def_f\ \ldots;$$
$$\quad [\![e]\!]\ C$$

MiniScala/AST to MiniScala/CPS Translation (3)

$$
\begin{aligned}
&[\![\ \texttt{e(}e_1\texttt{, }e_2\texttt{, ...)}\ ]\!]\ \texttt{C} = \\
&\quad [\![\texttt{e}]\!](\lambda \texttt{v}([\![e_1]\!](\lambda \texttt{v}_1([\![e_2]\!](\lambda \texttt{v}_2(... \\
&\quad \texttt{def}_c\ \underline{\texttt{c}}(\underline{\texttt{r}})\ \texttt{= \{ C[r] \};} \\
&\quad \texttt{v(c, }\texttt{v}_1\texttt{, }\texttt{v}_2\texttt{ ...)})))))))
\end{aligned}
$$

MiniScala/AST to MiniScala/CPS Translation (3)

$$[\![\ \texttt{e(}e_1\texttt{, }e_2\texttt{, ...)}\ ]\!]\ \texttt{C} =$$
$$[\![\texttt{e}]\!](\lambda \texttt{v}([\![e_1]\!](\lambda v_1([\![e_2]\!](\lambda v_2(\texttt{...}$$
$$\texttt{def}_c\ \underline{\texttt{c}}(\underline{\texttt{r}})\ \texttt{= \{ C[r] \};}$$
$$\texttt{v(c, }v_1\texttt{, }v_2\ \texttt{...})))))))$$

$$[\![\ \texttt{if (p(}e_1\texttt{, ...)) }e_2\ \texttt{else }e_3\ ]\!]\ \texttt{C} =$$
$$\texttt{def}_c\ \underline{\texttt{c}}(\underline{\texttt{r}})\ \texttt{= \{ C[r] \};}$$
$$\texttt{def}_c\ \underline{\texttt{ct}}()\ \texttt{= \{ }[\![e_2]\!](\lambda v_2(\texttt{c(}v_2\texttt{))) \};}$$
$$\texttt{def}_c\ \underline{\texttt{cf}}()\ \texttt{= \{ }[\![e_3]\!](\lambda v_3(\texttt{c(}v_3\texttt{))) \};}$$
$$[\![e_1]\!](\lambda v_1(\texttt{... (if (p(}v_1\ \texttt{...)) ct else cf)))}$$
where p is a logical primitive

MiniScala/AST to MiniScala/CPS Translation (4)

$$
\begin{array}{l}
[\![\ \texttt{if (}e_1\texttt{) } e_2 \texttt{ else } e_3\ ]\!]\ \texttt{C} = \\
\quad \texttt{def}_c\ \underline{\texttt{c}}\texttt{(}\underline{\texttt{r}}\texttt{) = \{ C[r] \};} \\
\quad \texttt{def}_c\ \underline{\texttt{ct}}\texttt{() = \{ }[\![e_2]\!](\lambda v_2(\texttt{c}(v_2)))\texttt{ \};} \\
\quad \texttt{def}_c\ \underline{\texttt{cf}}\texttt{() = \{ }[\![e_3]\!](\lambda v_3(\texttt{c}(v_3)))\texttt{ \};} \\
\quad \texttt{val}_l\ \underline{\texttt{f}}\texttt{ = false;} \\
\quad [\![e_1]\!](\lambda v_1(\texttt{if (}v_1 \neq \texttt{f) ct else cf))}
\end{array}
$$

MiniScala/AST to MiniScala/CPS Translation (4)

$$
\begin{array}{l}
[\![\ \texttt{if (}e_1\texttt{) } e_2 \texttt{ else } e_3\ ]\!]\ \texttt{C} = \\
\quad \texttt{def}_c\ \underline{\texttt{c}}(\underline{\texttt{r}}) = \{\ \texttt{C[r]}\ \}; \\
\quad \texttt{def}_c\ \underline{\texttt{ct}}() = \{\ [\![e_2]\!](\lambda v_2(\texttt{c}(v_2)))\ \}; \\
\quad \texttt{def}_c\ \underline{\texttt{cf}}() = \{\ [\![e_3]\!](\lambda v_3(\texttt{c}(v_3)))\ \}; \\
\quad \texttt{val}_l\ \underline{\texttt{f}} = \texttt{false}; \\
\quad [\![e_1]\!](\lambda v_1(\texttt{if } (v_1 \neq \texttt{f}) \texttt{ ct else cf}))
\end{array}
$$

$$
\begin{array}{l}
[\![\ \texttt{while (}e_1\texttt{) } e_2\texttt{; } e_3\ ]\!]\ \texttt{C} = \\
\quad \texttt{def}_c\ \underline{\texttt{loop}}() = \{ \\
\quad\quad \texttt{def}_c\ \underline{\texttt{c}}() = \{\ [\![e_3]\!]\ \texttt{C}\ \}; \\
\quad\quad \texttt{def}_c\ \underline{\texttt{ct}}() = \{\ [\![e_2]\!](\lambda v(\texttt{loop}()))\ \}; \\
\quad\quad \texttt{val}_l\ \underline{\texttt{f}} = \texttt{false}; \\
\quad\quad [\![e_1]\!](\lambda v(\texttt{if } (v \neq \texttt{f}) \texttt{ ct else c})) \\
\quad \}; \\
\quad \texttt{loop}()
\end{array}
$$

MiniScala/AST to MiniScala/CPS Translation (5)

$[\![$ p(e_1, e_2, ...) $]\!]$ C =
  $[\![$ if (p(e_1, e_2, ...)) true else false $]\!]$ C
  where p is a logical primitive

$[\![$ p(e_1, e_2, ...) $]\!]$ C =
  left as an exercise
  where p is not a logical primitive

Initial Context

In which context should a complete program be translated?

The simplest answer is a context that halts execution with an exit code of 0 (no error), that is:

λv(val$_l$ z = 0; halt(z))

An alternative would be to do something with the value v produced by the whole program, e.g. use it as the exit code instead of 0, print it, etc.

Example

Let k_0 be the initial context λv(val$_l$ z $=$ 0; halt(z)).

```
  ⟦ f(g(1, 2)) ⟧ k0
{ f(g(1, 2)) is a function application }
 =  ⟦f⟧(λv(⟦g(1, 2)⟧(λv1
    def_c c1(r) = { k0[r] };
    v(c1, v1))))
 = ???
```

Where Are We?

Take-home exercise: translate the MiniScala `gcd` function using the rules we have seen so far.

Today we have seen:

- A transformation from MiniScala to its CPS intermediate representation.
- Project 4 will ask you to implement (part of) the transformation.

Next time:

- Improve the CPS translation to produce better code!