Building a high-level untyped functional language

Before we begin...

Welcome back to the programming languages course! Just for the sake of better mutual understanding, let's give that handout syntax once again here:

- Whatever text is written in a regular, black-on-white font is essential and couldn't be skipped. This is usually something you should take your time and read through carefully. If it's **bold** or *italic*, doubly so!
- If text appears in a darker grey box with a **"Sidenote"** caption, it's usually an interesting sidenote on the topic in question. These are usually there to tell you things you would come to realize for yourselves at some later point.
- If text appears in a dashed, lighter grey box, with a caption like **"Don't drink too much Kool-aid"**, this is usually an example, counter-example, question or something to spend some time thinking about. We're interested in your opinions, if you'd give them.
- Code segments and equations are given in a **`monospaced font`**, with explanations usually following or preceding them.
- Anything else is a comment.

1 Overview

Having learned Scala for the previous assignment, we can use this new language as a template for our first escapade into making one of our own.

We do our best in this part of the course to create a language that looks very similar to Scala, and to show how to model its semantics. Out of this, you get first-hand experience building a complex high-level language, and we're leveraging the fact that Scala's expected behaviours are still fresh in your mind.

> ***Sidenote:*** Even though `SimpleScala` will initially be quite a bit different (lacking a type system and such), future assignments will bring us closer step by step:
>
> - with assignment 2, we will have an untyped high-level interpreter for `SimpleScala` ;
> - assignment 3 will introduce a simple type-system;
> - assignment 5 will add a polymorphic type-system and enable us to do generics;
>
> Somewhere along the way, we will try to write `SimpleScala` for a while, to see that it actually can be used as a real language. You will be pleasantly surprised.

First, we'll see how similar `SimpleScala` will be to Scala itself syntactically. Then, we will follow what was presented in the first handout (about describing semantics; look it up if you're not too sure), by building a **state description** and **state transition** table. Here, we'll introduce a new concept that's going to prove useful for more complex states: **continuations**. We haven't used continuations in the previous handout as the semantics there were simple enough not to require it, however, if we want any feature that you all know and love from any language more readable than MIPS, that's the way to go.

2 Syntax

> Controlling complexity is the essence of computer programming.
>
> Brian Kernighan

Syntactically, `SimpleScala` tries to be similar to Scala, where this is possible. To get to deserve being named "simple", `SimpleScala` cuts some of the features, and introduces some other features later on. From a high-end perspective, the biggest change is that `SimpleScala` is fully functional, while Scala is an object-oriented and functional hybrid. There are no **classes** or **traits** in our language, and the main building block is a function.

In fact, our whole program is a list of function definitions followed by a single expression which will be executed and (hopefully) use those defined functions for something useful!

> ***Sidenote:*** You might realize that this is exactly how program execution in `C` works: the entry point is the code inside the function `main`, which calls other functions defined in `main`'s range. `C`, however, isn't a functional language in the traditional sense (except in crazy musings...) and a better example might be something from the `ML` family, `Haskell` or `Lisp`.

2.1 Formalization

As shown in handout 1, we give a formal writeup of the syntax of `SimpleScala` . Here, we use $\overrightarrow{something}$ to mean "this is a list of *something*s".

$$x \in Variable \qquad str \in String \qquad b \in Boolean \qquad i \in \mathbb{Z} \qquad n \in \mathbb{N}$$
$$fn \in FunctionName \qquad cn \in ConstructorName$$

$$\begin{aligned}
e \in Exp ::= {} & x \mid str \mid b \mid i \mid \textbf{unit} \mid e_1 \oplus e_2 \\
& \mid x \Rightarrow e \mid fn(e) \mid e_1(e_2) \\
& \mid \textbf{if } (e_1)\ e_2\ \textbf{else } e_3 \\
& \mid \{\overrightarrow{val}\ e\} \\
& \mid (\ \vec{e}\ ) \mid e._n \\
& \mid cn(e) \mid e\ \textbf{match } \{\overrightarrow{case}\} \\
val \in Val ::= {} & \textbf{val } x = e \\
case \in Case ::= {} & \textbf{case } cn(x) \Rightarrow e \mid \textbf{case } (\ \vec{x}\ ) \Rightarrow e \\
\oplus \in Binop ::= {} & + \mid - \mid * \mid / \mid \wedge \mid \vee \mid < \mid \leq \\
def \in Def ::= {} & \textbf{def } fn(x) = e \\
prog \in Program ::= {} & \overrightarrow{def}\ e
\end{aligned}$$

The thing not written in the syntax is that $x \in Variable$ and $fn \in FunctionName$ **always** begin with a lower case, while $cn \in ConstructorName$ **always** start with an upper case.

2.2 Simplifications

There are several points of simplification that were made in the design of untyped `SimpleScala` that make it distinct from Scala. All of these will be briefly elaborated on before giving an outline of the syntax and semantics.

2.2.1 Tuples

Tuples are a part of the syntax, marked with `( a, b, ..., z )`. Tuple access is the same as in Scala, namely `tup._n`, where `n` is a natural number. Tuples can be matched upon, exactly as in Scala. Tuples must contain at least two elements.

2.2.2 Blocks

Val-expressions come only at the beginning of a block, followed by a single expression. This constitutes as a block, surrounded by a pair of braces. Blocks must contain at least one val-expression.

> ***Sidenote:*** The inability to put val-expressions anywhere except the start of a block might seem daunting and limiting, but you can think of it as preparation for some process: we prepare all the pieces and then do a computation with it. This is very common in functional programming, basically following this quote:
>
> "Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity." – David Gelernter

2.2.3 Functions

Functions always take one parameter, but as anonymous functions are part of the language, we can do the following:

```
def foo(bar) = baz ⇒ ...
```

To call these curried functions, we can call this precisely as in Scala:

```
foo(bar)(baz)
```

If you're not a fan of these, you are always free to pass a single tuple in, holding multiple arguments inside, as shown in the following example:

```
def foo(barbaz) = {
    val bar = barbaz._1
    val baz = barbaz._2
    ...
}
```

Another way to do the same thing would be to use pattern-matching on the tuple:

```
def foo(barbaz) = {
    barbaz match {
      case (bar, baz) ⇒ ...
    }
}
```

These two pieces of code have the same meaning.

To call a function with tuple arguments, we need to pass a tuple in:

```
foo((bar, baz))
```

> ***Don't drink too much Kool-Aid:*** *Why do you think we made all the argument lists have only one argument? Do you have an idea of how much more complexity it would add to have an arbitrary number of arguments for functions and constructors? An educated guess? Post it to Piazza if you want to talk about it!*

2.2.4 Data Constructors

Data constructors are tagged value, so to speak. They represent a value with a name tag on it. This tag makes it distinguishable, and even though it has no meaning by itself, it can be pattern matched upon. This will be used instead of Scala's `case classes`. We can construct a data constructor just by declaring it, like `Nil(unit)` in the example above.

3 Semantics

As for the semantics (remember, the **meaning** of our programs), this is where the differences really show. In this section, we will be talking about three topics, some of which might be new, introduced by our more complex semantics (thus, more important for you, the reader!):

- **Values**, which give different syntactic elements different domains, and thus specialized semantics;
- **Continuations** – a way to make computations splittable, which will make life easy for us;
- **States**, which we discussed previously (in handout 1), whose transition is what semantics is all about

Let's jump right in.

3.1 Values

Let's first do the possible values that can appear in our programs. First, because our programs now have variables (which are just named values), we define an **environment** that fits that description: a map between variables and values.

$$\rho \in Env = Variable \to Value$$

Now, we'll be defining separate value domains for strings, booleans, integers, unit, closures (anonymous functions), constructors and tuples. These will map into what our variables can hold in an environment.

$$\begin{aligned} v \in Value = \ & \textbf{strV}(String) + \\ & \textbf{boolV}(Boolean) + \\ & \textbf{intV}(\mathbb{Z}) + \\ & \textbf{unitV} + \\ & \textbf{closureV}(Variable \times Exp \times Env) + \\ & \textbf{constructorV}(ConstructorName \times Value) + \\ & \textbf{tupleV}(\overrightarrow{Value}) \end{aligned}$$

The extra **V** at the end of the semantic names are there to distinguish **values** from other semantic elements that we'll introduce soon. Values, as they are here, don't exist in the MIPS assembly semantics, because of how simple it is.

> ***Kyle rants:*** *Well, technically, values do exist in the MIPS assembly, it's just that there's only **one** kind of value. The big difference here is that we have different kinds of values (domains), and so we need semantic tags to distinguish them easily.)*

With every layer of indirection we introduce (every single thing that makes our language more complex and, in the end, useful), we also introduce new elements to our semantics.

> ***Sidenote:*** As a quick notation helper, you can read + as "or" and × as "and" when reading semantic descriptions. As always, we use $\overrightarrow{things}$ to mark a list of *things*.
>
> You might notice that all the semantic values are basically wrappers around syntactic AST nodes. This is correct: semantics gives meaning to syntax; it wraps certain elements of the syntax into a layer of meaning.

One can imagine that in a language that has complicated arithmetic expressions, these computations won't be as easy to do as with simple **addu** et al. in MIPS. As a motivating example, imagine that we need to calculate the value of this expression:

$$3 + 4 * 2$$

To do this calculation, we have to start at the left most operand, resolve it, and then find out what the right operand is. In the addition above, the right operand might be tricky to find (it might seem that it's the expression 4 while it's actually $4 * 2$), and it would be good for us to remember what we were doing previously, so that once we discover the full right operand, we can return and continue our addition.

These situations are what we will be using **continuations**, and a **continuation stack** for.

3.2 Continuations

Continuations are a very general mechanism and can be used to implement almost all of the control flow mechanisms (think: if, while, etc.) of a language! They are best learned through practice, but if you can't wait, here's a homegrown example.

This was given by one Luke Palmer and became a rather famous thing, called the **sandwich** example:

A computation starts...

Say you're in the kitchen in front of the fridge, thinking about eating a sandwich. Unfortunately, there's no sandwich around. You make a continuation right there (a sticky note) and put it on the fridge. You will look back at this to remember what you were thinking, and the continuation says "Eat sandwich!"

We remember where we were...

Then you get some turkey and bread out of the refrigerator and make yourself a sandwich. Once that action is done, you have nothing else to do, so you look to the fridge, to see if you forgot something and oh! There's a continuation![1]

We go back and continue!

You invoke the continuation on top of the stack of glued sticky-notes, and return to the moment when you were standing in front of the fridge, thinking about eating a sandwich. Fortunately, there's a sandwich on the counter, so you eat it.

3.2.1 Implementation

$$\begin{aligned}\kappa \in \mathit{Kont} = {} & \textbf{binopLeftK}(\mathit{Binop} \times \mathit{Exp}) + \\ & \textbf{binopRightK}(\mathit{Value} \times \mathit{Binop}) + \\ & \textbf{restoreK}(\mathit{Env}) + \\ & \textbf{anonFunLeftK}(\mathit{Exp}) + \\ & \textbf{anonFunRightK}(\mathit{Variable} \times \mathit{Exp} \times \mathit{Env}) + \\ & \textbf{namedFunK}(\mathit{FunctionName}) + \\ & \textbf{ifK}(\mathit{Exp} \times \mathit{Exp}) + \\ & \textbf{blockK}(\mathit{Variable} \times \overrightarrow{\mathit{Val}} \times \mathit{Exp}) + \\ & \textbf{tupleK}(\overrightarrow{\mathit{Exp}} \times \overrightarrow{\mathit{Value}}) + \\ & \textbf{accessK}(\mathbb{N}) + \\ & \textbf{constructorK}(\mathit{ConstructorName}) + \\ & \textbf{matchK}(\overrightarrow{\mathit{Case}})\end{aligned}$$

As with the values ending with `V`, we can see that continuations end with a `K`. The reasons behind the usage of the letter "K" and writing "Kont" instead of "Cont" for the shorter name of continuations is historic (and geographic, coming from the Netherlands in the 1960s). Never lose sight of what a continuation is: instead of presenting what we're doing now, it presents what we now *know we will definitely have to do* in the future! Continuations are put onto the continuation stack with a reminder of where we were, as we set off to do something else.

3.3 States

Our states in `SimpleScala` are defined as a product of three factors: our current term, which is either a semantic **value** (something that has known meaning) or a syntactic **expression** (something that we have to figure out); the environment that holds our variable bindings; and the continuation stack which remembers what we were doing before. If this stack is empty, the current term is what our program produced at its end (and there is nothing left to do, so we're done)!

$$t \in \mathit{Term} = \mathit{Exp} + \mathit{Value}$$

$$\varsigma \in \mathit{State} = \mathit{Term} \times \mathit{Env} \times \overrightarrow{\mathit{Kont}}$$

Because we distinguish functions from variables by nothing else than finding them defined with **def**, we have accumulated all the functions before running the semantics into a $defs$ mapping ($\mathit{Variable} \times \mathit{Exp}$, because we always have one argument and return an

[1] You are obviously a goldfish.

expression, but you knew that):

$$defs \in Defs = FunctionName \to (Variable \times Exp)$$

Now that we know what our states will be, we can talk about transitions. We'll break the table down into pieces, for ease of use (as it's quite large). First, let's see what happens if the current term is a syntactic expression (so, something that as of yet has no **meaning**). We'll have to cover all of the syntactic terms, of course, but some of them are less productive than others. Baby steps.

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
1	str		**strV**(str)	ρ	$\vec{\kappa}$
2	b		**boolV**(b)	ρ	$\vec{\kappa}$
3	i		**intV**(i)	ρ	$\vec{\kappa}$
4	**unit**		**unitV**	ρ	$\vec{\kappa}$
5	x	$x \in \texttt{keys}(\rho), v = \rho(x)$	v	ρ	$\vec{\kappa}$

What the first four rules are basically saying is that a certain kind of syntactic element is mapped exactly to a certain kind of semantic element. Everything else stays the same: there are no remembered actions and there are no changes to the environment.

The fifth rule (which happens when we access a variable), has a premise, saying that "if the current environment ρ has a key called x, let v be the value of that key. If these premises are okay, the new term we'll have is that v, and nothing else changes."

> ***Don't drink too much Kool-Aid:*** *This might be a bit hard to fathom at first. Try giving it a few reads and try going through an example by hand once you finish this handout. The semantic transition table should be traced, not remembered.*

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
6	$e_1 \oplus e_2$		e_1	ρ	**binopLeftK**$(\oplus, e_2) :: \vec{\kappa}$
7	v	$\vec{\kappa} =$ **binopLeftK**$(\oplus, e) :: \vec{\kappa_2}$	e	ρ	**binopRightK**$(v, \oplus) :: \vec{\kappa_2}$
8	v_2	$\vec{\kappa} =$ **binopRightK**$(v_1, \oplus) :: \vec{\kappa_2}$, $v_3 = \texttt{evalOp}(v_1, \oplus, v_2)$	v_3	ρ	$\vec{\kappa_2}$

The next three rules have to do with binary operations, and they will introduce the first case where we have the usage of a continuation: if we encounter a $e_1 \oplus e_2$ syntactic expression, we will try to resolve e_1, while remembering that we have to go back to **binopLeftK**$(\oplus, e_2)$, once we're done with it.

Then, if we found a value v as our current term, and the head of the continuation stack is a **binopLeftK**$(\oplus, e)$, try to resolve that e from the continuation, while remembering whatever it is we have as v in a **binopRightK**$(v, \oplus)$.

If we now come to a state at some point where we have a value v_2 as our current term, and the head of the continuation stack is a **binopRightK**$(v_1, \oplus)$ (don't mind the names and indices, they are arbitrary), we will deduce a value v_3 that is $\oplus$ executed on v_1 and v_2, represented by the helper function `evalOp`. This v_3 is now the next current value and our continuation stack is left without the binary operation, as we've completed it.

> ***Sidenote:*** The names, **binopLeftK** and **binopRightK**, are kind of weird:
>
> - **binopLeftK** indicates that the leftmost expression is currently being evaluated, and it stores what we'll need to evaluate next, once that is done
> - **binopRightK** indicates that the rightmost expression is currently being evaluated, and it saves what the left side gave us for after the right side is done
>
> After both sides are done, the meaning of the operation itself can be reasoned about.

Moving on, we reach function calls:

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
9	$x \Rightarrow e$		**closureV**(x, e, ρ)	ρ	$\vec{\kappa}$
10	$e_1(e_2)$		e_1	ρ	**anonFunLeftK**$(e_2) :: \vec{\kappa}$
11	**closureV**(x', e', ρ')	$\vec{\kappa} =$ **anonFunLeftK**$(e) :: \vec{\kappa_2}$	e	ρ	**anonFunRightK**$(x', e', \rho') :: \vec{\kappa_2}$
12	v	$\vec{\kappa} =$ **anonFunRightK**$(x', e', \rho') :: \vec{\kappa_2}$	e'	$\rho'[x' \mapsto v]$	**restoreK**$(\rho) :: \vec{\kappa_2}$
13	$fn(e)$	$fn \in \texttt{keys}(defs)$	e	ρ	**namedFunK**$(fn) :: \vec{\kappa}$
14	v	$\vec{\kappa} =$ **namedFunK**$(fn) :: \vec{\kappa_2}$, $(x \cdot e) = defs(fn)$	e	$\{\}[x \mapsto v]$	**restoreK**$(\rho) :: \vec{\kappa_2}$

If you followed the logic from the binary operation example, this is mostly the same. Rule 9 defines what happens when we meet a syntactic closure, $x \Rightarrow e$, very much in the spirit of rules 1-4. Rule 10 is quite free-form, so it explores what e_1 is, while preparing us for an anonymous function call, saving the argument e_2 in that continuation.

Once we figure out that e_1 indeed was a closure, we come to rules 11 and 12, which repeat the logic of **binopLeft** and **binopRight** almost to the letter. You might notice a difference here in rule 12's ρ_{new} column, as we encounter the $\rho'[x' \mapsto v]$ notation. This means "update ρ' with the mapping $x' \mapsto v$".

Rule 13 introduces actual named functions, putting a **namedFunK** onto the continuation stack while resolving the function argument: rule 14 *calls* the function once the argument is resolved. This, however, is quite different, looking at ρ and $\overrightarrow{\kappa_{new}}$: function calls create new environments, with only the function argument x mapped to the value v, and a continuation **restoreK** that **saves** the previous environment, so that we might return to it once the function call is done.

The rule to deal with the actual restoring of a state is the following:

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
15	v	$\vec{\kappa} =$ **restoreK**$(\rho') :: \vec{\kappa_2}$	v	ρ'	$\vec{\kappa_2}$

This is basically a return: if we only have a semantic value as the current term, and there's a **restoreK** next, we have our wanted result, so we bring it back: the same value is passed as the next current term, but the environment is replaced with the one from the continuation (the environment before the function call).

The rest of the table follows closely, and a big part of the second assignment is turning it into actual code, so we'll leave it to you. The full table follows at the end of this handout, but before that, there are several helper functions used here that need at least some explaining.

3.4 Helpers

3.4.1 Retrieve Keys from Map

A simple **keys** function is used in multiple rules to retrieve the set of keys of a given map. This does exactly what it tells you.

3.4.2 Reverse a List

The **reverse** function reverses the contents of a list. This is useful for one rule that work with tuples.

3.4.3 Initial State

The **initialState** helper creates the initial program state, given only the program. The current term is determined by the program expression, and the environment and continuation stack are both made empty.

3.4.4 Binary Operation Evaluation

The one helper function already used in the handout is **evalOp**$(\mathbf{a}, \oplus, \mathbf{b})$, which evaluates a binary arithmetic operation or relation on two domains. This mainly follows the following constraints:

- **a** and **b** have to be of the same domain of value, e.g. both have to be **IntV**
- the $\leq$ and $<$ relations can be executed only on **IntV** and are exactly as expected

- the logical operations $\vee$ and $\wedge$ can be executed only on **BoolV**
- the meaning of the arithmetic operations $+, -, \times$ over **IntV** are integer addition, subtraction, and multiplication
- the meaning of the arithmetic operation $\div$ is integer division, for any non-zero **b**
- the meaning of the operation + over **StringV** is string concatenation
- any other combination of **a**, $\oplus$ and **b** fails by getting stuck

3.4.5 Tuple and Case Lookup

Because tuples and constructors are matchable in different ways, there need to be some functions around that do the lookup for the appropriate matches. In our case, we give the functions `tupleAccess`, `constructorCaseLookup` and `tupleCaseLookup` for the appropriate situations.

The `tupleAccess` helper accesses the n^{th} element of a tuple. As we index tuples with natural numbers (look at the syntax on page 2), the first index is 1.

The `constructorCaseLookup` helper gives us a matching case, given a constructor name and a list of cases. In a similar light, `tupleCaseLookup` gives us a matching environment and expression (therefore, a case), given a list of values, environment, and a list of cases to match through.

4 Done!

Okay, phew! You have just been instructed on how to write a high-level language semantics! It may not be the next C++, but rest assured, this is a big step! Let's see what we covered in this handout:

- We learned how to write syntax and semantics rules for languages with several different value domains
- We saw that semantics basically wraps around the syntax, adding meaning to it
- We went through a few samples of continuations, which serve the purpose of stopping a computation if we're not ready for it, and enabling us to continue with it once we are
- We saw that we can sometimes reuse the same general concept to implement a bunch of different things, like using continuations to implement both binary operations, function calls and a lot of the other things in our language

We're getting to the fun bit. You now have a language of the same power as Scala. It's Turing-complete (prove it to us, we dare you!), and you can write programs in it and run them through the parser to see the result. It lacks some sophistication, but that's why we have assignment 3, and adding a type-checker to the language you just created!

P.S. The table starts out nice and sorted, and then slowly descends into chaos. Don't implement it without thinking, connect the dots as you go. Continuations lead one to another!

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
1	str		**strV**(str)	ρ	$\vec{\kappa}$
2	b		**boolV**(b)	ρ	$\vec{\kappa}$
3	i		**intV**(i)	ρ	$\vec{\kappa}$
4	**unit**		**unitV**	ρ	$\vec{\kappa}$
5	x	$x \in \texttt{keys}(\rho), v = \rho(x)$	v	ρ	$\vec{\kappa}$
6	$e_1 \oplus e_2$		e_1	ρ	**binopLeftK**($\oplus, e_2$) :: $\vec{\kappa}$
7	v	$\vec{\kappa}$ = **binopLeftK**($\oplus, e$) :: $\vec{\kappa_2}$	e	ρ	**binopRightK**($v, \oplus$) :: $\vec{\kappa_2}$
8	v_2	$\vec{\kappa}$ = **binopRightK**($v_1, \oplus$) :: $\vec{\kappa_2}$, v_3 = $\texttt{evalOp}(v_1, \oplus, v_2)$	v_3	ρ	$\vec{\kappa_2}$
9	$x \Rightarrow e$		**closureV**(x, e, ρ)	ρ	$\vec{\kappa}$
10	$e_1(e_2)$		e_1	ρ	**anonFunLeftK**(e_2) :: $\vec{\kappa}$
11	**closureV**(x', e', ρ')	$\vec{\kappa}$ = **anonFunLeftK**(e) :: $\vec{\kappa_2}$	e	ρ	**anonFunRightK**(x', e', ρ') :: $\vec{\kappa_2}$
12	v	$\vec{\kappa}$ = **anonFunRightK**(x', e', ρ') :: $\vec{\kappa_2}$	e'	$\rho'[x' \mapsto v]$	**restoreK**(ρ) :: $\vec{\kappa_2}$
13	$fn(e)$	$fn \in \texttt{keys}(defs)$	e	ρ	**namedFunK**(fn) :: $\vec{\kappa}$
14	v	$\vec{\kappa}$ = **namedFunK**(fn) :: $\vec{\kappa_2}$, $(x \cdot e) = defs(fn)$	e	$\{\}[x \mapsto v]$	**restoreK**(ρ) :: $\vec{\kappa_2}$
15	v	$\vec{\kappa}$ = **restoreK**(ρ') :: $\vec{\kappa_2}$	v	ρ'	$\vec{\kappa_2}$
16	**if** (e_1) e_2 **else** e_3		e_1	ρ	**ifK**(e_2, e_3) :: $\vec{\kappa}$
17	{(**val** x = e_1) :: $\overrightarrow{val}$ e_2}		e_1	ρ	**blockK**($x, \overrightarrow{val}, e_2$) :: $\vec{\kappa}$
18	(e_1 :: $\vec{e_2}$)		e_1	ρ	**tupleK**($\vec{e_2}$, []) :: $\vec{\kappa}$
19	$e.\_n$		e	ρ	**accessK**(n) :: $\vec{\kappa}$
20	$cn(e)$		e	ρ	**constructorK**(cn) :: $\vec{\kappa}$
21	e **match** {$\overrightarrow{case}$}		e	ρ	**matchK**($\overrightarrow{case}$) :: $\vec{\kappa}$
22	**boolV**(**true**)	$\vec{\kappa}$ = **ifK**(e_1, e_2) :: $\vec{\kappa_2}$	e_1	ρ	$\vec{\kappa_2}$
23	**boolV**(**false**)	$\vec{\kappa}$ = **ifK**(e_1, e_2) :: $\vec{\kappa_2}$	e_2	ρ	$\vec{\kappa_2}$
24	v	$\vec{\kappa}$ = **blockK**(x_1, (**val** x_2 = e_1) :: $\overrightarrow{val}, e_2$) :: $\vec{\kappa_2}$	e_1	$\rho[x_1 \mapsto v]$	**blockK**($x_2, \overrightarrow{val}, e_2$) :: **restoreK**($\rho$) :: $\vec{\kappa_2}$
25	v	$\vec{\kappa}$ = **blockK**(x, [], e) :: $\vec{\kappa_2}$	e	$\rho[x \mapsto v]$	**restoreK**(ρ) :: $\vec{\kappa_2}$
26	v_1	$\vec{\kappa}$ = **tupleK**(e_1 :: $\vec{e_2}, \vec{v_2}$) :: $\vec{\kappa_2}$	e_1	ρ	**tupleK**($\vec{e_2}, v_1$ :: $\vec{v_2}$) :: $\vec{\kappa_2}$
27	v_1	$\vec{\kappa}$ = **tupleK**([], $\vec{v_2}$) :: $\vec{\kappa_2}$, $\vec{v_3}$ = $\texttt{reverse}(v_1 :: \vec{v_2})$	**tupleV**($\vec{v_3}$)	ρ	$\vec{\kappa_2}$
28	**tupleV**($\vec{v_1}$)	$\vec{\kappa}$ = **accessK**(n) :: $\vec{\kappa_2}$, v_2 = $\texttt{tupleAccess}(\vec{v_1}, n)$	v_2	ρ	$\vec{\kappa_2}$
29	v	$\vec{\kappa}$ = **constructorK**(cn) :: $\vec{\kappa_2}$	**constructorV**(cn, v)	ρ	$\vec{\kappa_2}$
30	**constructorV**(cn, v)	$\vec{\kappa}$ = **matchK**($\overrightarrow{case}$) :: $\vec{\kappa_2}$, $(x \cdot e)$ = $\texttt{constructorCaseLookup}(cn, \overrightarrow{case})$	e	$\rho[x \mapsto v]$	**restoreK**(ρ) :: $\vec{\kappa_2}$
31	**tupleV**($\vec{v_1}$)	$\vec{\kappa}$ = **matchK**($\overrightarrow{case}$) :: $\vec{\kappa_2}$, $(\rho' \cdot e)$ = $\texttt{tupleCaseLookup}(\vec{v_1}, \rho, \overrightarrow{case})$	e	ρ'	**restoreK**(ρ) :: $\vec{\kappa_2}$
32	v	$\vec{\kappa}$ = []	v	ρ	$\vec{\kappa}$