Recap: code generation notation

We use brackets **[s]** to denote “result of compiling s”

For compilation of expressions, we thus write:

$[\ e_1 + e_2\ ]$ =
  $[e_1]$
  $[e_2]$
  `i32.add`

$[\ e_1\ \&\ e_2\ ]$ =
  $[e_1]$
  $[e_2]$
  `i32.and`

Recap: simple way of translating if-then-else

Simple conditionals compiled with blocks and branches:

$[\![$ **if** e_{cond} **then** e_{then} **else** e_{else} $]\!]$ =

```
  block nAfter
    block nElse
      [! e_cond]
      br_if nElse
      [e_then]
      br nAfter
    end // nElse:
    [e_else]
  end // nAfter:
```

Question: how to translate this?

How to compile the *short-circuiting* **logical and** operator?

$$[\![\, e_1 \ \&\& \ e_2 \,]\!] =$$

Question: how to translate this?

How to compile the *short-circuiting* **logical and** operator?

$[\![\ e_1\ \&\&\ e_2\ ]\!]$ =
  $[\![e_1]\!]$
  $[\![e_2]\!]$ // problem: e_2 could be expensive and have side effects
  i32.and

Question: how to translate this?

How to compile the *short-circuiting* **logical and** operator?

```
[ e₁ && e₂ ] =
  [e₁]
  [e₂] // problem: e₂ could be expensive and have side effects
  i32.and
```

We need to use branches

Question: how to translate this?

How to compile the *short-circuiting* **logical and** operator?

$[\![e_1 \;\&\&\; e_2]\!]$ =
  $[\![e_1]\!]$
  $[\![e_2]\!]$ // problem: e_2 could be expensive and have side effects
  i32.and

We need to use branches

Idea: desugar to $[\![e_1 \;\&\&\; e_2]\!] = [\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else false}]\!]$

Problem: suboptimal code translation

Example:

`if` e_1 && e_2 **`then`** e_3 **`else`** e_4

Problem: suboptimal code translation

Example:

if e_1 && e_2 **then** e_3 **else** e_4

essentially compiles to the same thing as

if (**if** e_1 **then** e_2 **else** **false**) **then** e_3 **else** e_4

Problem: suboptimal code translation

Example:

$$\texttt{if } e_1 \ \&\& \ e_2 \ \texttt{then } e_3 \ \texttt{else } e_4$$

essentially compiles to the same thing as

$$\texttt{if } (\texttt{if } e_1 \ \texttt{then } e_2 \ \texttt{else false}) \ \texttt{then } e_3 \ \texttt{else } e_4$$

$\Rightarrow$ unnecessary branches

Could have compiled to (pseudo-code)

```
if e1 then
  if e2 then
    e3
  else "goto" L
else
L: e4
```

Translating control flow structures more efficiently

Introduce an imaginary large instruction **`branch`**`(c, nThen, nElse)`.

Here `c` is a potentially complex boolean expression

(the main reason why **`branch`** is not a built-in bytecode instruction),

whereas `nTrue` and `nFalse` are the labels we jump to,

depending on the boolean value of `c`.

Translating control flow structures more efficiently

Introduce an imaginary large instruction **`branch`**`(c, nThen, nElse)`.

Here `c` is a potentially complex boolean expression

(the main reason why **`branch`** is not a built-in bytecode instruction),

whereas `nTrue` and `nFalse` are the labels we jump to,

depending on the boolean value of `c`.

We will show how to

- use **`branch`** to compile **`if`** and short-circuiting operators,
- by expanding **`branch`** recursively into concrete bytecode instructions.

Translating control flow structures more efficiently

$$[\![\texttt{if}\ (e_{cond})\ e_{then}\ \texttt{else}\ e_{else}]\!] :=$$

```
block nAfter
  block nElse
    block nThen
      branch(e_cond, nThen, nElse)
    end //nThen:
    ⟦e_then⟧
    br nAfter
  end //nElse:
  ⟦e_else⟧
end //nAfter:
⟦e_rest⟧
```

Decomposing conditions in branch

```
branch(!e,nThen,nElse) :=
  branch(e,nElse,nThen)

branch(e1 && e2,nThen,nElse) :=
  block nLong
    branch(e1,nLong,nElse)
  end //nLong:
  branch(e2,nThen,nElse)

branch(e1 || e2,nThen,nElse) :=
  block nLong
    branch(e1,nThen,nLong)
  end //nLong:
  branch(e2,nThen,nElse)
```

Decomposing conditions in `branch`

```
branch(true,nThen,nElse) :=
  br nThen

branch(false,nThen,nElse) :=
  br nElse

branch(b,nThen,nElse) :=   (where b is a local var)
  get_local #b
  br_if nThen
  br nElse
```

Decomposing conditions in `branch`

branch($e_1 == e_2$,nThen,nElse) := (*where* e_1, e_2 *are of type* int)

$[\![e_1]\!]$
$[\![e_2]\!]$

```
i32.eq
br_if nThen
br nElse
```

... *analogously for other relations*

Returning the result from branch

Consider storing $x = c$
where x, c are boolean and c contains $\&\&$ or $\|$.

How do we put the result of c on the stack so it can be stored in x?

$[x = c]$:=

```
block nAfter
  block nElse
    block nThen
      branch(c,nThen,nElse)
    end //nThen:
    i32.const 1
    br nAfter
  end //nElse:
  i32.const 0
end //nAfter:
set_local #x
```

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)`

we had two arguments `nThen` and `nElse`,

which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the “rest” of the program.

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)`

we had two arguments `nThen` and `nElse`,

which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the “rest” of the program.

$\Rightarrow$ We can generalize our translation function $[\![\cdot]\!]$ to take a destination label designating the “rest” in the surrounding code.

Destination label parameters

Recall that in **branch**`(c,nThen,nElse)`

we had two arguments `nThen` and `nElse`,

which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

⇒ We can generalize our translation function $[\![\cdot]\!]$ to take a destination label designating the "rest" in the surrounding code.

$$[\![\cdot]\!] \Rightarrow [\![\cdot]\!]\,\texttt{nAfter}$$

⇒ The caller of the translation function determines where to continue!

Translations with an `nAfter` label parameter (1)

$[\![x = e]\!]$ `nAfter :=`

```
  block nSet
    [e] nSet
    // note that the rest of this block is never reached!
  end //nSet:
  set_local #x
  br nAfter
```

$[\![s_1; s_2]\!]$ `nAfter :=`

```
  block nSecond
    [s1] nSecond
  end //nSecond:
  [s2] nAfter
```

Translations with an `nAfter` label parameter (2)

$[$ *if* $(e_{cond})\ e_{then}$ *else* $e_{else}]$ nAfter $:=$
```
  block nElse
    block nThen
      branch(e_cond,nThen,nElse)
    end //nThen:
    [e_then] nAfter
  end //nElse:
  [e_else] nAfter
```

$[$ *return* $e]$ nAfter $:=$
```
  block nRet
    [e] nRet
  end //nRet:
  return
```

Switch statements

Let us assume our language had a switch statement (like C and Java do, for instance):

$$
\begin{array}{l}
\texttt{\textbf{switch}}\ (e_{scrutinee})\ \{ \\
\quad \texttt{\textbf{case}}\ c_1\texttt{\textbf{:}}\ e_1 \\
\quad \texttt{\textbf{...}} \\
\quad \texttt{\textbf{case}}\ c_n\texttt{\textbf{:}}\ e_n \\
\quad \texttt{\textbf{default:}}\ e_{default} \\
\}
\end{array}
$$

▷ How can we compile such switch statements?

Compiling switch statements

```
[s_switch] nAfter :=
  block nDefault
    block nCase_n
      ...
        block nCase_1
          block nTest
            [e_scrutinee] nTest
          end //nTest:
          tee_local #s   (where s is some fresh local of type i32)
          i32.const c_1; i32.eq; br_if nCase_1
          get_local #s
          i32.const c_2; i32.eq; br_if nCase_2
          ...
          br nDefault
        end //nCase_1:
        [e_1] nCase_2
      ...
    end //nCase_n:
    [e_n] nDefault
  end //nDefault:
  [e_default] nAfter
```

Compiling switch statements

```
[s_switch] nAfter :=
  block nDefault
    block nCase_n
      ...
        block nCase_1
          block nTest
            [e_scrutinee] nTest
          end //nTest:
          tee_local #s   (where s is some fresh local of type i32)
          i32.const c_1; i32.eq; br_if nCase_1
          get_local #s
          i32.const c_2; i32.eq; br_if nCase_2
          ...
          br nDefault
        end //nCase_1:
        [e_1] nCase_2
      ...
    end //nCase_n:
    [e_n] nDefault
  end //nDefault:
  [e_default] nAfter
```

- ▷ How do we translate `break`?

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

$\Rightarrow$ Let us extend the translation function by another label parameter.

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

$\Rightarrow$ Let us extend the translation function by another label parameter.

$$[\cdot]\ \texttt{nAfter} \ \Rightarrow\ [\cdot]\ \texttt{nAfter}\ \texttt{nBreak}$$

$\Rightarrow$ The caller of the translation function determines where to continue in the "normal" case, but also when `break` is called!

Compiling switch statements

Translating `break` then is straightforward: One simply ignores `nAfter` and follows `nBreak` instead.

$[\![$*break*$]\!]$ `nAfter nBreak` :=
  br `nBreak`

▷ What do we have change in our translation of `switch` statements?

Compiling switch statements with breaks

```
[s_switch] nAfter nBreak :=
  block nDefault
    block nCase_n
      ...
        block nCase_1
          block nTest
            [e_scrutinee] nTest nBreak
          end //nTest:
          tee_local #s   (where s is some fresh local of type i32)
          i32.const c_1; i32.eq; br_if nCase_1
          get_local #s
          i32.const c_2; i32.eq; br_if nCase_2
          ...
          br nDefault
        end //nCase_1:
        [e_1] nCase_2 nAfter
      ...
    end //nCase_n:
    [e_n] nDefault nAfter
  end //nDefault:
  [e_default] nAfter nAfter
```

Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

[**while** (cond) stmt] `nextLabel =`

Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

```
[ while (cond) stmt ] nextLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel
  end
```

break Statement

In many languages, a break statement can be used to exit from the loop. For example, it is possible to write code such as this:

```
while (cond1) {
  code1
  if (cond2) break;
  code2
}
```

Loop executes `code1` and checks the condition `cond2`. If condition holds, it exists. Otherwise, it continues and executes `code2` and then goes to the beginning of the loop, repeating the process.

Give translation scheme for this loop construct and explain how the translation of other constructs needs to change.

break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

[**if** (cond) thenC **else** elseC] `nextL loopExitL =`

break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

```
[ if (cond) thenC else elseC ] nextL loopExitL =
  block elseL
    block thenL
      branch(cond, thenL, elseL)
    end // thenL
    [thenC] nextL loopExitL
  end // elseL
  [elseC] nextL loopExitL
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
 br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
 loop startLabel
   block bodyLabel
     branch(cond, bodyLabel, nextLabel)
   end // bodyLabel
   [ stmt ]
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
 br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
 loop startLabel
   block bodyLabel
     branch(cond, bodyLabel, nextLabel)
   end // bodyLabel
   [ stmt ] startLabel
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

What if we want to have **continue** that goes to beginning of the loop?

Loops with break and continue

Translating **break**:

```
[ break ] nextL loopExitL loopStartL =
  br loopExitL
```

Translating **continue**:

```
[ continue ] nextL loopExitL loopStartL =
  br loopStartL
```

Translating while:

```
[ while (cond) stmt ] nextL loopExitL loopStartL =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextL)
    end // bodyLabel
    [ stmt ] startLabel nextL startLabel
  end
```

Explain difference between labels `loopStartL` and `startLabel`