L-values

• Simple Variables:
  - If the variable is living in a temporary t then return Tree.TEMP(t).
  - If the variable is living in the current stack frame then return Tree.MEM(FP + byte offset)
  - If the variable is in a previous stack frame, follow the static links to the right level, then add the byte offset and apply Tree.MEM

L-values

• Field selections
  - Given e.l, translate e to get a pointer to a record.
  - Look at the type of this record, and where the l field occurs in this type.
  - Add the appropriate byte offset to the beginning of the record. and apply Tree.MEM
    • Easy here because all fields are one word
  - Improvement: Check that the pointer to the record is non-null; make external call to exit with non-zero argument otherwise.

L-values

• Array subscripting
  - Given e1[e2], translate e1 to get a pointer to the beginning of the array, and e2 to get the index
  - Check that the index is in bounds?
    • Retrieve the array length from the array
    • Array subscript trick
  - Find the byte offset of the right element and apply Tree.MEM.
    • Easy here because arrays elements always 1 word
    • Otherwise, have to look at the type of the array to determine element sizes
Arithmetic

- Each arithmetic operator in the abstract syntax corresponds to a Tree.BINOP.
  - Recall: unary negation implemented as subtraction from zero.
  - Could be problematic in IEEE floating-point, where 0 - 0 is not the same as -0.

Record Creation

- *External* call to allocRecord with number of words needed (not bytes).
  - Don’t pass a static link
- Then initialize the record with
  \[\text{Temp.MOVE(Temp.MEM(...), value)}\]
  for each field.

Array Creation

- External call to initArray with the number of words needed and the initializing value.
  - Returns pointer to pre-initialized array
  - Array length?

While Loops

One possibility:

- Allocate fresh pair of labels for each loop.
- Any `break` inside the loop that’s not inside an inner loop can just branch to the `Done` label.
  - Hence need to keep track of break target during translation
For Loops

One possibility, (where limit is a fresh variable)

for i := lo to hi do body

let
  var i := lo
  var limit := hi
in while i <= limit
  do (body; i := i+1)
end

Problem: what happens when hi is maxint?

Function Call

- The function call $f(a_1, \ldots, a_n)$ becomes
  
  $$\text{CALL(NAME } l_f, [s_l, e_1, \ldots, e_n])$$

  where
  
  - $l_f$ is the code label for $f$
  - $s_l$ is the right static link for $f$
    - i.e., $fp$ for the activation record for the statically enclosing procedure
  - $e_i$ is the translation of the argument $a_i$.

Declarations

- Code for the declaration
  
  ```
  var x := e
  ```
  
  can be the same as for the assignment statement
  
  ```
  x := e
  ```

- Type declarations yield no code

- Function definitions are kept in a global list

  ```
  Translate.procEntryExit : {level:level, body: exp} -> unit
  ```

  (no actual code returned from translation)

String Literals

- Allocate string as data (via "string fragment"); mark with a fresh label $lab$
- Translation of the literal is then

  ```
  Tree.NAME(lab)
  ```
Conditionals

• Two ways to translate a boolean expression:
  – Numerical representation
    • Expression that evaluates to zero or one.
  – Control-flow representation
    • As code which will jump to one of two labels.

• Both are useful, depending on context

Numerical Representation

• Source code $x := (a > b)$

• Intermediate code

```
if (a>b) goto L1
x ← 0
goto L2
L1:
x ← 1
L2:
```

Question

• Translation of $\text{if } z \text{ then } e_1 \text{ else } e_2$ will probably start like

  ```
  CJUMP(EQ, z, 1, ..., ...)
  ```

• Should the abstract syntax

  ```
  if (a>b) then e1 else e2
  ```

  start with

  ```
  CJUMP(EQ, "a>b", 1, ..., ...)
  ```

Real target of translation

```python
 datatype exp =
  Ex of Tree.exp
 | Nx of Tree.stm
 | Cx of (Temp.label * Temp.label
        -> Tree.stm)
```

• Boolean expressions default to $Cx$ representation: given a true-destination and a false-destination, return statement that jumps to the right label.
Examples

Then the code \( a > b \) can be translated to
\[
Cx(fn \ (t,f) \Rightarrow CJUMP(GT, a, b, t, f))
\]

Similarly, \((a > b) \mid (c < d)\) becomes
\[
Cx(fn \ (t,f) \Rightarrow SEQ(CJUMP(GT, a, b, t, z),
SEQ(Label\ z,\ CJUMP(LT, c, d, t, f)))
\]
for some new label \( z \).

Conversions

- Appel suggests writing the conversion functions

\[
\begin{align*}
unEx &: \ exp \rightarrow Tree.exp \\
unNx &: \ exp \rightarrow Tree.stm \\
unCx &: \ exp \rightarrow (Temp.label \times Temp.label \rightarrow Tree.stm)
\end{align*}
\]

Using conversions

- When compiling assignments

\[
\begin{align*}
x &= 3 \\
x &= (a > b) \mid (c < d)
\end{align*}
\]

translate the right-hand-side expression, apply \texttt{unEx}, and create a \texttt{MOVE}.
Compiling Conditionals

• Simplest: Apply \(\text{unCx}\) to the tested expression and \(\text{unEx}\) to the arms.

• Better: Special case for when
  - at least one arm is a \(N_x\) (return an \(N_x\))
  - at least one arm is a \(C_x\) (return a \(C_x\))

Digression: Compiling \textit{switch}

• Several possibilities for compiling a switch statement.
  - Sequence of tests
  - Array of jump targets (labels)
    • Best when possible values of switch expression are dense in some range
  - Hash table of jump targets
    • Better when there are many possible values, sparsely distributed
  - Combination of these.
    • E.g., use tests to narrow down to a specific range, then use the appropriate array.