Compilers Course
Lecture 13: AST to RTL Translation: Basics

Problem: where are local variables located?
1. Assume all local variables are on the stack, access variables via SP (or FP) plus offset.
   - works but makes the RTL very machine specific, and prevents some optimizations in the back-end .
2. Treat local variables as temps, delay storage assignment (stack slots or registers) to the back-end.
   + better for portability and code optimization
   - needs virtual stack storage for arrays and any variable accessed via a pointer

We will follow approach 2 here. For now we only consider scalar variables, arrays and other data structures will be handled later.

Scalar Variable Declarations

1. Global variable
   * map name to label
   * output RTL global variable declaration

```
GLOBAL(x, int) // x needs space for an 'int'
```

2. Local variable or function parameter
   * create a new temp for that variable

Use a symbol table to keep track of the location (LABEL L or temp t) of each scalar program variable.

Constants "c"

```
result = c
```

Expressions

Expressions are translated to sequences computing their values into a result temporary.

Reading the value of a variable "x"

if x is a local var in temp_x :
```
result = temp_x
```

if x is in memory at LABEL_x :
```
address = LABEL_x
result = load(address)
```

Assigning x = E

First translate E, assume its value is in temp_e:

```
temp_e = E
```

if x is a local var in temp_x

```
temp_x = temp_e
```

if x is in memory at LABEL_x

```
address = LABEL_x
store(address, temp_e)
```

Unary expressions: op(E)

```
temp = E            // recursively translate E
result = op temp    // translate op
```

Binary expressions: E1 op E2

```
temp1 = E1                // recursively translate E1
temp2 = E2                // recursively translate E2
result = temp1 op temp2  // translate op
```

Function calls: f(E1, ..., En)

```
temp_1 = E1
...
temp_n = En
result = f(temp_1, ..., temp_n)
```

Summary

Translation of expressions is done by a recursive procedure:

- Parameter: expression (AST)
- Parameter: symbol table mapping variables to "locations" (temporaries or global labels)
- Result is a list of instructions and a temp containing the final value, alternatively pass in the desired result temp as a parameter
- Inspects the shape of the expression
- Recursively translates subexpressions
- Combines the results to finish translation of the expression

Statements

Statements are translated to sequences performing their effects in the correct order.

Sequences: S1 ; S2

```
S1
S2
```

IF statements

```
if E1 then
   S1
else if E2 then
   S2
else
   S3
```

Generally we emit instructions in the same order as they occur in the program, with jumps to control execution order:

```
     temp1 = E1
     if not temp1 goto L2
     S1
     goto Lnext
L2:  temp2 = E2
     if not temp2 goto L3
     S2
     goto Lnext
L3:  S3
Lnext:
```

Some expressions also have control flow:
`e1 && e2, e1 || e2, e1 ? e2 : e3`

WHILE loops

```
while E do S
```

Naive translation:

```
Ltest:    temp = E
```

```
     if not temp goto Lnext
     S
     goto Ltest
Lnext:
```

The number of jumps executed (whether taken or not) is 2N+1 for a loop with N iterations.

Improved translation:

```
     goto Ltest
Lbody:    S
Ltest:    temp = E
     if temp goto Lbody
Lnext:
```

Now the number of jumps is N+2.

BREAK/CONTINUE

A break is a goto to the statement following the current loop.
A continue is a goto to the iteration test of the current loop.
So to translate them we must place labels at these points and pass those labels as parameters to the translation procedure.

```
while E1 do begin
  if E2 break       // goto Lnext
  if E3 continue    // goto Ltest
end
```

DO loops

```
do S while E
```

Exactly like the improved version of WHILE loops, except we start in the loop body not the test:

```
Lbody:    S
Ltest:    temp = E
     if temp goto Lbody
Lnext:
```

"break" becomes "goto Lnext"
"continue" becomes "goto Ltest"

FOR loops

```
for (init; condition; step) S

     init
     goto Lcond
Lbody:    S
Lstep:    step
Lcond:    temp = cond
     if temp goto Lbody
Lnext:
```

"break" becomes "goto Lnext"
"continue" becomes "goto "Lstep"

SWITCH

```
switch (E) {
  case C1: S1; break;
  case C2: S2; break;
  ...
  default: Sn
}
```

The semantics in C is that S1; S2; ...; Sn are output in that order, with a label Li at each case statement Si. This sequence is preceded by code that compares E with C1, then C2, and so on until a match is found. If a match is found, a jump is made to the corresponding label:

```
     temp = E
     if temp == C1 goto L1
     if temp == C2 goto L2
     ...
     goto Ln
L1:  S1
L2:  S2
...
Ln:  Sn
Lnext:
```

A "break" in any Si becomes "goto Lnext".
The initial tests can also be implemented using binary search, a jump table: an array where element Ci contains Li, or by a loop over an array of <Ci, Li> elements.

RETURNS

```
return E

     temp = E
     tempRV = temp
     goto Lreturn
```

where tempRV is the temp used for the function's return value, and Lreturn is the label of the epilogue code (deallocate frame and return).

Example:

```
int max(int a, int b)
{
    if (a > b)
     return a;
    else
     return b;
}

sum(ta, tb):
     tcond = ta > tb
     if not tcond goto Lelse
     tempRV = ta
     goto Lreturn
Lelse:
     tempRV = tb;
Lreturn:
     <backend will add return code here>
```

Summary

Translation of statements is done by a recursive procedure:

- Parameter: statement (AST)
- Parameter: symbol table (passed on to expressions)
- Parameters: labels for break/continue/return
- Result is a list of instructions
- Inspects the shape of the statement
- Emits code for jumps etc mixed with recursive calls to translate sub-statements