Last Time ...

- Code generation designed in three steps:
  code templates, operand conditions, decision tree

- Implementation strategy trees (ISTs) encode the code selection process in this way

- The most important code templates are those for variable access:
  subscripting, field selection and pointer dereferencing

This time:
Generating code for expressions - using S/SL
Generating Code For Expressions

• Generating working code for expressions is relatively easy - generating good code is more difficult

• The basic rule of thumb for generating better code is the delay principle:

  wait as long as possible before generating any code!

• The idea is that the longer we wait the more information we have, and therefore the more optimization we can do
Generating Code For Expressions

• The Delay Principle

Example:  
\[
\begin{align*}
\text{push} & \quad x \\
\text{push} & \quad 1 \\
\text{add} & \\
\end{align*}
\]

If we generate code for these intermediate code instructions one by one, immediately as we go (e.g, x86):

\[
\begin{align*}
\text{push} & \quad x \quad \text{movl} \quad x, \%eax \\
\text{push} & \quad 1 \quad \text{movl} \quad \$1, \%ebx \\
\text{add} & \quad \text{addl} \quad \%ebx, \%eax \\
\end{align*}
\]

But if instead we wait until all three intermediate code instructions are known, we can instead generate:

\[
\begin{align*}
\text{push} & \quad x \\
\text{push} & \quad 1 \\
\text{add} & \quad \text{movl} \quad x, \%eax \\
\text{add} & \quad \text{incl} \quad \%eax \\
\end{align*}
\]
The Operand Stack

• The *operand stack* is a structure used to remember operands as they are given to us by the semantic analyzer in order to delay generation of code.

• We stack the operands as we receive them in T-code, and when we have sufficient to emit the appropriate code, we pop them from the stack - once again, this is a simulation of the ES.

• Each element of the operand stack is a *data descriptor* that corresponds to the remembered operand.
The Operand Stack

• A semantic mechanism called the *Operand* mechanism is used to manage the operand stack.

• The top three elements of the stack are directly manipulatable using semantic operations of the mechanism - they are referred to as the *right, left* and *target* operands.

• Example:

  \[
a := b + c
  \]

  
  \[
  \begin{array}{c}
  \text{push address a} \\
  \text{push b} \\
  \text{push c} \\
  \text{add}
  \end{array}
  \]

  
  \[
  \begin{array}{ccc}
  & \text{c} & \text{right} \\
  & \text{b} & \text{left} \\
  & \text{addr(a)} & \text{target}
  \end{array}
  \]

* Note: the target operand is called the *dest* ("destination") operand in the PT code generator.
Addition Example

OperandAddAssignPopPopPopPop:
[ oOperandCompareLeftAndTarget
  | same:
    oOperandSwap
    @OperandFreePop
    @OperandAddRightToLeftPop
    @OperandFreePop
  | *:
    oOperandSwap
[ oOperandCompareLeftAndTarget
  | same:
    oOperandSwap
    @OperandFreePop
    @OperandAddRightToLeftPop
    @OperandFreePop
  | *:
    oOperandSwap
    @OperandAddPop
    @OperandAssignPopPopPop

];
Addition Example (cont'd)

OperandAddRightToLeftPop:
[ oOperandChooseManifestValue
 | one:
   oOperandPop
   @OperandForceAddressable
   oEmit(iInc)
   oEmitOperand
 | minusOne:
   oOperandPop
   @OperandForceAddressable
   oEmit(iDec)
   oEmitOperand
 | zero:
   oOperandPop
 |*:
 % ... a b
 % a += 1
 % ... a
 % ... a
 % inc a
 % ... a
 % ... a
 % dec a
 % ... a
 % not manifest

... continued on next slide
Addition Example (cont'd)

```plaintext
[ oOperandCompareLeftAndRight
  | same:
    @OperandFreePop
    @OperandForceAddressable
    oEmit(iAsl)
    oEmitOperand
  | *:
    @OperandForceLeftAndRightAddressable
    oEmit(iAdd)
    @EmitOperandFreePop
    oEmitOperand

  ]
];
```
Addition Example (cont'd)

OperandAddPop:
  [oOperandLeftAndRightManifest
   | yes:
     oOperandFoldaddManifestValues
     oOperandPop
   | *:
     [oOperandIsManifest
       | yes:
         oOperandSwap
       | *:
     ]
   ]
@OperandForceIntoTemporary % ... c b
  @OperandAddRightToLeftPop % ... c temp
  oOperandSwap % ... temp c
  @OperandAddRightToLeftPop % ... temp
];
Temporaries

• Temporary storage locations must be allocated and freed as necessary by the code generator

• Most often, temporaries are registers, but on machines with very few registers (PDP-11, x86) the memory stack may be used

• A semantic mechanism called the *Temporary* mechanism is used to manage the allocation of temporaries

• Code generator rules use the temporary mechanism to *allocate* a free temporary location when needed in a template, and to *free* them when they are no longer in use
Temporaries

OperandForceIntoTemporary :
oTempAllocate oOperandPushNewTemp oOperandSwap @OperandForceAddressable oEmit(iMov) @EmitOperandFreePop oEmitOperand;

% allocate a free temporary
% push it on the operand stack
% put original operand on top
% make sure it is addressable
% mov X,temp
% leave temp on operand stack

OperandFreePop :
oTempFreeOperand oOperandPop;

% free temporary if any
% pop operand

EmitOperandFreePop :
oEmitOperand @OperandFreePop;

Summary

• *Delay principle*: To get better code, delay code choice as long as possible, in order to have enough information to make optimal decisions.

• In S/SL code generators, the *Operand Stack* mechanism implements this idea by holding operands until all are known.

• **Next**: Generating code for Boolean expressions.