Intermediate Representations

Intermediate Representation (IR):
- An abstract machine language
- Expresses operations of target machine
- Not specific to any particular machine
- Independent of source language

IR code generation not necessary:
- Semantic analysis phase can generate real assembly code directly.
- Hinders portability and modularity.

Suppose we wish to build compilers for \( n \) source languages and \( m \) target machines.

Case 1: no IR
- Need separate compiler for each source language/target machine combination.
- A total of \( n \times m \) compilers necessary.
- Front-end becomes cluttered with machine specific details, back-end becomes cluttered with source language specific details.

Case 2: IR present
- Need just \( n \) front-ends, \( m \) back ends.
Intermediate Representations

FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.
From Modern Compiler Implementation in ML,
Cambridge University Press, ©1998 Andrew W. Appel

Properties of a Good IR

- Must be convenient for semantic analysis phase to produce.
- Must be convenient to translate into real assembly code for all desired target machines.
  - RISC processors execute operations that are rather simple.
    - Examples: load, store, add, shift, branch
    - IR should represent abstract load, abstract store, abstract add, etc.
  - CISC processors execute more complex operations.
    - Examples: multiply-add, add to/from memory
    - Simple operations in IR may be “clumped” together during instruction selection to form complex operations.

IR Representations

The IR may be represented in many forms:

Expression trees:
- `exp`: constructs that compute some value, possibly with side effects.
- `stm`: constructs that perform side effects and control flow.

```
signature TREE = sig
datatype exp = CONST of int
               | NAME of Temp.label
               | TEMP of Temp.temp
               | BINOP of binop * exp * exp
               | MEM of exp
               | CALL of exp * exp list
               | BSEQ of stm * exp
```
IR Expression Trees

TREE continued:

and stm = MOVE of exp * exp
| EXP of exp
| JUMP of exp * Temp.label list
| CJUMP of relop * exp * exp *
| Temp.label * Temp.label
| SEQ of stmt * stmt
| LABEL of Temp.label
and binop = PLUS|MINUS|MUL|DIV|AND|OR|
| LSHIFT|RSHIFT|ARSHIFT|XOR
and relop = EQ|NE|LT|GT|LE|GE|ULT|ULE|UGT|UGE

Expressions

Expressions compute some value, possibly with side effects.

`CONST(t)` integer constant `t`

`NAME(n)` symbolic constant `n` corresponding to assembly language label (abstract name for memory address)

`TEMP(t)` temporary `t`, or abstract/virtual register `t`

`BINOP(op, e1, e2)` `e1` `op` `e2` evaluated before `e2`

- integer arithmetic operators: PLUS, MINUS, MUL, DIV
- integer bit-wise operators: AND, OR, XOR
- integer logical shift operators: LSHIFT, RSHIFT
- integer arithmetic shift operator: ARSHIFT

Expressions

`MEM(e)` contents of wordSize bytes of memory starting at address `e`

- wordSize is defined in Frame module.
- if MEM is used as left operand of MOVE statement ⇒ store
- if MEM is used as right operand of MOVE statement ⇒ load

`CALL(f, l)` application of function `f` to argument list `l`

- subexpression `f` is evaluated first
- arguments in list `l` are evaluated left to right

`BSEQ(s, e)` the statement `s` evaluated for side-effects, `e` evaluated next for result
Statements

Statements have side effects and perform control flow.

MOVE (TEMP (t), c) evaluate c and move result into temporary t.

MOVE (MEM (c1), c2) evaluate c1, yielding address a; evaluate c2, store result in wordsize bytes of memory starting at address a

EXP (c) evaluate expression c, discard result.

JUMP (c, labs) jump to address c
   • c may be literal label (NAME (l)), or address calculated by expression
   • labs specifies all locations that c can evaluate to (used for dataflow analysis)
   • jump to literal label l: JUMP (NAME (l), [l])

CJUMP (op, c1, c2, t, f) evaluate c1, then c2; compare results using op; if true, jump to t, else jump to f
   • EQ, NE: signed/unsigned integer equality and non-equality
   • LT, GT, LE, GE: signed integer inequality
   • ULT, UGT, ULE, UGE: unsigned integer inequality

Statements

SEQ (s1, s2) statement s1 followed by s2

LABEL (l) label definition - constant value of l defined to be current machine code address
   • similar to label definition in assembly language
   • use NAME (l) to specify jump target, calls, etc.
   • The statements and expressions in TREE can specify function bodies.
   • Function entry and exit sequences are machine specific and will be added later.

Translation of Abstract Syntax

• if Absyn.exp computes value ⇒ Tree.exp
• if Absyn.exp does not compute value ⇒ Tree.stm
• if Absyn.exp has boolean value ⇒ Tree.stm and Temp.labels

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

• Ex “expression” represented as a Tree.exp
• Nx “no result” represented as a Tree.stm
• Cx “conditional” represented as a function. Given a false-destination label and a true-destination label, it will produce a Tree.stm which evaluates some conditionals and jumps to one of the destinations.
Conditional:

\[ x > y: \]
\[ \text{Cx}(\text{fn } (t, f) \rightarrow \text{CJUMP}(\text{GT}, x, y, t, f)) \]

\[ a > b \mid c < d: \]
\[ \text{Cx}(\text{fn } (t, f) \Rightarrow \text{SEQ}(	ext{CJUMP}(\text{GT}, a, b, t, z), \text{SEQ}(	ext{LABEL } z, \text{CJUMP}(\text{LT}, c, d, t, f)))) \]

**May need to convert conditional to value:**

\[ a := x > y: \]

\( \text{Cx} \) corresponding to “\( x > y \)” must be converted into \( \text{Tree.exp } e \).

\[ \text{MOVE}(\text{TEMP}(a), e) \]

Need three conversion functions:

\[
\begin{align*}
\text{val unEx: exp } & \rightarrow \text{Tree.exp} \\
\text{val unNn: exp } & \rightarrow \text{Tree.stm} \\
\text{val unCx: exp } & \rightarrow (\text{Temp.label } \ast \text{Temp.label } \rightarrow \text{Tree.stm})
\end{align*}
\]

**Translation of Abstract Syntax (Conditionals)**

The three conversion functions:

\[
\begin{align*}
\text{val unEx: exp } & \rightarrow \text{Tree.exp} \\
\text{val unNn: exp } & \rightarrow \text{Tree.stm} \\
\text{val unCx: exp } & \rightarrow (\text{Temp.label } \ast \text{Temp.label } \rightarrow \text{Tree.stm})
\end{align*}
\]

\[ a := x > y: \]

\[ \text{MOVE}(\text{TEMP}(a), \text{unEx}(\text{Cx}(t,f)) \Rightarrow ...) \]

\( \text{unEx} \) makes a \( \text{Tree.exp} \) even though \( e \) was \( \text{Cx} \).

**Translation of Abstract Syntax**

**Implementation of function** \( \text{UnEx} \):

structure \( T = \text{Tree} \)

fun unEx(Ex(e)) = e

| unEx(Nx(s)) = T.ESEQ(s, T.CONST(0))
| unEx(Cx(genstm)) =
  | let val r = Temp.newtemp()
  | val t = Temp.newlabel()
  | val f = Temp.newlabel()
  | in T.ESEQ(seq{T.MOVE(T.TEMP(r), T.CONST(1))},
    | genstm(t, f),
    | T.LABEL(f),
    | T.MOVE(T.TEMP(r), T.CONST(0)),
    | T.LABEL(t)),
    | T.TEMP(r))
  | end
Translation of Abstract Syntax

- Recall type and value environments \texttt{tenv, venv}.
- The function \texttt{transVar} return a record \{\texttt{exp, ty}\} of \texttt{Translate.exp and Types.ty}.
- \texttt{exp} is no longer a place-holder

Simple Variables

- **Case 1:** variable \(v\) declared in current procedure’s frame

\[
\text{InFrame}(k): \\
\quad \text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP}(\text{FP}), \text{CONST}(k)))
\]

\(k:\) offset in own frame

\texttt{FP} is declared in \texttt{FRAME} module.

- **Case 2:** variable \(v\) declared in temporary register

\[
\text{InReg}(_{t.103}): \\
\quad \text{TEMP}(_{t.103})
\]

Simple Variables

- **Case 3:** variable \(v\) not declared in current procedure’s frame, need to generate IR code to follow static links

\[
\text{InFrame}(k_n): \\
\quad \text{MEM}(\text{BINOP}(\text{PLUS}, \text{CONST}(_{k_n}), \\
\quad \text{MEM}(\text{BINOP}(\text{PLUS}, \text{CONST}(_{k_{n-1}}), \\
\quad \quad \text{MEM}(\text{BINOP}(\text{PLUS}, \text{CONST}(_{k_{n-2}}), \\
\quad \quad \quad \text{MEM}(\text{BINOP}(\text{PLUS}, \text{CONST}(_{k_1}), \text{TEMP}(\text{FP})))))))))))
\]

\(k_1, k_2, ..., k_{n-1}:\) static link offsets

\(k_n:\) offset of \(v\) in own frame
Simple Variables

To construct simple variable IR tree, need:

- \( l_f \): level of function \( f \) in which \( v \) used
- \( l_g \): level of function \( g \) in which \( v \) declared
- MEM nodes added to tree with static link offsets \((k_1, \ldots, k_{n-1})\)
- When \( l_g \) reached, offset \( k_n \) used.

Array Access

Given array variable \( a \),

\[
\begin{align*}
&a[a[0]] = a \\
&a[a[1]] = a + w, \text{ where } w \text{ is the word-size of machine} \\
&a[a[2]] = a + (2 * w) \\
&\ldots
\end{align*}
\]

Let \( e \) be the IR tree for \( a \):

\[
\begin{aligned}
&a[i]: \\
&\text{MEM}(\text{BINOP}(\text{PLUS}, e, \text{BINOP}(\text{MUL}, i, \text{CONST}(w))))
\end{aligned}
\]

Compiler must emit code to check whether \( i \) is out of bounds.

Record Access

```plaintext
type rectype = {f1:int, f2:int, f3:int}

<table>
<thead>
<tr>
<th>offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

var a:rectype := rectype{f1=4, f2=5, f3=6}
```

Let \( e \) be IR tree for \( a \):

\[
\begin{aligned}
&a.f3: \\
&\text{MEM}(\text{BINOP}(\text{PLUS}, e, \text{BINOP}(\text{MUL}, \text{CONST}(3), \text{CONST}(w))))
\end{aligned}
\]

Compiler must emit code to check whether \( a \) is nil.
Conditional Statements

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

- Treat \( e_1 \) as Cx expression ⇒ apply unCx.
- Treat \( e_2, e_3 \) as Ex expressions ⇒ apply unEx.

\[
\text{Ex}(\text{BSEQ}(\text{SEQ}(\text{unCx}(e_1))(t, f),
                            \text{SEQ}(\text{LABEL}(t),
                            \text{SEQ}(\text{MOVE}(\text{TEMP}(r), \text{unEx}(e_2)),
                            \text{SEQ}(\text{JUMP}(\text{NAME}(\text{join})),
                            \text{SEQ}(\text{LABEL}(f),
                            \text{SEQ}(\text{MOVE}(\text{TEMP}(r), \text{unEx}(e_3)),
                            \text{LABEL}(\text{join}))))))))
\]
\[
\text{TEMP}(r))
\]

Strings

- All string operations performed by run-time system functions.
- In Tiger, C, string literal is constant address of memory segment initialized to characters in string.
  - In assembly, label used to refer to this constant address.
  - Label definition includes directives that reserve and initialize memory.

```
"foo"
```
1. Translate module creates new label \( l \).
2. Tree . NAME \( l \) returned: used to refer to string.
3. String fragment “foo” created with label \( l \). Fragment is handed to code emitter, which emits directives to initialize memory with the characters of “foo” at address \( l \).

Strings

**String Representation:**

- **Pascal** fixed-length character arrays, padded with blanks.
- **C** variable-length character sequences, terminated by ‘\000’
- **Tiger** any 8-bit code allowed, including ‘\000’

<table>
<thead>
<tr>
<th>label</th>
<th>&quot;foo&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>0</td>
<td>o</td>
</tr>
</tbody>
</table>
Strings

- Need to invoke run-time system functions
  - string operations
  - string memory allocation
- `Frame.externalCall: string * Tree.exp -> Tree.exp`
  - `Frame.externalCall("stringEqual", [s1, s2])`
- Implementation takes into account calling conventions of external functions.
- Easiest implementation:
  ```
  fun externalCall(s, args) =
    T.CALL(T.NAME(Temp.namedlabel(s)), args)
  ```

Array Creation

type intarray = array of int
var a:intarray := intarray[10] of 7

Call run-time system function `initArray` to `malloc` and initialize array.
  ```
  Frame.externalCall("initArray", [CONST(10), CONST(7)])
  ```

Record Creation

type rectype = { f1:int, f2:int, f3:int }
var a:rectype := rectype{f1 = 4, f2 = 5, f3 = 6}

SEQ{SEQ( MOVE(TEMP(result),
    Frame.externalCall("allocRecord",
    [CONST(12)])),
    SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(0*w)),
        CONST(4)),
    SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(1*w)),
        CONST(5)),
    SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(2*w)),
        CONST(6))))),
    TEMP(result))}

- `allocRecord` is an external function which allocates space and returns address.
- `result` is address returned by `allocRecord`. 
While Loops

One layout of a **while loop**:

```
while CONDITION do BODY
```

test:
  if not(CONDITION) goto done
  BODY
  goto test
done:

A **break** statement within body is a JUMP to label done.

**transExp** and **transDec** need formal parameter “break”:
- passed done label of nearest enclosing loop
- needed to translate breaks into appropriate jumps
- when translating while loop, **transExp** recursively called with loop done label in order to correctly translate body.

For Loops

Basic idea: Rewrite AST into let/while AST; call **transExp** on result.

```
for i := lo to hi do
  body
```

Becomes:

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

Complication:
If **limit == maxint**, then increment will overflow in translated version.

Function Calls

```
f(a1, a2, ..., an) =>
  CALL (NAME(1_f), s1::[e1, e2, ..., en])
```

- **s1** static link of f (computable at compile-time)
- To compute static link, need:
  - 1_f : level of f
  - 1_g : level of g, the calling function
- Computation similar to simple variable access.
Consider type checking of “let” expression:

```haskell
def transExp (venv, tenv) =
    ... |
    \exp (A.LetExp {decs, body, pos}) =
        let
            \{venv = venv’, tenv = tenv’\} =
                transDecs (venv, tenv, decs)
        in
            transExp (venv’, tenv’) body
        end
```

- Need level, break.
- What about variable initializations?

**Function Declarations**

- Cannot specify function headers with IR tree, only function bodies.
- Special “glue” code used to complete the function.
- Function is translated into assembly language segment with three components:
  - prologue
  - body
  - epilogue
Function Prolog

Prologue preceeds body in assembly version of function:
1. Assembly directives that announce beginning of function.
2. Label definition for function name.
3. Instruction to adjust stack pointer (SP) - allocate new frame.
4. Instructions to save escaping arguments into stack frame, instructions to move non-escaping arguments into fresh temporary registers.
5. Instructions to store into stack frame any callee-save registers used within function.

Function Epilog

Epilogue follows body in assembly version of function:
6. Instruction to move function result (return value) into return value register.
7. Instructions to restore any callee-save registers used within function.
8. Instruction to adjust stack pointer (SP) - deallocate frame.
9. Return instructions (jump to return address).
10. Assembly directives that announce end of function.

- Steps 1, 3, 8, 10 depend on exact size of stack frame.
- These are generated late (after register allocation).
- Step 6:

```
MOVE (TEMP(RV), unEx(body))
```

Fragments

signature FRAME = sig
... 
datatype frag = STRING of Temp.label * string
| PROC of {body:Tree.stm, frame:frame}
end

- Each function declaration translated into fragment.
- Fragment translated into assembly.
- body field is instruction sequence: 4, 5, 6, 7
- frame contains machine specific information about local variables and parameters.
Problem with IR trees generated by the Translate module:

- Certain constructs don’t correspond exactly with real machine instructions.
- Certain constructs interfere with optimization analysis.
- CJUMP jumps to either of two labels, but conditional branch instructions in real machine only jump to one label. On false condition, fall-through to next instruction.
- ESEQ, CALL nodes within expressions force compiler to evaluate subexpression in a particular order. Optimization can be done most efficiently if subexpressions can proceed in any order.
- CALL nodes within argument list of CALL nodes cause problems if arguments passed in specialized registers.

Solution: Canonicalizer

Canonicalizer

Canonicalizer takes Tree.stm for each function body, applies following transforms:

1. Tree.stm becomes Tree.stm list, list of canonical trees. For each tree:
   - No SEQ, ESEQ nodes.
   - Parent of each CALL node is EXP(...) or MOVE(TMP(t), ...)

2. Tree.stm list becomes Tree.stm list list, statements grouped into basic blocks
   - A basic block is a sequence of assembly instructions that has one entry and one exit point.
   - First statement of basic block is LABEL.
   - Last statement of basic block is JUMP, CJUMP.
   - No LABEL, JUMP, CJUMP statements in between.

3. Tree.stm list list becomes Tree.stm list
   - Basic blocks reordered so every CJUMP immediately followed by false label.
   - Basic blocks flattened into individual statements.

Instruction Selection

Instruction Selection

- Process of finding set of machine instructions that implement operations specified in IR tree.
- Each machine instruction can be specified as an IR tree fragment → tree pattern
- Goal of instruction selection is to cover IR tree with non-overlapping tree patterns.
Our Architecture

- Load/Store architecture
- Relatively large, general purpose register file
  - Data or addresses can reside in registers (unlike Motorola 68000)
  - Each instruction can access any register (unlike x86)
- \( r_0 \) always contains zero.
- Each instruction has latency of one cycle.
- Execution of only one instruction per cycle.

### Arithmetic:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( r_d = r_{x_1} + r_{x_2} )</td>
</tr>
<tr>
<td>ADDI</td>
<td>( r_d = r_{x_1} + c )</td>
</tr>
<tr>
<td>SUB</td>
<td>( r_d = r_{x_1} - r_{x_2} )</td>
</tr>
<tr>
<td>SUBI</td>
<td>( r_d = r_{x_1} - c )</td>
</tr>
<tr>
<td>MUL</td>
<td>( r_d = r_{x_1} \times r_{x_2} )</td>
</tr>
<tr>
<td>DIV</td>
<td>( r_d = r_{x_1} / r_{x_2} )</td>
</tr>
</tbody>
</table>

### Memory:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>( r_d = M[r_{x_1} + c] )</td>
</tr>
<tr>
<td>STORE</td>
<td>( M[r_{x_1} + c] = r_{x_2} )</td>
</tr>
<tr>
<td>MOVEM</td>
<td>( M[r_{x_1}] = M[r_{x_2}] )</td>
</tr>
</tbody>
</table>

---

**Pseudo-ops**

*Pseudo-op* - An assembly operation which does not have a corresponding machine code operation. Pseudo-ops are resolved during assembly.

<table>
<thead>
<tr>
<th>Pseudo-op</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>( r_d = r_j )</td>
</tr>
<tr>
<td>ADDI</td>
<td>( r_d = r_x + 0 )</td>
</tr>
<tr>
<td>MOV</td>
<td>( r_d = r_j )</td>
</tr>
<tr>
<td>ADD</td>
<td>( r_d = r_{x_1} + r_0 )</td>
</tr>
<tr>
<td>MOVI</td>
<td>( r_d = c )</td>
</tr>
<tr>
<td>ADDI</td>
<td>( r_d = r_0 + c )</td>
</tr>
</tbody>
</table>

(Pseudo-op can also mean assembly directive, such as `.align`.)
### Instruction Tree Patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$r_i$</td>
<td>TEMP $0$</td>
</tr>
<tr>
<td>ADD</td>
<td>$r_j + r_2$</td>
<td>1</td>
</tr>
<tr>
<td>MUL</td>
<td>$r_j \times r_2$</td>
<td>2</td>
</tr>
<tr>
<td>SUB</td>
<td>$r_j - r_k$</td>
<td>3</td>
</tr>
<tr>
<td>DIV</td>
<td>$r_j / r_k$</td>
<td>4</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_j + c$</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>SUBI</td>
<td>$r_j - c$</td>
<td>8</td>
</tr>
<tr>
<td>LOAD</td>
<td>$M[r_j + c]$</td>
<td>MEM $0$, MEM $1$, MEM $2$, MEM $3$</td>
</tr>
</tbody>
</table>

---

### Instruction Tree Patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE</td>
<td>$M[r_j + c]$</td>
<td>MOVE MEM $1$, MEM $2$, MEM $3$</td>
</tr>
<tr>
<td>MOVEM</td>
<td>$M[r_j]$</td>
<td>MOVE MEM $1$, MEM $2$</td>
</tr>
</tbody>
</table>

---

### Example

$a[i] := x$ assuming $i$ in register, $a$ and $x$ in stack frame.

![Example Diagram](attachment:image.png)
Individual Node Selection

```
MOVE
  MEM
  MEM
    PLUS
    PLUS
      MEM
      MULT
    TEMP
    CONST
      PLUS
      TEMP
      CONST
      FP
      offset-x

TEMP
CONST
temp-i
4

FP
offset-a
```

Individual Node Selection

```
ADDI r1 = r0 + offset_a
ADD r2 = r1 + FP
LOAD r3 = M[r2 + 0]

ADDI r4 = r0 + 4
MUL r5 = r4 * r_i
ADD r6 = r3 + r5

ADDI r7 = r0 + offset_x
ADD r8 = r7 + FP
LOAD r9 = M[r8 + 0]
STORE M[r6 + 0] = r9

9 registers, 10 instructions
```

Random Tiling

```
MOVE
  MEM
  MEM
    PLUS
    PLUS
      MEM
      MULT
    TEMP
    CONST
      PLUS
      TEMP
      CONST
      FP
      offset-x

TEMP
CONST
temp-i
4

FP
offset-a
```
Random Tiling

ADDI r1 = r0 + offset_a
ADD r2 = r1 + FP
LOAD r3 = M[r2 + 0]
ADDI r4 = r0 + 4
MUL r5 = r4 * r_i
ADD r6 = r3 + r5
ADDI r7 = r0 + offset_x
ADD r8 = r7 + FP
MOVEM M[r6] = M[r8]

Saves a register (9 → 8) and an instruction (10 → 9).

Node Selection

- There exist many possible tilings - want tiling/covering that results in instruction sequence of least cost
  - Sequence of instructions that takes least amount of time to execute.
  - For single issue fixed-latency machine: fewest number of instructions.
- Suppose each instruction has fixed cost:
  - Optimum Tiling: tiles sum to lowest possible value - globally “the best”
  - Optimal Tiling: no two adjacent tiles can be combined into a single tile of lower cost - locally “the best”
- Optimal instruction selection easier to implement than Optimum instruction selection.
- Optimal is roughly equivalent to Optimum for RISC machines.
- Optimal and Optimum are noticeably different for CISC machines.
- Instructions are not self-contained with individual costs.

Optimal Instruction Selection:
Maximal Munch

- Cover root node of IR tree with largest tile t that fits (most nodes)
  - Tiles of equivalent size ⇒ arbitrarily choose one.
- Repeat for each subtree at leaves of t.
- Generate assembly instructions in reverse order - instruction for tile at root emitted last.
Maximal Munch

LOAD  r3 = M[FP + offset_a]
ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i
ADD   r6 = r3 + r5
ADD   r8 = FP + offset_x
MOVEM M[r6] = M[r8]

5 registers, 6 instructions

Assembly Representation

structure Assem = struct
  type reg = string
  type temp = Temp.temp
  type label = Temp.label

  datatype instr = OPER of
    {assem: string,
     dst: temp list,
     src: temp list,
     jump: label list option}
    | ...
    ...
end
```ocaml
fun codegen(frame) { stm: Tree.stm; Assem.instr list =
  let
    val ilist = ref(nil: Assem.instr list)
    fun emit(x) = ilist := x::ilist
    fun munchStm: Tree.stm -> unit
    fun munchExp: Tree.exp -> Temp.temp
    in
      munchStm(stm);
      rev(!ilist)
  end

---

Statement Munch

```ocaml
fun munchStm(
  T.MOVE(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c))), e2)
) =
  emit(Assem.OPER{assem="STORE M[\'s0 + "$ ^
                              int(c) ""] = \'s1\n",
                    src=[munchExp(e1), munchExp(e2)],
                    dst=[],
                    jump=NONE})
| munchStm(T.MOVE(T.MEM(e1), T.MEM(e2))) =
  emit(Assem.OPER{assem="MOVEM M[\'s0] = M[\'s1]" ^
                    src=[munchExp(e1), munchExp(e2)],
                    dst=[],
                    jump=NONE})
| munchStm(T.MOVE(T.MEM(e1), e2)) =
  emit(Assem.OPER{assem="STORE M[\'s0] = \'s1\n",
                    src=[munchExp(e1), munchExp(e2)],
                    dst=[t],
                    jump=NONE})

...

---

Expression Munch

```ocaml
and munchExp(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c)))) =
  let
    val t = Temp.newtemp()
  in
    emit(Assem.OPER{assem="LOAD 'd0 = M[\'s0 +" ^
                     int(c) ""]\n",
                    src=[munchExp(e1)],
                    dst=[t],
                    jump=NONE})

    t
  end
```
Expression Munch

```haskell
| munchExp(T.BINOP(T.PLUS, el, T.CONST(c))) =
  let
    val t = Temp.newTemp()
  in
    emit(Assem.OPER{assem="ADDI \text{ \texttt{d}0 = \texttt{s0} +} \text{ int(c)} ^\text{ \texttt{n}}",
                   src=[munchExp(el)],
                   dst=[t],
                   jump=NONE});
    t
  end

... |
| munchExp(T.TEMP(t)) = t
```

Optimum Instruction Selection

- Find optimum solution for problem (tiling of IR tree) based on optimum solutions for each subproblem (tiling of subtrees)
- Use Dynamic Programming to avoid unnecessary recomputation of subtree costs.
- *cost* assigned to *every* node in IR tree
  - Cost of best instruction sequence that can tile subtree rooted at node.
- Algorithm works bottom-up (Maximum Munch is top-down) - Cost of each subtree $s_j(c_j)$ has already been computed.
- For each tile $t$ of cost $c$ that matches at node $n$, cost of matching $t$ is:
  \[ c_t + \sum_{\text{all \texttt{bin} from \texttt{t}}} c_i \]
- Tile is chosen which has minimum cost.

Optimum Instruction Selection – Example

```plaintext
MEM(BINOP(PLUS, CONST(1), CONST(2))))
MEM(PLUS(CONST(1), CONST(2)))
```

```
        MEM
       /    |
      PLUS  
     /     |
    CONST  CONST
        1    2
```
Step 1: Find cost of root node
(a,b): a is minimum cost, b is corresponding pattern number

Consider PLUS node:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
<th>Leaves Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) PLUS(e1, e2)</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(6) PLUS(CONST(c), e1)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(7) PLUS(e1, CONST(c))</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 2: Emit instructions
ADDI r1 = r0 + 1
LOAD r2 = M[r1 + 2]
Optimum Instruction Selection – Big Example

```
MOVE
  MEM
  PLUS
  MEM
  PLUS
  MEM
  MULT
  TEMP
  CONST
  FP
  offset-x
  TEMP
  CONST
  temp-i
  4
  FP
  offset-a

LOAD  r3 = M[FP + offset_a]
ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i
ADD   r6 = r3 + r5
LOAD  r9 = M[FP + offset_x]
STORE M[r6] = r9

5 registers, 6 instructions
Optimal tree generated by Maximum Munch is also optimum...
```