Back-End Code Generation

- Given a list of itree fragments, how to generate the corresponding assembly code?

```
datatype frag = PROC of {
  name : Tree.label,  -- function name
  body : Tree.stm,    -- itree body
  frame : Frame.frame} -- static frame layout
| DATA of string
```

- Main challenges: certain aspects of itree statements and expressions do not correspond exactly with machine languages:
  - # of temp. registers on real machines are limited
  - real machine’s conditional-JUMP statement takes only one label
  - high-level constructs ESEQ and CALL --- side-effects

Itree Stmts and Exprs

- itree statements and itree expressions

```
datatype stm = SEQ of stm * stm
| LABEL of label
| JUMP of exp
| CJUMP of test * label * label
| MOVE of exp * exp
| EXP of exp

and exp = BINOP of binop * exp * exp
| CVTOP of cvtop * exp * size * size
| MEM of exp * size
| TEMP of temp
| NAME of label
| CONST of int
| CONSTF of real
| ESEQ of stm * exp
| CALL of exp * exp list
```

Side-Effects

- Side-effects means updating the contents of a memory cell or a temporary register. What are itree expressions that might cause side effects? ESEQ and CALL nodes

- ESEQ(s,e) where s is a list of statements that may contain MOVE statement

  The natural way to generate assembly code for
  ```
  BINOP(op, t1, t2)
  instructions to compute t1 into r1;
  instructions to compute t2 into r2;
  r1 <- r1 op r2
  ```

  But it won’t work for this:
  ```
  BINOP(PLUS, TEMP a, ESEQ(MOVE(TEMP a, u), v))
  ```

- CALL(e,el) by default puts the result in the return-result register.

  ```
  BINOP(PLUS, CALL(...) CALL(...))
  ```

Summary: IR -> Machine Code

- Step #1: Transform the itree code into a list of canonical trees
  a. eliminate SEQ and ESEQ nodes
  b. the arguments of a CALL node should never be other CALL nodes
     ---- the parent of each CALL node should either be EXP(...) or MOVE(TEMP t, ...)

- Step #2: Perform various code optimizations on canonical trees

- Step #3: Rearrange the canonical trees (into traces) so that every
  CJUMP (cond, l1, l2) is immediately followed by LABEL(l2).

- Step #4: Instruction Selection ---- generate the pseudo-assembly code from the canonical trees in the step #3.

- Step #5: Perform register allocations on pseudo-assembly code
Canonical Trees

- A canonical tree is a simple itree statement in the following form (it is really a restricted-kind of itree statement):

  ```
  datatype stm = LABEL of label |
  | JUMP of exp |
  | CJUMP of test * label * label |
  | MOVE of exp * exp |
  | EXP of exp |
  |
  and exp = BINOP of binop * exp * exp |
  | CVTOP of cvtop * exp * size * size |
  | MEM of exp * size |
  | TEMP of temp |
  | NAME of label |
  | CONST of int |
  | CONSTF of real |
  | CALL of exp * exp list 
  ```

Restrictions:
- no SEQ statements, no ESEQ expressions.
- each CALL node doesn't contain other CALL nodes as subtrees

Canonicalizer

- The body of each PROC fragment is translated into an ordered list of canonical trees

  ```
  stms: Tree.stm list 
  ```

  - Step 1: transformation on CALL nodes.

    ```
    CALL(...) =======>
    ESEQ(MOVE(TEMP t, CALL(...)), TEMP t) 
    ```

  - Step 2: elimination of ESEQ nodes. (see Appel pp 174-179)

    ```
    lift them higher and higher until they become SEQ nodes ... 
    ```

Rearranging itree statements

- Goal: rearrange the list of canonical trees so that every CJUMP (cond, l_1, l_2) is immediately followed by its false branch LABEL(l_2).
- Step #1: take a list of canonical trees and form them into basic blocks

  A basic block is a sequence of statements that is always entered at the beginning and exited at the end:

  1. the first statement is a LABEL
  2. the last statement is a JUMP or CJUMP
  3. there are no other LABELs, JUMPs, or CJUMPs in between

  basic blocks are often used to analyze a program's control flow
- Step #2: re-order the list of basic blocks into traces

Canonical Trees => Basic Blocks

- Input: a sequence of statements (i.e., canonical trees --- the body of a function);
- Output: a set of basic blocks
- Algorithm:
  - if a new LABEL is found, end the current block and start a new block;
  - if a JUMP or CJUMP is found, end the current block;
  - if it results a block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block;
  - if it results a block without a LABEL at the beginning, invent a new LABEL and stuck it there;
  - invent a new label done for the beginning of the epilogue;
  - put JUMP(NAME done) at the end of the last basic block.
Basic Blocks => Traces

- **Control Flow Graph (CFG):** basic blocks as the nodes, pairs (a, b) as the edges if block a ends with a CJUMP or JUMP statement to block b.
- **Basic blocks can be arranged in any order, but we want:**
  - each CJUMP is followed by its false label
  - each JUMP should be followed by its target label whenever possible
- A trace is a path in the CFG --- it characterizes some fragment of a real program execution.
- **Algorithm for gathering traces:** just do the depth-first traversal of the CFG ------ (can also take advantage of branch prediction information)

Traces => List of Statements

- Flatten the traces back to an ordered list of statements (canonical trees):
  a) any CJUMP followed by its false label: do nothing;
  b) any CJUMP followed by its true label: switch its true and false label, and negate the condition;
  c) remove JUMP (l) if it is followed by its target l;
  d) any CJUMP (cond, lt, lf) followed by neither label: invent a new false label ln, rewrite it into:

\[
\text{CJUMP} \left( \text{cond}, \text{lt}, \text{ln} \right) \\
\text{LABEL} \left( \text{ln} \right) \\
\text{JUMP} \left( \text{NAME, lf} \right)
\]

- We are now ready to do instruction selection: generate assembly code for your favourite target machine.

Instruction Selection

- **Input:** an ordered list of canonical trees;
- **Output:** the pseudo-assembly code (without register assignments)
- **Algorithm:** translating each canonical tree into an assembly code sequence, and then concatenate all sequences together.
- **Main Problem:** how to map the canonical tree to the assembly code?
- Each machine instruction can be expressed as a tree pattern --- a fragment of the canonical tree:
  - Load the value at addr e + c in the memory!
  - Each machine instruction may correspond to several layer of itree expressions

Instruction Selection via Tiling

- Express each machine instruction as a tree pattern.
- Given a canonical tree, the instruction selection is just to tile the tree using various tree patterns (for all possible machine instructions) ------ cover the canonical tree using nonoverlapping tiles.
- **Optimum Tiling:** ---- one whose tiles sum to the lowest possible value (suppose we give each machine instruction a cost)
- **Optimal Tiling:** ---- one where no two adjacent tiles can be combined into a single tile of lower cost
- Even optimum tiling is also optimal, but no vice versa.
- **Algorithm:** maximum munch finds the optimal tiling; dynamic programming finds the optimum tiling.
Maximal Munch

- **Algorithm** --- *Maximal Munch*:
  
  Start at the root of a canonical tree, find the *largest tile* that fits; the *largest tile* is the one with the most nodes (if tie, break arbitrarily).

  Cover the root node and perhaps several other nodes near the root with this tile, leaving several subtrees; the instruction corresponding to the *tile* is generated.

  Repeat the same algorithm for each subtrees.

- *Maximal Munch* generates the instructions in *reverse* order.

- **Implementation**: see Appel pp 190-191, 204-205.

---

Dynamic Programming

- the *dynamic programming* algorithm is used to find the *optimum tiling*
  
  **Main Idea**: assign a cost to every node in the tree (via bottom-up).
  
  the algorithm works bottom-up: at each node, we calculate the cost of the best instruction sequence that can tile the subtree rooted at that node.

  - after the cost of the root node (thus the entire tree) is found, we do the *instruction emission*:
    
    Emission(node $n$): for each leaves $l_i$ of the tile selected at node $n$, perform Emission($l_i$). Then emit the instruction matched at node $n$.

---

Code-Generator Generator

- Same as *Lex* and *Yacc*, the *instruction selection* phase can also be automatically built, using a *code-generator generator*.

- The *input specification* is a set of grammar rules used to specify the *tree pattern* for each machine instruction:

  each grammar rule is associated with a *cost* and an *action*; *cost* is for finding optimum tiling; *action* is for instruction emission.

  **Example**:

  \[
  \begin{align*}
  d &\rightarrow \text{MEM}(\ast, \text{CONST}) \\
  d &\rightarrow \text{MEM}(\ast, \text{a}) \\
  d &\rightarrow \text{MEM}(\text{CONST}) \\
  d &\rightarrow \text{MEM}(\text{a}) \\
  \end{align*}
  \]

  $\ast$: expressions for “addressing”  
  $\text{a}$: expressions for “data”

- The code-generator generator computes the minimum-cost match at each node for each nonterminal of the grammar using *dynamic programming* (Appel pp 191-193)

---

Instruction Selection for Tiger

- we will implement the *maximal munch* for instruction selection in the Tiger compiler (using ML pattern matching)

- **main problem**: how to deal with registers?

- **solution**: the register allocation will occur after instruction selection, the instruction selection phase will generate instructions with simple register templates.

  first, generate the *assembly tree* --- the assembly language without register assignments;

  second, do the *register allocation*

  third, emit the procedure entry exit sequence
Assembly Trees

- the assembly language without register assignments in ML datatype:

```ml
structure Assem : sig
  type reg = string
  type temp = Temp.temptype
  type label = Temp.label
  datatype instr =
    OPER of {assem: string, dst: temp list, src: temp list, jump: label list option}
    | LABEL of {assem: string, lab: label}
    | MOVE of {assem: string, dst: temp, src: temp}

  val format: (temp -> string) -> instr -> string
end
```

The `format` function will fill in the register information in the future.

Tiger Assembly Trees (cont’d)

- A `OPER` node `OPER(assem, dst, src, jump)` holds an assembly-language instruction `assem`. The source registers are `src`, the target registers are `dst`; `jump` would be `NONE` if it is not a branch instruction.

- A `MOVE` node `MOVE(assem, dst, srcs)` holds an assembly-language `move` instruction `assem` that moves from `src` to `dst`. If later in register allocation, `src` and `dst` are assigned the same register, then this instruction will be deleted.