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.

```
```

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’

```
label: 3
       f
       o
       o
```

"foo"
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}

ESEQ(SEQ( MOVE(TMP(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**: 

```plaintext
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.

\[
\text{for } i := lo \text{ to } hi \text{ do }
\]
\[
\text{ body }
\]

Becomes:

\[
\text{let }
\]
\[
\text{ var } i := lo \\
\text{ var limit := hi }
\]
\[
\text{ in }
\]
\[
\text{ while } (i \leq \text{ limit) do }
\]
\[
(\text{ body; }
\]
\[
\text{ i := i + 1) }
\]
\[
\text{ end }
\]

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

\[ f(a_1, a_2, \ldots, a_n) \Rightarrow \]
\[ \text{CALL}(\text{NAME}(l_f), \text{sl}::[e_1, e_2, \ldots, e_n]) \]

- \( \text{sl} \) static link of \( f \) (computable at compile-time)
- To compute static link, need:
  - \( l_f \) : level of \( f \)
  - \( l_g \) : level of \( g \), the calling function
- Computation similar to simple variable access.
Declarations

Consider type checking of “let” expression:

```ml
fun transExp(venv, tenv) =
    ...
    | trexp(A.LetExp{decs, body, pos}) =
        let
            val {venv = venv', tenv = tenv'} =
                transDecs(venv, tenv, decs)
            in
                transExp(venv', tenv') body
            end
```

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

Need to modify code to handle IR translation:

1. `transExp, transDec` require `level` to handle variable references.

2. `transExp, transDec` require `break` to handle breaks in loops.

3. `transDec` must return `Translate.exp` list of assignment statements corresponding to variable initializations.
   - Will be prepended to body.
   - `Translate.exp` will be empty for function and type declarations.
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 Prologue

Prologue precedes 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 Epilogue

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:

  \[ \text{MOVE (TEMP (RV), unEx (body))} \]
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

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 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 $MOVE(\text{TEMP}(t), \ldots)$

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.
Canonicalizer

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.