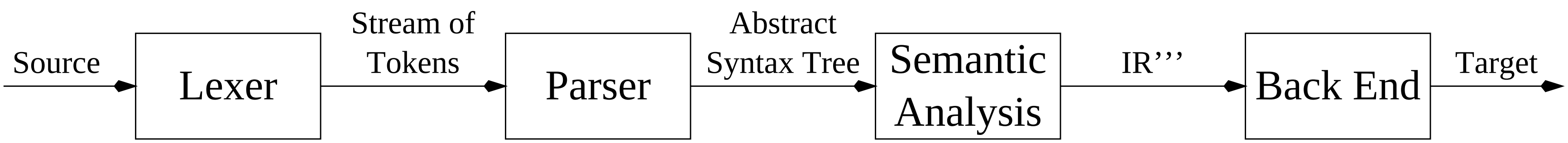
Intermediate Representation



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.

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.

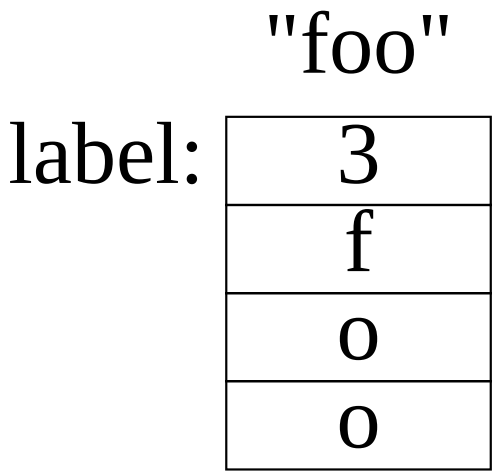
Strings

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’



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)])
```

Array Accesses

Given array variable `a`,

```
&(a[0]) = a
&(a[1]) = a + w, where w is the word-size of machine
&(a[2]) = a + (2 * w)
...
```

Let `e` be the IR tree for `a`:

```
a[i]:
  MEM(BINOP(PLUS, e, BINOP(MUL, i, CONST(w))))
```

Compiler must emit code to check whether `i` is out of bounds.

Record Creation

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

ESEQ(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`.

Record Accesses

```
type rectype = {f1:int, f2:int, f3:int}
                  |       |       |
        offset:   0       1       2
```

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

Let `e` be IR tree for `a`:

```
a.f3:
  MEM(BINOP(PLUS, e, BINOP(MUL, CONST(3), CONST(w))))
```

Compiler must emit code to check whether `a` is `nil`.

Conditional Statements

`if` e_1 `then` e_2 `else` e_3

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

```
Ex(ESEQ(SEQ(unCx(e1)(t, f),
         SEQ(LABEL(t),
          SEQ(MOVE(TEMP(r), unEx(e2)),
           SEQ(JUMP(NAME(join)),
            SEQ(LABEL(f),
             SEQ(MOVE(TEMP(r), unEx(e3)),
              LABEL(join)))))))
        TEMP(r)))
```

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(l_f), sl::[e1, e2, ..., en])
```

- `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:

```
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:

```
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.

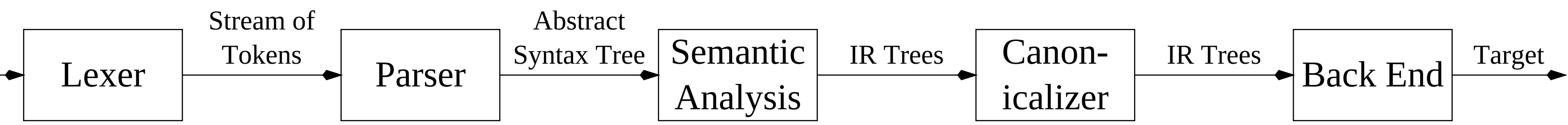
Problem with IR Trees

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: Rewrite the IR Trees produced by Translate so they are *semantically equivalent* but do not satisfy the conditions above

Canonicalizer



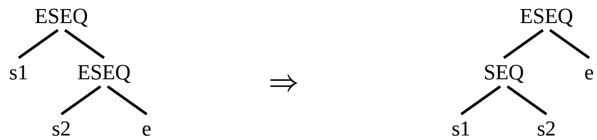
Canonicalizer takes `Tree.stm` for each function body, applies following transforms:

1 `Tree.stm` becomes `Tree.stm list`, list of canonical trees. For each tree:

- Rotate `SEQ, ESEQ` nodes from deep within the tree, higher and higher.
- Finally, there are no `SEQ, ESEQ` nodes anywhere inside the statement; they are all at the top with one `SEQ` following another. Eliminate the `SEQ` statements in favor of a list.
- Simultaneously, rotate each `CALL` node up the tree until `CALL` is surrounded by `EXP(...)` or `MOVE(TEMP(t), ...)`

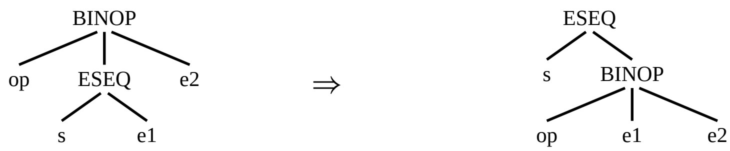
At all times, we must convince ourselves that rotations are *semantics preserving*.

(1)



$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) = \text{ESEQ}(\text{SEQ}(s_1, s_2), e)$

(2)



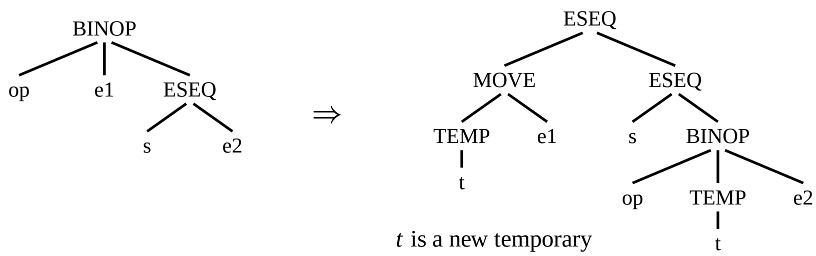
$\text{BINOP}(op, \text{ESEQ}(s, e_1), e_2) = \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

$\text{MEM}(\text{ESEQ}(s, e_1)) = \text{ESEQ}(s, \text{MEM}(e_1))$

$\text{JUMP}(\text{ESEQ}(s, e_1)) = \text{SEQ}(s, \text{JUMP}(e_1))$

$\text{CJUMP}(op, \text{ESEQ}(s, e_1), e_2, l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$

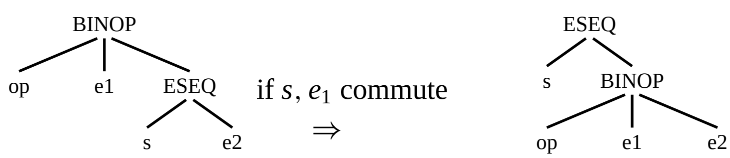
(3)



$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) = \text{ESEQ}(\text{MOVE}(\text{TEMP}\ t, e_1), \text{ESEQ}(s, \text{BINOP}(op, \text{TEMP}\ t, e_2)))$

$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) = \text{SEQ}(\text{MOVE}(\text{TEMP}\ t, e_1), \text{SEQ}(s, \text{CJUMP}(op, \text{TEMP}\ t, e_2, l_1, l_2)))$

(4)



if s, e_1 commute

$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) = \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$

FIGURE 8.1. Identities on trees (see also Exercise 8.1).

Canonicalizer

When do statements and expressions commute?

- We can never tell exactly, so we must make a *conservative* approximation.
  - `CONST C` commutes with any other statement or expression
  - `NAME L` commutes with any other statement or expression
  - Does `MOVE(MEM(x),y)` commute with `BINOP(MEM(x),y)`?
  - Does `MOVE(MEM(x),y)` commute with `BINOP(MEM(z),y)`?
  - Does `CALL(f,args)` commute with `BINOP(MEM(z),y)`?

Canonicalizer

- Implement `ESEQ` eliminator using the equivalences we just looked at.
- Must also rewrite calls:
  - Eg: `BINOP(PLUS,CALL(...),CALL(...)` = ...?
  - `CALL(f,args)` = `ESEQ(MOVE(TEMP t,CALL(f,args)), TEMP t)`
  - Now `ESEQ` eliminator will lift the `CALL` out of the `BINOP` expression

Canonicalizer

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(cond,a,b,t,f)` immediately followed by false label. Three cases:
  * We move basic block with false label to the point after the `CJUMP`.
  * We move basic block with true label to the point after the `CJUMP`, switch true and false labels and negate the condition.
  * We create a new false label `L’` and rewrite: `CJUMP(cond,a,b,t,L’); LABEL L’; JUMP f`
- Basic blocks flattened
- Further Optimization: whenever possible have the block for `L` follow `JUMP L` and delete the `JUMP L` instruction
- Further Optimization: give priority to `JUMP` and `CJUMP` inside loops. But how do we detect loops now that we just have jump statements everywhere??