Topic 7: Intermediate Representations

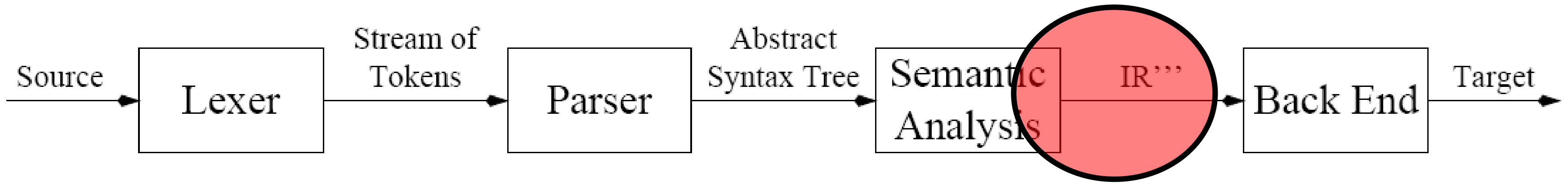
COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

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.

Intermediate Representations

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 $n * m$ 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

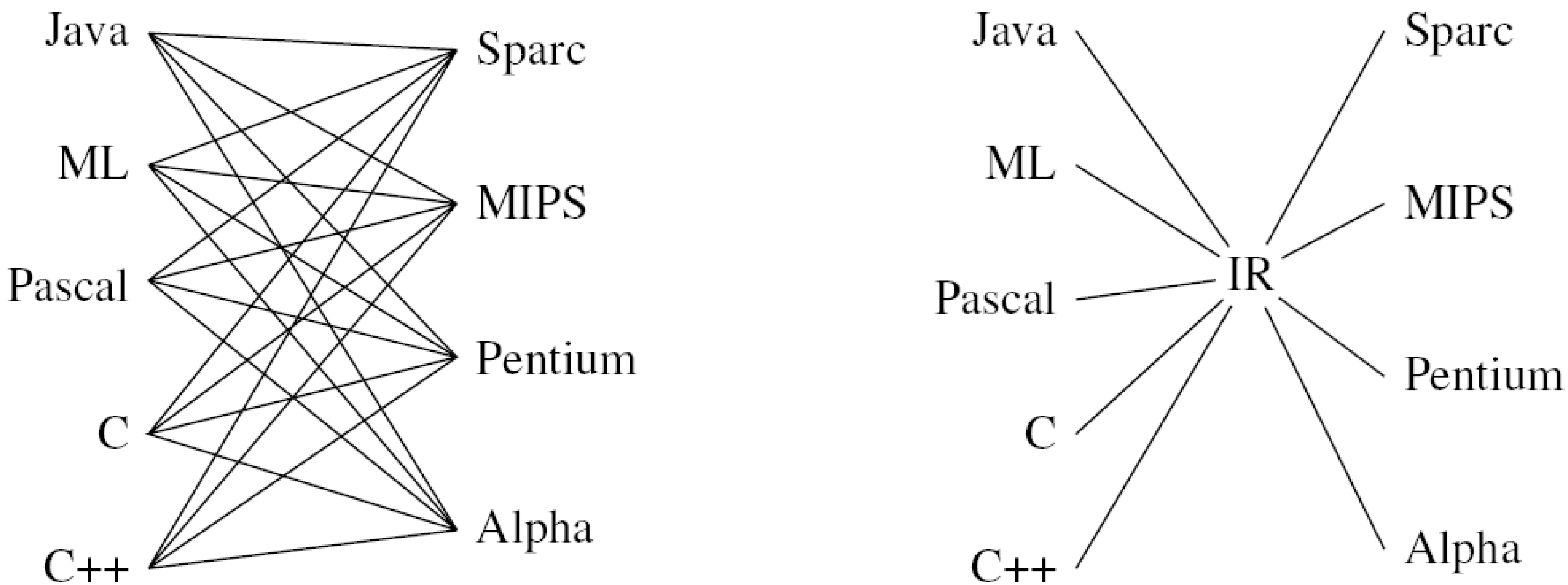


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.
From *Modern Compiler Implementation in ML,* Cambridge University Press, ©1998 Andrew W. Appel

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:

- Liberty, IMPACT, and Elcor compilers use *pseudo-assembly*.
- gcc and Tiger use *expression trees*.
- Intel's Electron, and HP's production compiler use both.

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
               | ESEQ of stm * exp
```

(Explanations on next slides)

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 stm * stm
               | 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
end
```

(Explanations on next slides)

Expressions

Expressions compute some value, possibly with side effects.

CONST(i) integer constant i

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, e_1, e_2) e_1 op e_2, e_1 evaluated before e_2

- 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 $\Rightarrow$ store
- if `MEM` is used as right operand of `MOVE` statement $\Rightarrow$ 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

`ESEQ(`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`), `e`)` evaluate e and move result into temporary t.

`MOVE(MEM(`e_1`), `e_2`)` evaluate e_1, yielding address a; evaluate e_2, store result in `wordSize` bytes of memory stating at address a

`EXP(`e`)` evaluate expression e, discard result.

`JUMP(`e`, `$labs$`)` jump to address e

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

`CJUMP(`op`, `e_1`, `e_2`, `t`, `f`)` evaluate e_1, then e_2; 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(`s_1`, `s_2`)` statement s_1 followed by s_2

`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.

Next:

- generation of IR code from Absyn
- heavily interdependent with design of FRAME module in MCIL (abstract interface of activation records, architecture-independent)

But first …

Midterm exam info

When? Thursday, March 10th, 3pm – 4:20pm

Where? CS 104 (HERE)

Closed book / notes, no laptop/smartphone….
Honor code applies

Material in scope:

- up to HW 3 (parser), and
- anything covered in class until this Friday.

Preparation:

- exercises at end of book chapters in MCIL
- old exams: follow link on course home page

Midterm exam prep QUIZ (also because it's Tuesday)

Problem 3: (20%) (Spring 2011)
Consider the expression language from the typing lectures, without functions, products, or subtypes, as summarized below. Define the typing context $\Gamma = [y : \textbf{ref int}, b : \textbf{bool}]$ and the expression e by

$$\textbf{let } x = 3 \textbf{ in if } (x < !y) \vee b \textbf{ then alloc } (x+1) \textbf{ else let } z = y{:=}8 \textbf{ in } 4 \textbf{ end end}.$$

Is there some type τ such that $\Gamma \vdash e : \tau$ is derivable using the rules? If no, say why not, i.e. show where an attempt to construct a typing derivation fails. If yes, give a suitable typing derivation.

$$\begin{array}{rcl} e & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \textbf{tt} \mid \textbf{ff} \mid e \oplus e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid x \\ & & \mid \textbf{let } x = e \textbf{ in } e \textbf{ end} \mid \textbf{alloc } e \mid !e \mid e{:=}e \\ \oplus & ::= & + \mid - \mid \times \mid \wedge \mid \vee \mid < \mid = \\ \tau & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{ref } \tau \mid \textbf{unit} \end{array}$$

$$\textsc{Bool}\ \frac{e \in \{\textbf{tt}, \textbf{ff}\}}{\Gamma \vdash e : \textbf{bool}} \qquad \text{NUM}\ \frac{n \in \{\ldots, -1, 0, 1, \ldots\}}{\Gamma \vdash n : \textbf{int}} \qquad \textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\text{IOp}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \textbf{int} \\ \Gamma \vdash e_2 : \textbf{int}\end{array}}{\Gamma \vdash e_1 \oplus e_2 : \textbf{int}} \oplus \in \{+, -, \times\} \qquad \text{BOp}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \textbf{bool} \\ \Gamma \vdash e_2 : \textbf{bool}\end{array}}{\Gamma \vdash e_1 \oplus e_2 : \textbf{bool}} \oplus \in \{\wedge, \vee\}$$

$$\text{COp}\ \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \oplus e_2 : \textbf{bool}} \oplus \in \{<, =\} \qquad \text{ITE}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \textbf{bool} \\ \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau\end{array}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma[x : \sigma] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \qquad \textsc{Alloc}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{alloc } e : \textbf{ref } \tau}$$

$$\textsc{Read}\ \frac{\Gamma \vdash e : \textbf{ref } \tau}{\Gamma \vdash !e : \tau} \qquad \textsc{Write}\ \frac{\Gamma \vdash e_1 : \textbf{ref } \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1{:=}e_2 : \textbf{unit}}$$

Translation of Abstract Syntax

Goal: function Translate: Absyn.exp => "IR"

Observation: different expression forms in Absyn.exp suggest use of different parts of IR

- if `Absyn.exp` computes value $\Rightarrow$ `Tree.exp`
- if `Absyn.exp` does not compute value $\Rightarrow$ `Tree.stm`
- if `Absyn.exp` has boolean value $\Rightarrow$ `Tree.stm` and `Temp.labels`

Solution 1: given e:Absyn.exp, always generate a Tree.exp term:

- case A: immediate
- case B: instead of a Tree.stm s, generate Tree.ESEQ(s, Tree.CONST 0)
- case C: "Tree.ESEQ (s, Tree.TEMP r)": cf expression on slide 16

Resulting code less clean than solution 2

Translation of Abstract Syntax

Solution 2:
define a wrapper datatype with "injections" (ie constructors) for the 3 cases

```
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.

Translation of Abstract Syntax (Conditionals)

Translation of a simple conditional into a Tree.stm:

```
x > y:
  Cx(fn (t, f) => CJUMP(GT, x, y, t, f))
```

Translation of a more complex conditional into a Tree.stm:

```
a > b | c < d:
  Cx(fn (t, f) => SEQ(CJUMP(GT, a, b, t, z),
                      SEQ(LABEL z, CJUMP(LT, c, d, t, f))))
```

Translation of a more conditional into a Tree.exp, to be assigned to a variable:

```
a := x > y:
```

Cx corresponding to "x > y" must be converted into `Tree.exp` e. Then, can use

```
  MOVE(TEMP(a), e)
```

Need conversion function unEx: exp => Tree.exp. Convenient to have unNx and unCx, too:

```
val unEx: exp -> Tree.exp
val unNx: exp -> Tree.stm
val unCx: exp -> (Temp.label * Temp.label -> Tree.stm)
```

Translation of Abstract Syntax (Conditionals)

The three conversion functions:

```
val unEx: exp -> Tree.exp
val unNx: exp -> Tree.stm
val unCx: exp -> (Temp.label * Temp.label -> Tree.stm)
```

```
a := x > y:
  MOVE(TEMP(a), unEx(Cx(t,f) => ...)
```

`unEx` makes a `Tree.exp` even though e was `Cx`.

Implementation?

Translation of Abstract Syntax

Implementation of function unEx: exp => Tree.exp:

```
structure T = 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
```

Example: flag := x>y:
genstmt = fun (t,f) =>
CJUMP (GT, Temp x, Temp y, t_label, f_label)

Pseudocode

Temp r := 1;
CJUMP (GT, Temp x, Temp y, t_label, f_label);
f_label: Temp r := 0;
t_label: flag := r (*program continuation*)

Implementation of unNx and unCx similar.

Translation of Abstract Syntax

Primary result of sematic analysis:

- a type environment **TENV**: collects type declarations, ie maps type names to representations of type
- a value environment **VENV**: collects variable declarations, ie maps variable names x to either
  - a type (if x represents a non-function variable)
  - a lists of argument types, and a return type (if x represents a function)

But also: generate IR code

Tiger: translation functions transExpr, transVar, transDec, and transTy based on the syntactic structure of Tiger's Abysn.

In particular: TransExp returns record {Tree.exp, ty}.

IR code generation complex for Tiger

- don't want to be processor specific: abstract notion of frames, with abstract parameter slots ("access"): constructors **inFrame** or **inReg**,
- further abstraction layer to separate implementation of **translate** function from use of these functions in semantic analysis (type check)

IR code generation complex for Tiger

- don’t want to be processor specific: abstract notion of frames, with abstract parameter slots (“access”): constructors **inFrame** or **inReg**,
- further abstraction layer to separate implementation of **translate** function from use of these functions in semantic analysis (type check)

Root problem: escaped variables

Address-taken variables

Call by reference variables

Nested functions

Stack-allocated data structures (records)

:

IR code generation complex for Tiger

- don't want to be processor specific: abstract notion of frames, with abstract parameter slots ("access"): constructors **inFrame** or **inReg**,
- further abstraction layer to separate implementation of **translate** function from use of these functions in semantic analysis (type check)

Root problem: escaped variables

Address-taken variables

Call by reference variables

Nested functions

Stack-allocated data structures (records)

:

Absence of these features would make frame stack construction and hence IR emission MUCH easier

IR code generation complex for Tiger

- don't want to be processor specific: abstract notion of frames, with abstract parameter slots ("access"): constructors **inFrame** or **inReg**,
- further abstraction layer to separate implementation of **translate** function from use of these functions in semantic analysis (type check)

<u>Root problem: escaped variables</u>

Address-taken variables

Call by reference variables

Nested functions

Stack-allocated data structures (records)

:

Absence of these features would make frame stack construction and hence IR emission MUCH easier

Compiler design

Language design

Example of modern language that avoids these features: Java

IR code generation: "local" variable access

- **Case 1:** variable v declared in current procedure's frame

```
InFrame(k):
  MEM(BINOP(PLUS, TEMP(FP), CONST(k)))
```

```
k: offest in own frame
```

`FP` is declared in FRAME module.

- **Case 2:** variable v declared in temporary register

```
InReg(t_103):
  TEMP(t_103)
```

Choice as to which variables are inFrame and which ones are inReg is architecture-specific, so implemented inside FRAME module.
FRAME also provides mechanism to construct abstract activation records, containing one inFrame/inReg access for each formal parameter.

IR code generation: variable access

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

```
InFrame(k_n):
  MEM(BINOP(PLUS, CONST(k_n),
      MEM(BINOP(PLUS, CONST(k_n-1),
           ...
           MEM(BINOP(PLUS, CONST(k_2),
               MEM(BINOP(PLUS, CONST(k_1), TEMP(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,..,k_n-1`)
- When l_g reached, offset `k_n` used.

Thus, IR code generation for a function body can be done using uniform notion of "parameter slots" ("access") in an abstract description of Frames:

- interface of frame says for each parameter whether it's inFrame or inReg
- different implementations of Frame module can follow different policies
- given any Frame implementation, Translate generates suitable code

Array Access

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 Access

```
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~~ 2), CONST(w))))

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

Records: allocation and deallocation

Records can outlive function invocations, so

- allocation happens not on stack but on heap, by call to an other runtime function **ALLOC** to which a call is emitted by the compiler
  - should include code for (type-correct) initialization of components
  - details below

Records: allocation and deallocation

Records can outlive function invocations, so

- allocation happens not on stack but on heap, by call to an other runtime function **ALLOC** to which a call is emitted by the compiler
  - should include code for (type-correct) initialization of components
  - details below
- deallocation: no explicit instruction in the source language, so either
  - **no** deallocation (poor memory usage: memory leak), or
  - compiler has an **analysis** phase that (conservatively) estimates lifetime of records (requires alias analysis) and inserts calls to runtime function **FREE** at appropriate places, or
  - dynamic **garbage collection** (future lecture)

Similar issues arise for allocation/deallocation of arrays.

Conditional Statements – if e1 then e2 else e3

Approach 1: use CJUMP

- ok in principle, but doesn't work well if e1 contains &, |

Conditional Statements – if e1 then e2 else e3

Approach 1: use CJUMP

- ok in principle, but doesn’t work well if e1 contains &, |

Approach 2: exploit Cx constructor

- yields good code if e1 contains &, |
- treat e1 as Cx expression ➔ apply unCx
- use fresh labels as “entry points” for e2 and e3
- treat e2, e3 as Ex expressions ➔ apply unEx

Conditional Statements – if e1 then e2 else e3

Approach 1: use CJUMP

- ok in principle, but doesn't work well if e1 contains &, |

Approach 2: exploit Cx constructor

- yields good code if e1 contains &, |
- treat e1 as Cx expression ➔ apply unCx
- use fresh labels as "entry points" for e2 and e3
- treat e2, e3 as Ex expressions ➔ apply unEx

Pseudocode

```
"if e1 then JUMP t else JUMP f";
t: r := e2 (*code for e2, leaving result in r*)
   JUMP join
f: r := e3 (*code for e3, leaving result in r*)
join: … (*program continuation, can use r*)
```

Conditional Statements – if e1 then e2 else e3

Pseudocode

```
“if e1 then JUMP t else JUMP f”;
t: r := e2 (*code for e2, leaving result in r*)
   JUMP join
f: r := e3 (*code for e3, leaving result in r*)
join: … (*program continuation, can use r*)
```

```
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)))
```

Optimizations possible, e.g. if e2/e3 are themselves Cx expressions – see MCIL

Strings

Can think of as additional function definitions, for which the compiler silently generates code, too

- 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.

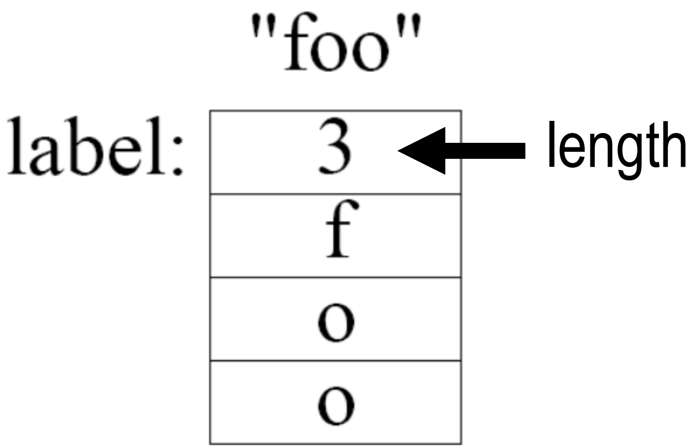
Strings

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)])
```

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 (3*w) ] ) ),
     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.

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
```

(Approx.) solution hinted to in MCIL:

```
in if lo <= hi
   then lab: body
        if i < limit
        then i++; JUMP lab
        else JUMP done
   else JUMP done
   done:
```

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: basic idea:

```
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
```

Declarations

Consider type checking of "let" expression: basic idea:

```
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
```

Complications:

- need auxiliary info level, break inside translation of body
- need to insert code for variable initialization.

Declarations

Consider type checking of "let" expression: basic idea:

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

Complications:

- need auxiliary info level, break inside translation of body
- need to insert code for variable initialization. Thus, transDecs is modified to additionally return an expression list **e** of assignment expressions that's inserted **HERE** (and 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 Prolog

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 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.

inFrame, inReg etc

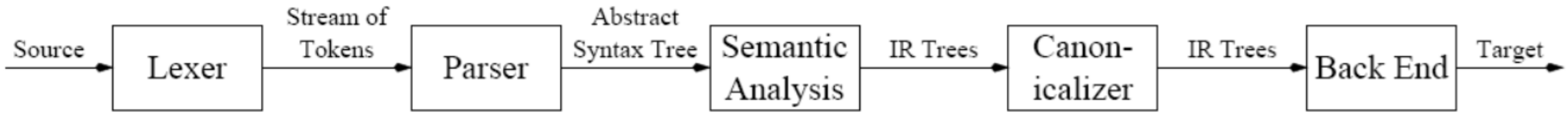
Problem with IR Trees

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: overview

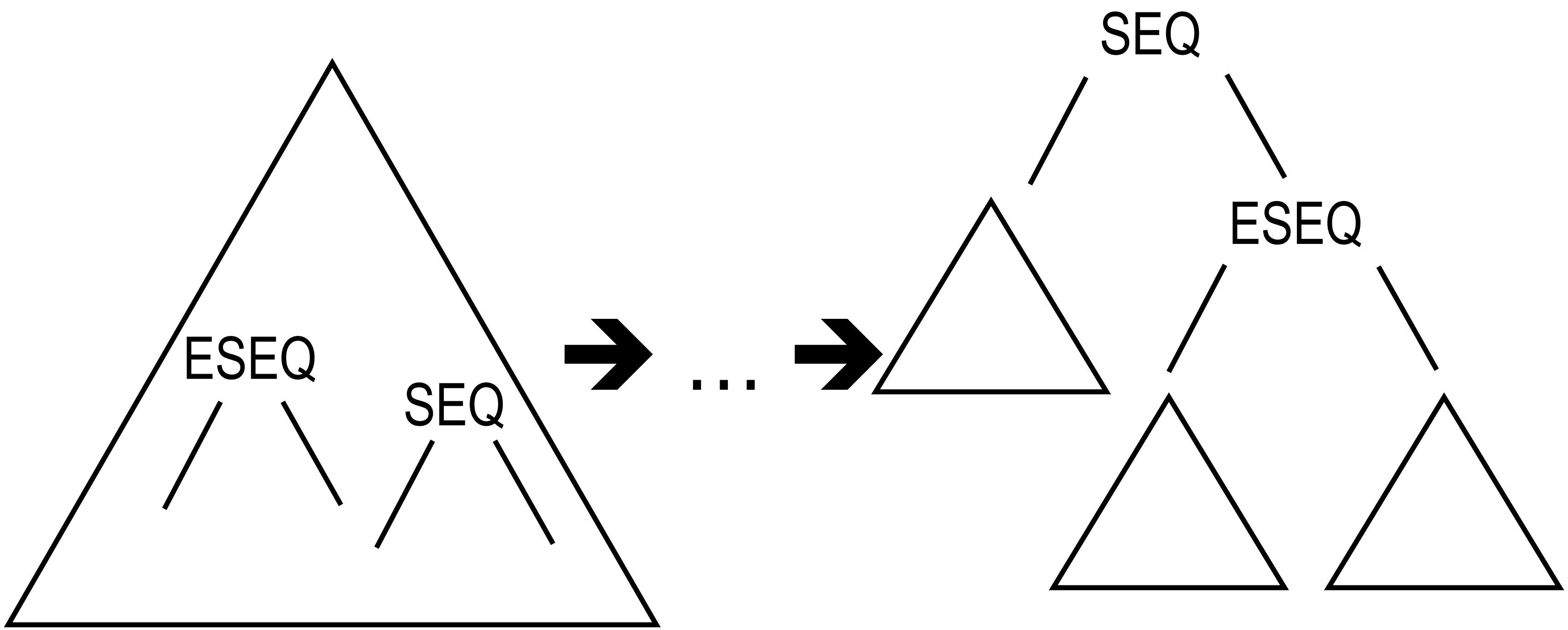


Canonicalizer takes `Tree.stm` for each function body, applies following transforms:

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(TEMP(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.

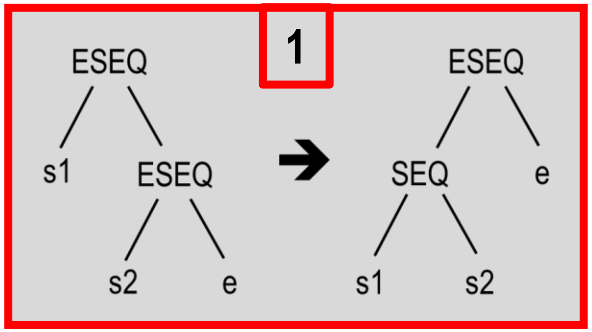
Elimination of (E)SEQs

Goal: Move ESEQ and SEQ nodes towards the top of a Tree.stm by repeatedly applying local rewrite rules

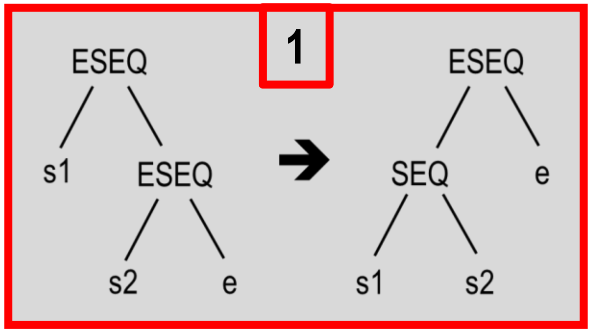


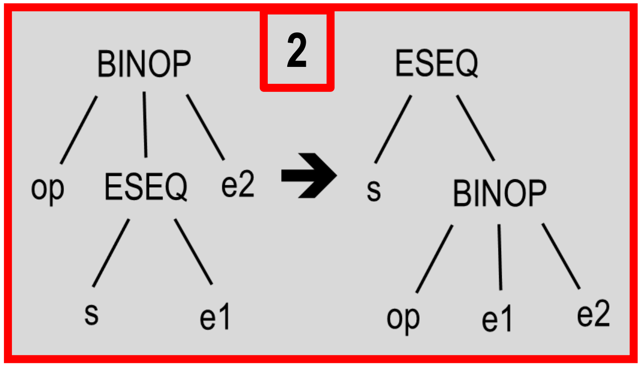
(selected rewrite rules on next slides)

Rewrite rules



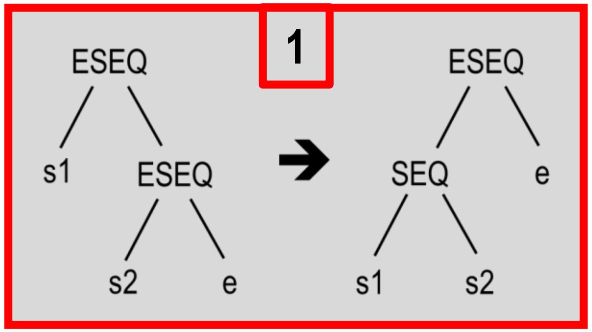
Rewrite rules

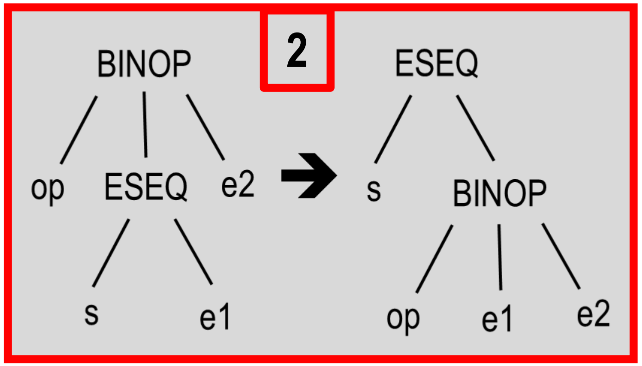




(also for MEM, JUMP, CJUMP in place of BINOP)

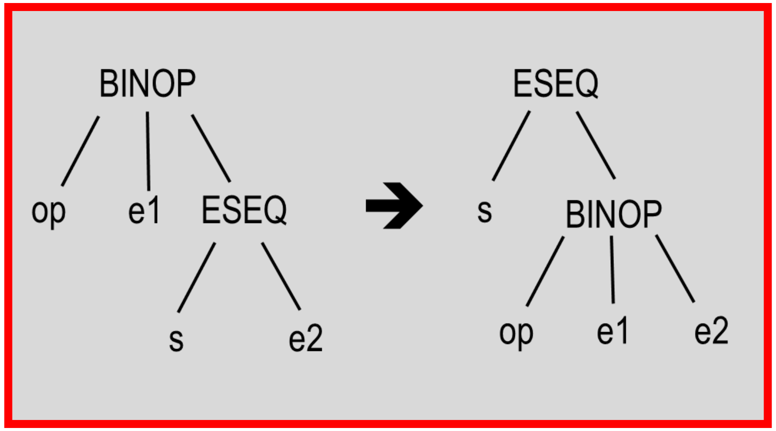
Rewrite rules



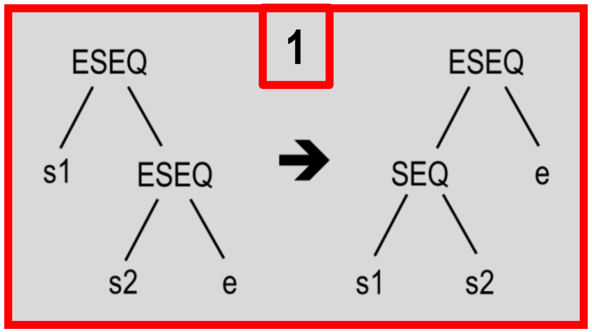


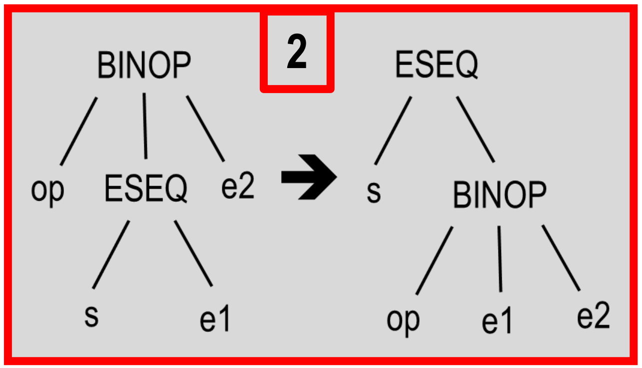
(also for MEM, JUMP, CJUMP in place of BINOP)

What about this:



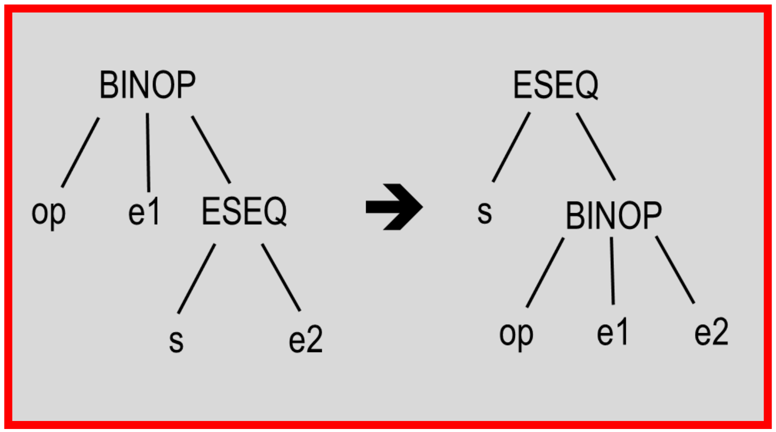
Rewrite rules





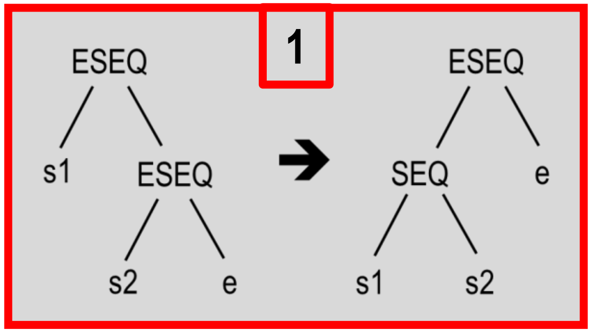
(also for MEM, JUMP, CJUMP in place of BINOP)

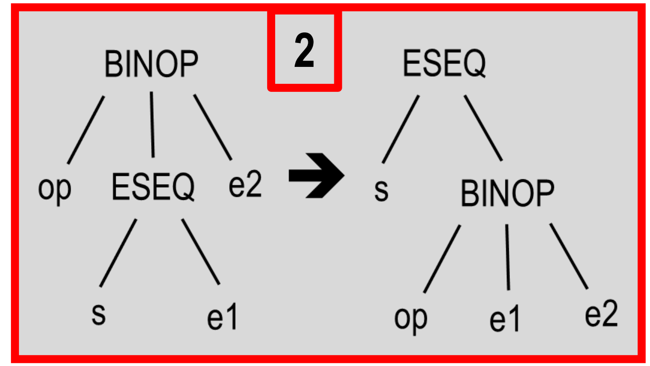
What about this:



Incorrect if s contains assignment to a variable read by e1!

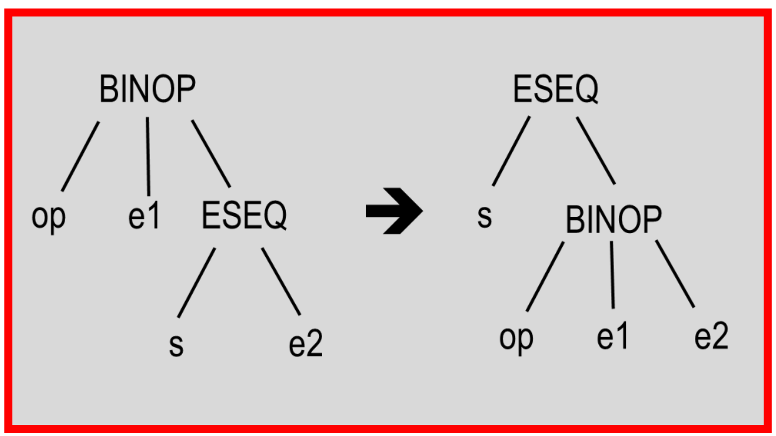
Rewrite rules





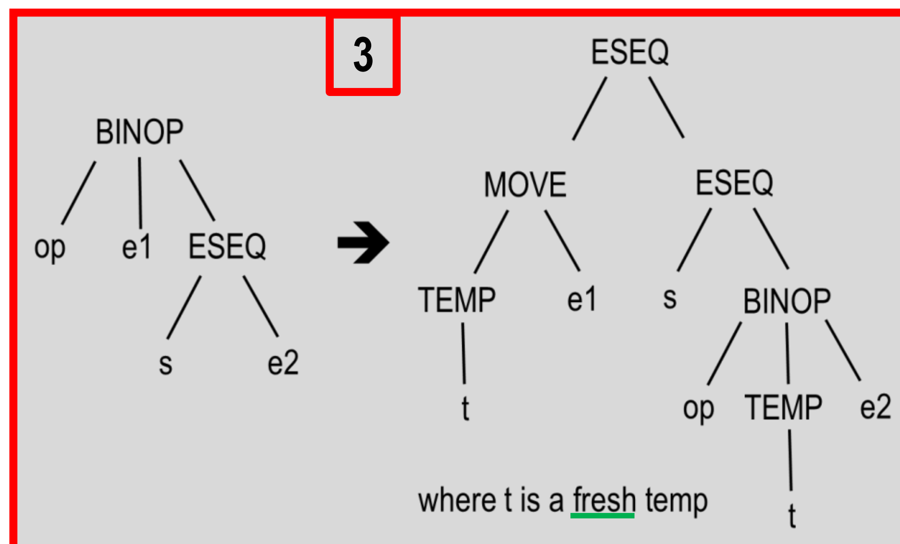
(also for MEM, JUMP, CJUMP in place of BINOP)

What about this:



Incorrect if s contains assignment
to a variable read by e1!

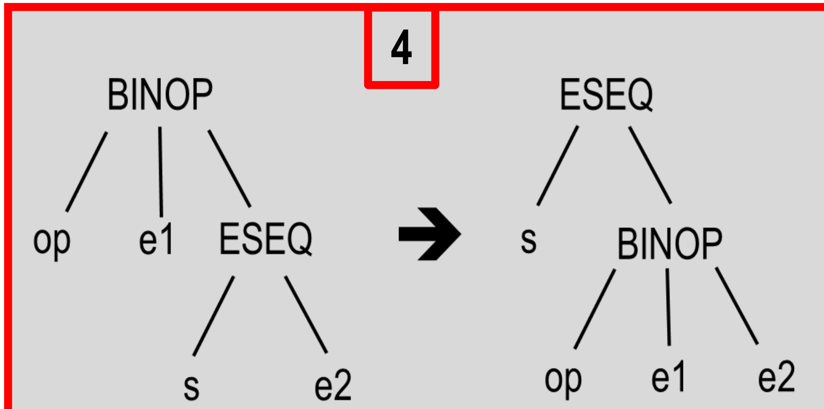
General Solution:



(also for CJUMP in place of BINOP)

Rewrite Rules

Specific solution:



In fact correct if s and e1 **commute**!

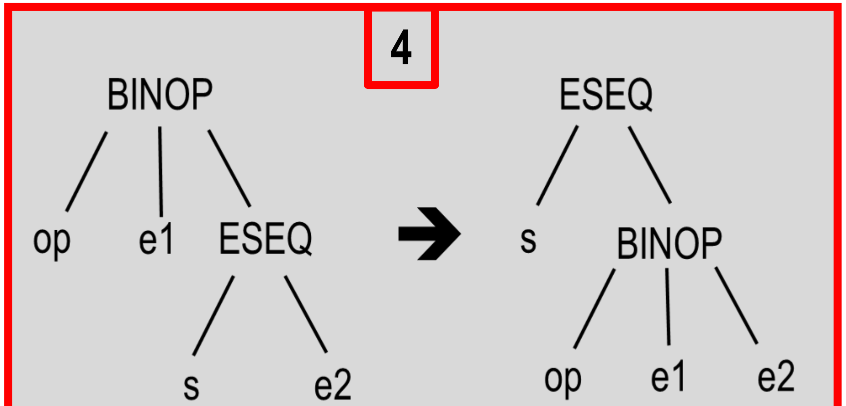
(Similarly for CJUMP)

<u>When do s and e commute?</u>

- variables and memory locations accessed by s are **disjoint** from those accessed by e
- no disjointness but all accesses to such a shared resource are READ

Rewrite Rules

Specific solution:



In fact correct if s and e1 **commute**!

(Similarly for CJUMP)

When do s and e commute?

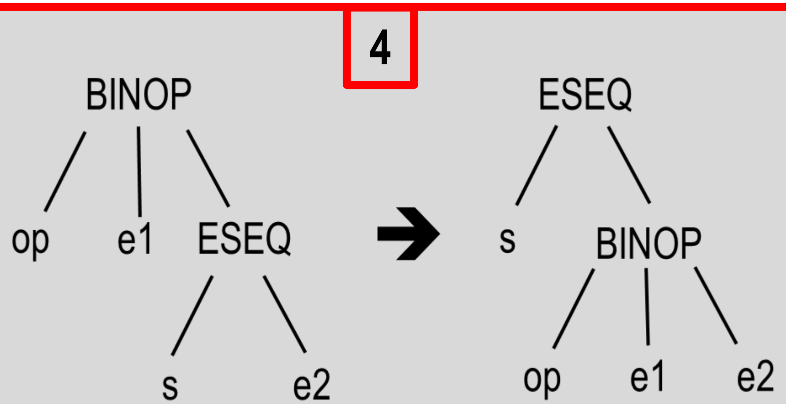
- variables and memory locations accessed by s are **disjoint** from those accessed by e
- no disjointness but all accesses to such a shared resource are READ

But: deciding whether MEM(x) and MEM(z) represent the same location requires deciding whether x and z may be equal

Undecidable in general!

Rewrite Rules

Specific solution:



In fact correct if s and e1 **commute**!

(Similarly for CJUMP)

When do s and e commute?

- variables and memory locations accessed by s are **disjoint** from those accessed by e
- no disjointness but all accesses to such a shared resource are READ

But: deciding whether MEM(x) and MEM(z) represent the same location requires deciding whether x and z may be equal

Undecidable in general!

Solution:

Compiler conservatively approximates disjointness / commutability, ie performs the rewrite cautiously

example: e1 == CONST(i)

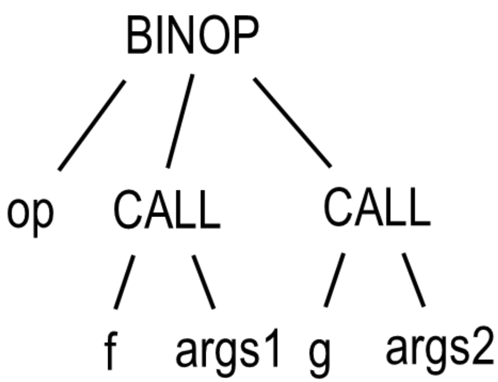
Rewrite Rules

Goal 2: ensure that parent of a CALL is EXP (…) or MOVE(TEMP t, …)

Motivation: calls leave their result in dedicated register rv. Now consider tree **T**.

What could go wrong?

T:

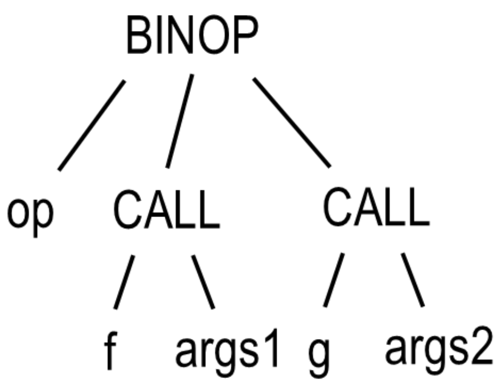


Rewrite Rules

Goal 2: ensure that parent of a CALL is EXP (…) or MOVE(TEMP t, …)

Motivation: calls leave their result in dedicated register rv. Now consider tree **T**.

T:



What could go wrong? Call to g will overwrite result of f (held in rv) before BINOP is executed!

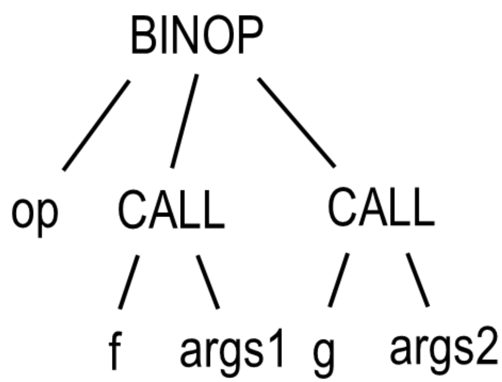
Solution?

Rewrite Rules

Goal 2: ensure that parent of a CALL is EXP (…) or MOVE(TEMP t, …)

Motivation: calls leave their result in dedicated register rv. Now consider tree **T**.

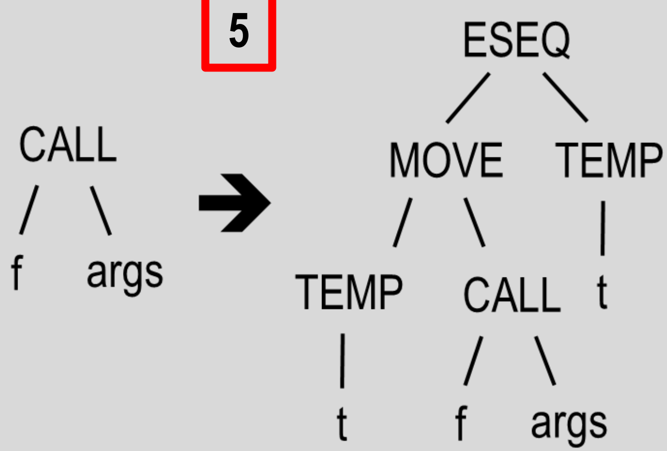
T:



What could go wrong? Call to g will overwrite result of f (held in rv) before BINOP is executed!

Solution: save result of call to f before calling g, to avoid overwriting rv!

5



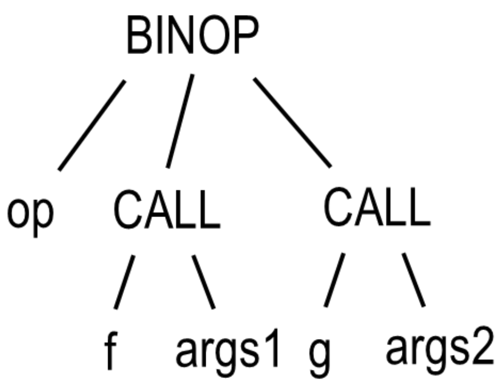
(where t is a fresh temp; don't apply recursively under the pattern MOVE(TEMP(..), CALL(..)…))

Rewrite Rules

Goal 2: ensure that parent of a CALL is EXP (…) or MOVE(TEMP t, …)

Motivation: calls leave their result in dedicated register rv. Now consider tree **T**.

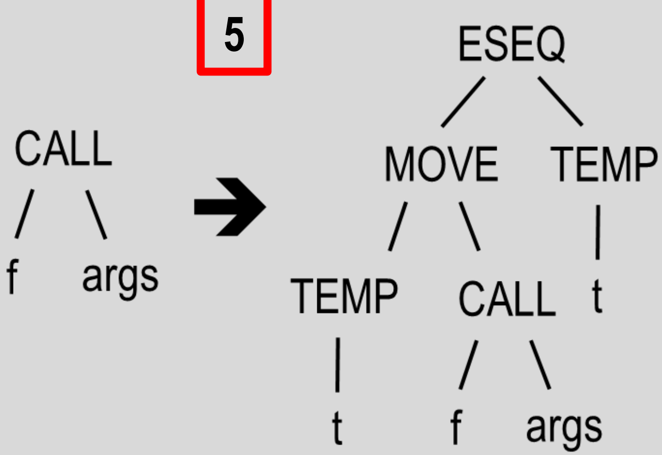
T:



What could go wrong? Call to g will overwrite result of f (held in rv) before BINOP is executed!

Solution: save result of call to f before calling g, to avoid overwriting rv!

5



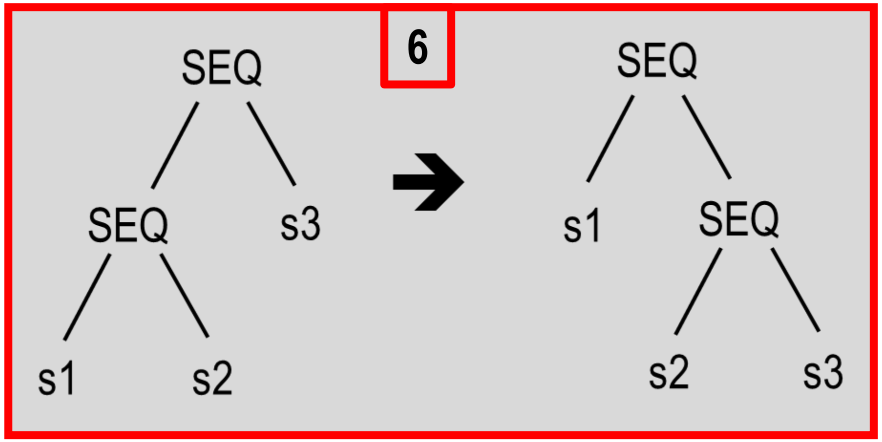
(where t is a fresh temp; don't apply recursively under the pattern MOVE(TEMP(..), CALL(..)…))

Homework 1: apply rules 1-5 to rewrite **T** until no more rules can be applied. Is your solution optimal?

Homework 2: can you think of additional rules, for nested calls?

Elimination of SEQs

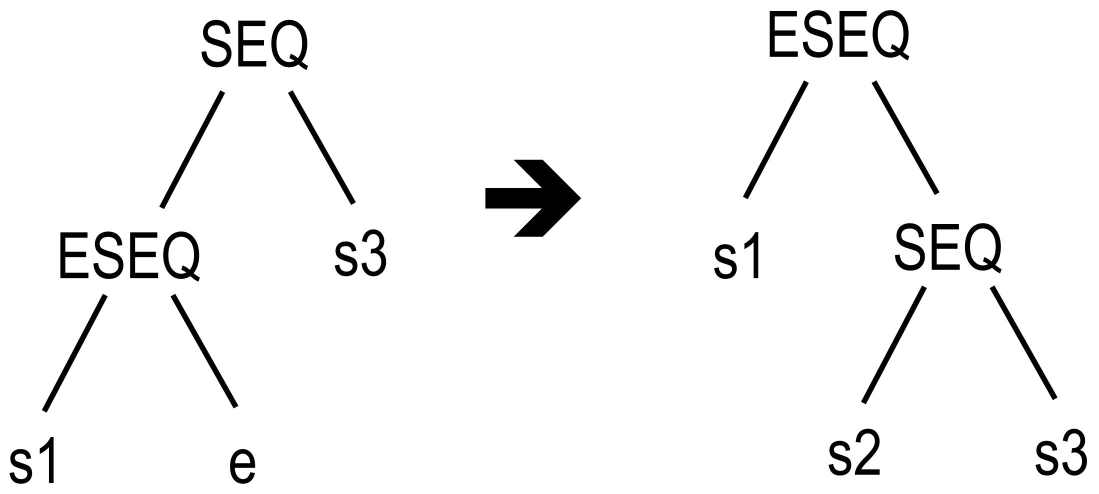
Associativity of SEQ:



Final step: once all SEQ's are at top of tree, collect list of statements left-to-right

Elimination of SEQs

Associativity of SEQ:



Final steps: once all SEQ's are at top of tree, extract list of statements left-to-right

Midterm exam

End of lecture material that's relevant for the midterm.

Thus, MCIL material up to (and including) Section 8.1 "Canonical Trees" is fair game.

Normalization of branches

Remember conditional branch instruction of TREE:

"true" branch

"false" branch

CJUMP (relop, exp, exp, label, label)

Assembly languages: conditional branches typically have only one label

So need to analyze **control flow** of program: what's the order in which an execution might "walk through program", ie execute instructions?

Normalization of branches

Remember conditional branch instruction of TREE:

"true" branch "false" branch

CJUMP (relop, exp, exp, label, label)

Assembly languages: conditional branches typically have only one label

So need to analyze **control flow** of program: what's the order in which an execution might "walk through program", ie execute instructions?

- sequence of non-branching instructions: trivial, in sequential order
- unconditional jumps: obvious – follow the goto
- CJMUP: cannot predict outcome, so need to assume either branch may be taken

➔ For analysis of control flow, can consider sequences of non-branching instructions as single node ("**basic block**")

Basic blocks

A basic block is a sequence of statements such that

- the first statement is a LABEL instruction
- the last statement is a JUMP or CJUMP
- there are no other LABELSs, JUMPs, or CJUMPs

Basic blocks

A basic block is a sequence of statements such that

- the first statement is a LABEL instruction
- the last statement is a JUMP or CJUMP
- there are no other LABELSs, JUMPs, or CJUMPs

Task: partition a sequence of statements (Ln: LABEL n; si = straight-line stmt)

L1	s1	s2	s3	CJUMP .. (L1, L3)	L2	s4	JUMP L1	L3	s5	s6	s7	JUMP L2

into a sequence of basic blocks

L1	s1	s2	s3	CJUMP .. (L1, L3)

L2	s4	JUMP L1

L3	s5	s6	s7	JUMP L2

Partitioning into basic blocks

Naïve algorithm:

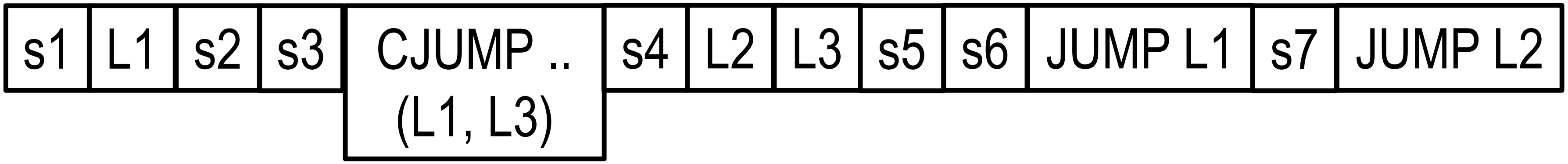
- traverse left-to-right
- whenever a LABEL is found, start a new BB (and end current BB)
- whenever a JUMP or CJUMP is found, end current BB (and start new BB)

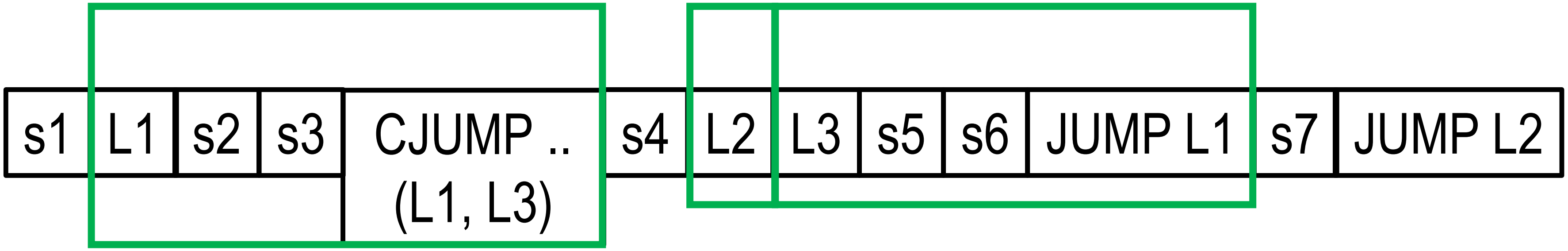
s1	L1	s2	s3	CJUMP .. (L1, L3)	s4	L2	L3	s5	s6	JUMP L1	s7	JUMP L2

Partitioning into basic blocks

Naïve algorithm:

- traverse left-to-right
- whenever a LABEL is found, start a new BB (and end current BB)
- whenever a JUMP or CJUMP is found, end current BB (and start new BB)

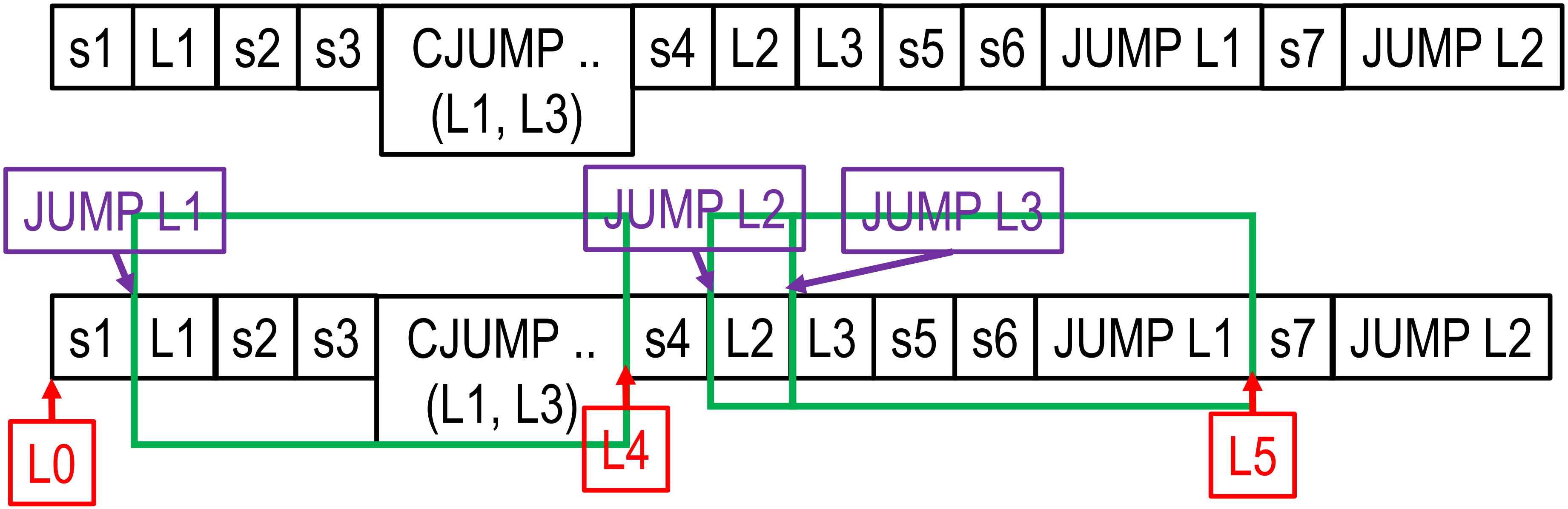




Partitioning into basic blocks

Better algorithm:

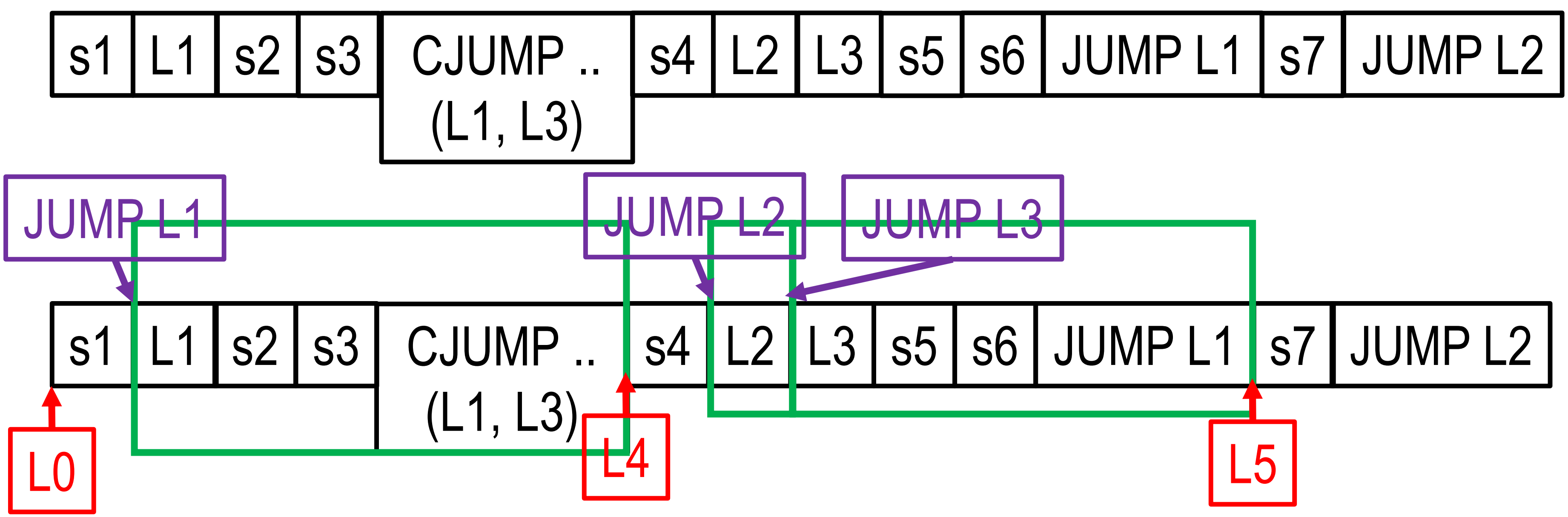
- traverse left-to-right
- whenever a LABEL is found, start a new BB (and end current BB)
- whenever a JUMP or CJUMP is found, end current BB (and start new BB)
- insert fresh LABELs at beginnings of BBs that don't start with a LABEL
- insert JUMPs at ends of BBs that don't end with a JUMP or CJUMP



Partitioning into basic blocks

Better algorithm:

- traverse left-to-right
- whenever a LABEL is found, start a new BB (and end current BB)
- whenever a JUMP or CJUMP is found, end current BB (and start new BB)
- insert fresh LABELs at beginnings of BBs that don't start with a LABEL
- insert JUMPs at ends of BBs that don't end with a JUMP or CJUMP
- convenient to also add a special LABEL D for epilogue and add JUMP D



Ordering basic blocks

Given that basic blocks have entry labels and jumps at end

- relative order of basic blocks irrelevant
- so reorder to ensure (if possible) that a block ending in
  - CJUMP is followed by the block labeled with the “FALSE” label
  - JUMP is followed by its target label

More precisely: cover the collection of basic blocks by a set of **traces**:

- sequences of stmts (maybe including jumps) that are potentially executed sequentially
- aims:
  - have each basic block covered by only one trace
  - use low number of traces in order to reduce number of JUMPS