
Topic 7: Intermediate Representation and Instruction Selection

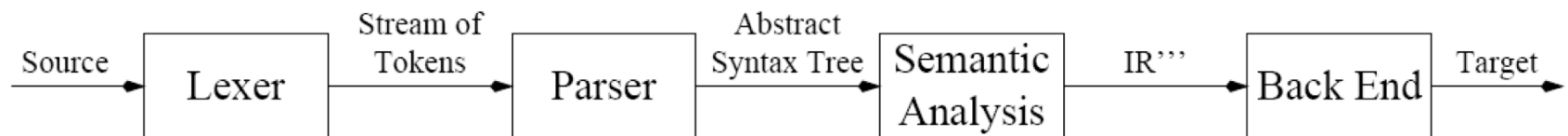
COS 320

Compiling Techniques

Princeton University
Spring 2018

Prof. David August

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

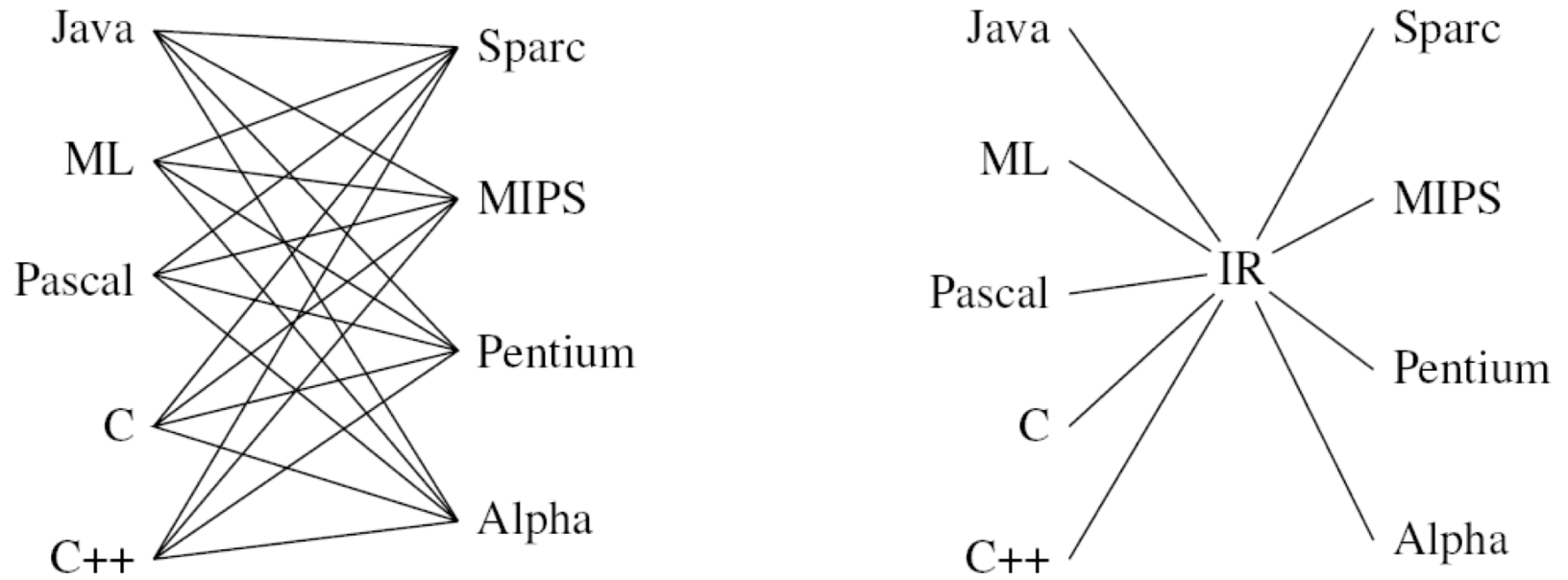


FIGURE 7.1. Compilers for five languages and four target machines:
(left) without an IR, (right) with an IR.

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:

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
```

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
```

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

$\text{MEM}(e)$ 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

$\text{CALL}(f, l)$ application of function f to argument list l

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

$\text{ESEQ}(s, e)$ 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 starting 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.

Translation of Abstract Syntax

- 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`

```
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)

Conditional:

$x > y$:

$Cx(fn\ (t, f) \Rightarrow CJUMP(GT, x, y, t, f))$

$a > b \mid c < d$:

$Cx(fn\ (t, f) \Rightarrow SEQ(CJUMP(GT, a, b, t, z),$
 $SEQ(LABEL\ z, CJUMP(LT, c, d, t, f))))$

May need to convert conditional to value:

$a := x > y$:

Cx corresponding to “ $x > y$ ” must be converted into $Tree.exp\ e$.

$MOVE(TEMP(a), e)$

Need three conversion functions:

$val\ unEx: exp \rightarrow Tree.exp$

$val\ unNx: exp \rightarrow Tree.stm$

$val\ unCx: exp \rightarrow (Temp.label * Temp.label \rightarrow Tree.stm)$

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`.

Translation of Abstract Syntax

Implementation of function UnEx:

```
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
```

Translation of Abstract Syntax

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

Simple Variables

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

InFrame(k) :

MEM(BINOP(PLUS, TEMP(FP), CONST(k)))

k : offset in own frame

FP is declared in FRAME module.

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

InReg(t_{103}) :

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

```
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, \dots, 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, \dots, k_{n-1})
- When l_g reached, offset k_n used.

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 :

$a[i]$:

$\text{MEM}(\text{BINOP}(\text{PLUS}, e, \text{BINOP}(\text{MUL}, i, \text{CONST}(w))))$

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), 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 \Rightarrow apply unCx.
- Treat e_2, e_3 as Ex expressions \Rightarrow 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) ) )
```

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’

"foo"

label:	3
	f
	o
	o

# 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(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

---

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

- $sl$  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

---

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?

# 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.

# Problem with IR Trees

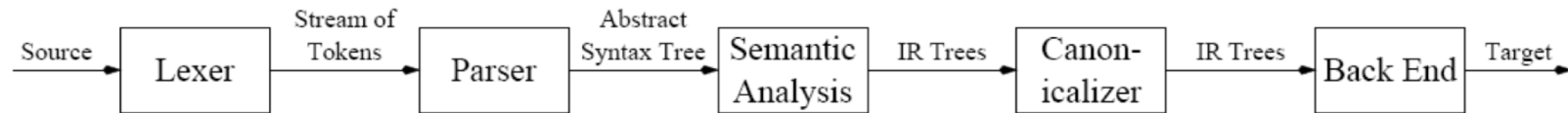
---

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

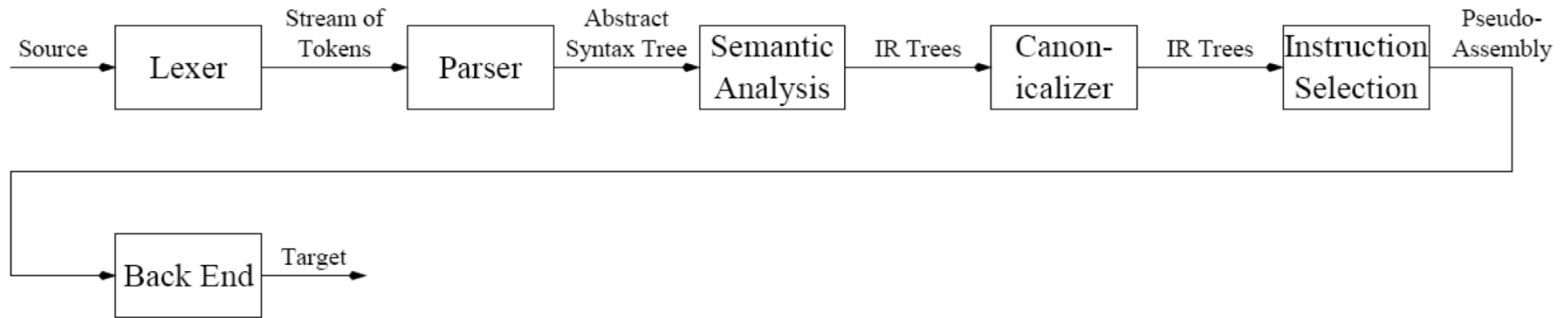


Canonicalizer takes `Tree.stm` for each function body, applies following transforms:

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.

# 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  $\rightarrow$  *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.

# Our Architecture

---

Arithmetic:

ADD  $r_d = r_{s1} + r_{s2}$

ADDI  $r_d = r_s + c$

SUB  $r_d = r_{s1} - r_{s2}$

SUBI  $r_d = r_s - c$

MUL  $r_d = r_{s1} * r_{s2}$

DIV  $r_d = r_{s1} / r_{s2}$

Memory:

LOAD  $r_d = M[r_s + c]$

STORE  $M[r_{s1} + c] = r_{s2}$

MOVEM  $M[r_{s1}] = M[r_{s2}]$

# Pseudo-ops

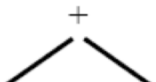

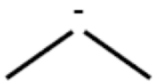
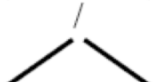
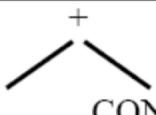

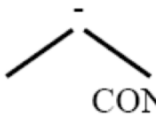
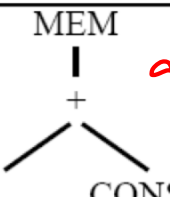
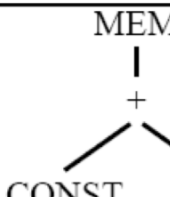


---

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

MOV	$r_d = r_s$	ADDI	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD	$r_d = r_{s1} + r_0$
MOVI	$r_d = c$	ADDI	$r_d = r_0 + c$

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

# Instruction Tree Patterns

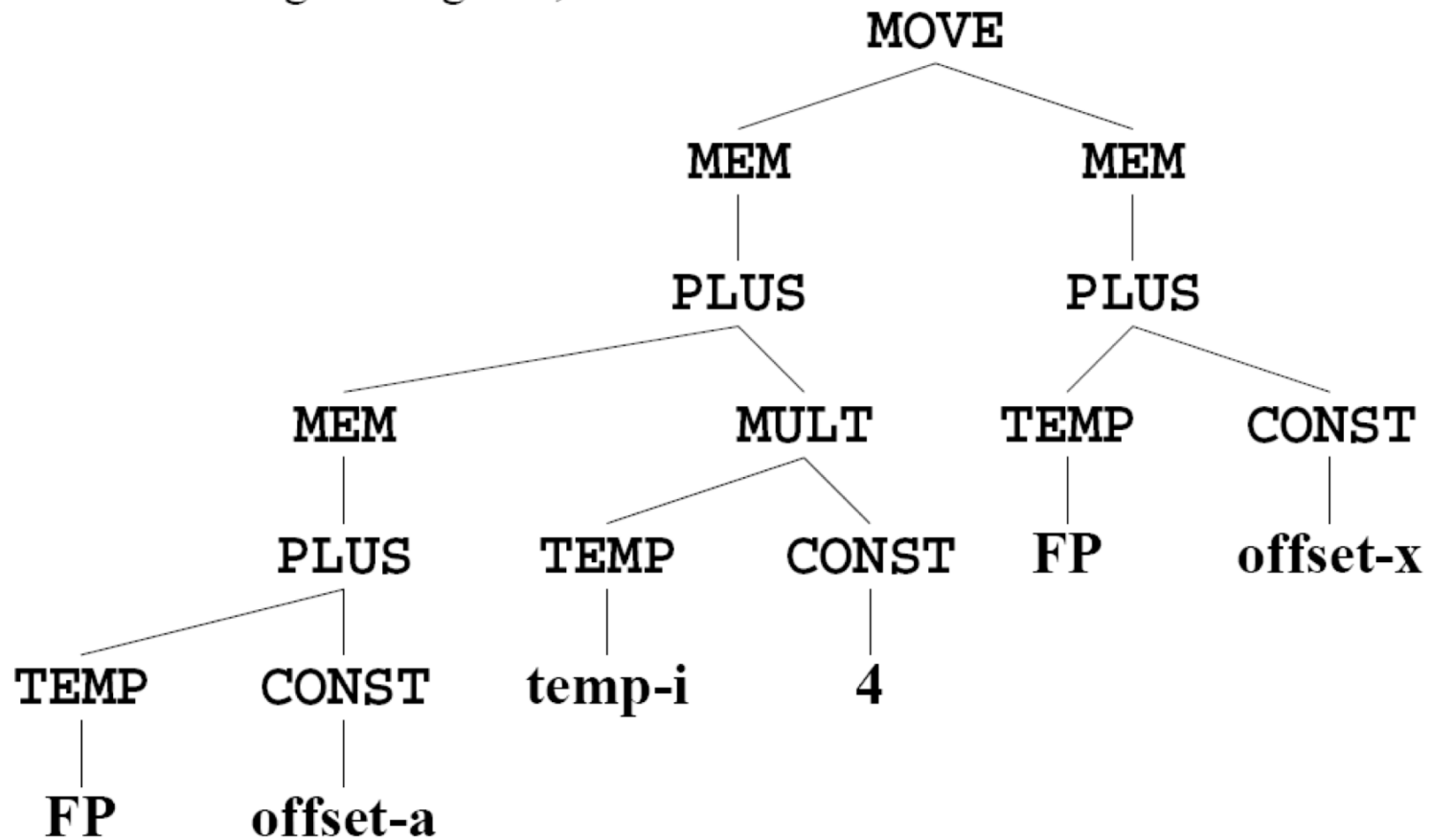
Name	Effect	Trees
—	$r_i$	TEMP 0
ADD	$r_i \quad r_j + r_k$	 1
MUL	$r_i \quad r_j \times r_k$	 2
SUB	$r_i \quad r_j - r_k$	 3
DIV	$r_i \quad r_j / r_k$	 4
ADDI	$r_i \quad r_j + c$	 5  6      CONST 7
SUBI	$r_i \quad r_j - c$	 8
LOAD	$r_i \quad M[r_j + c]$	 9  10  11  12

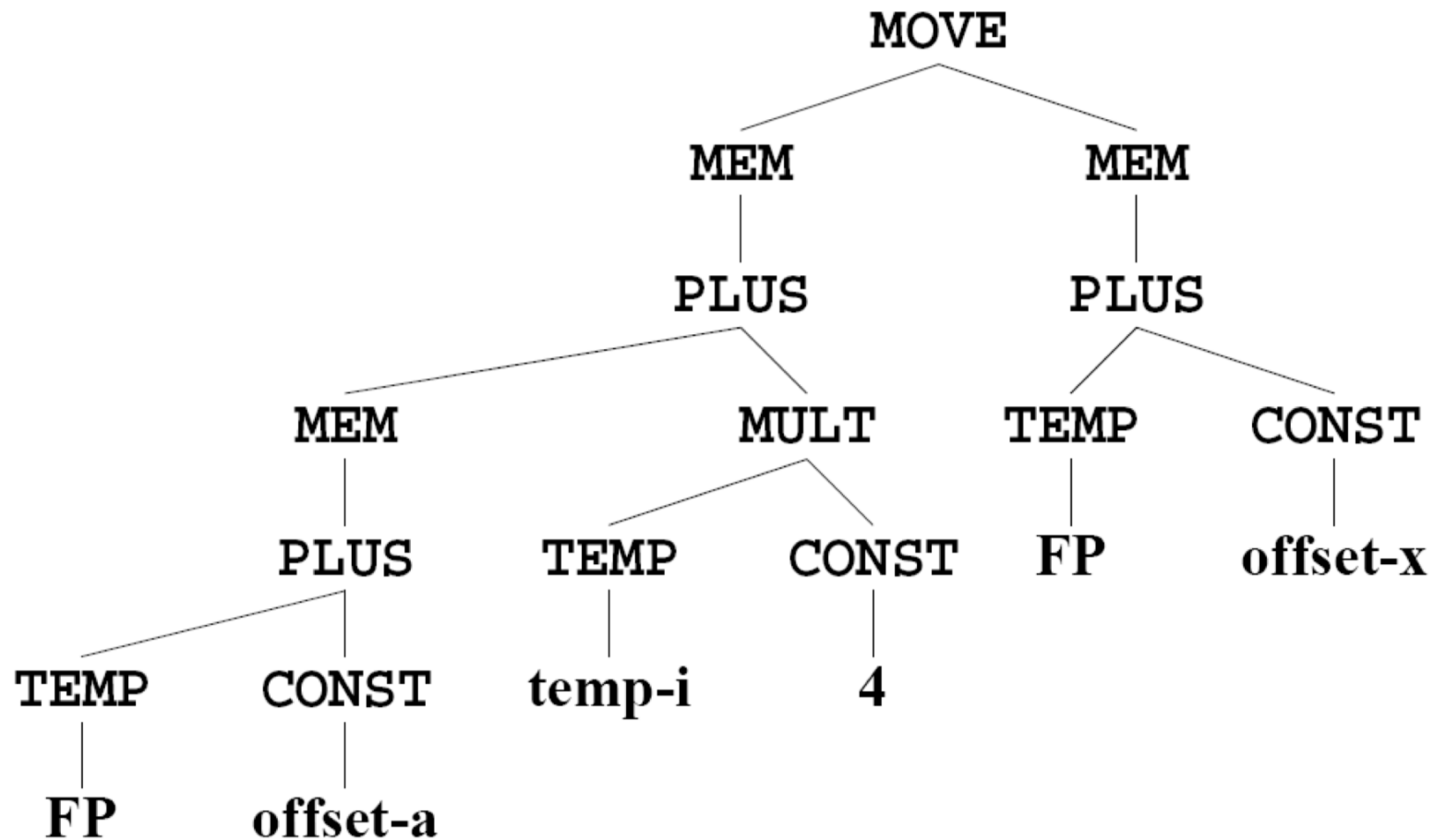
# Instruction Tree Patterns

STORE	$M[r_j + c]$	$r_i$	
MOVEM	$M[r_j]$	$M[r_i]$	

# Example

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





# 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

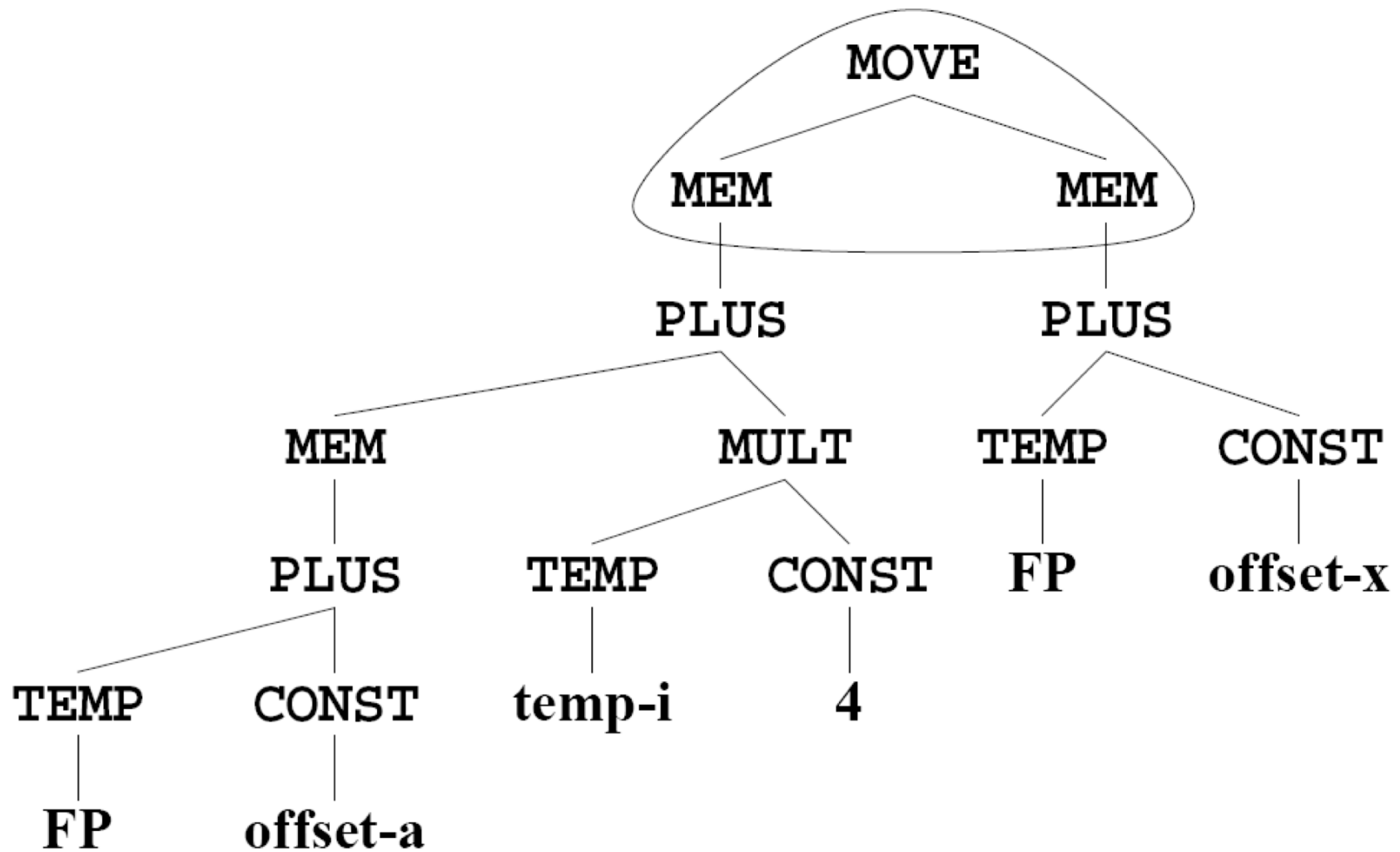
LOAD    r9 = M[r8 + 0]

STORE M[r6 + 0] = r9

**9 registers, 10 instructions**

# Random Tiling

---



# 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 ( $9 \rightarrow 8$ ) and an instruction ( $10 \rightarrow 9$ ).**

# Node Selection

---

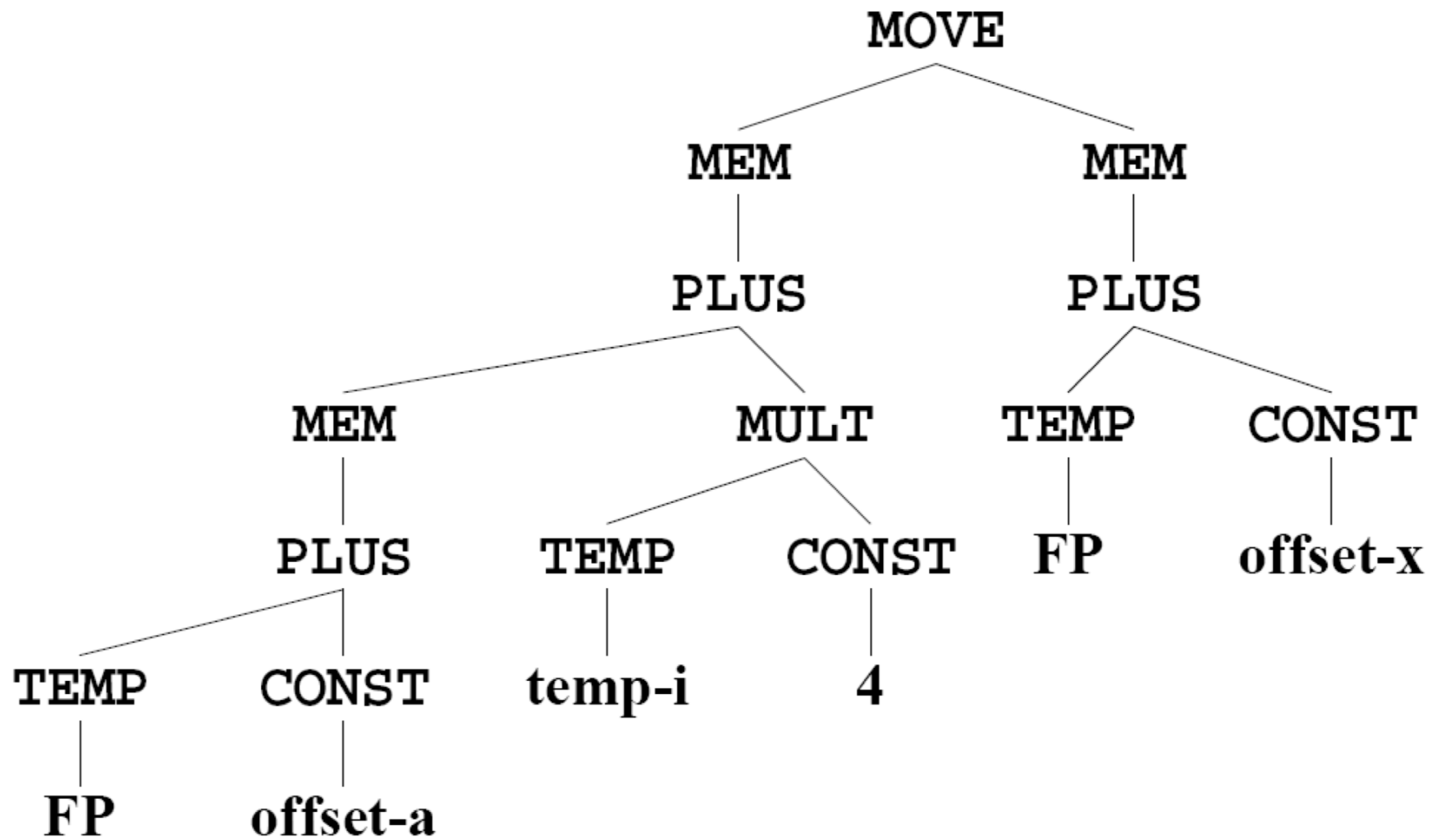
- There exist many possible tilings - want tiling/covering that results in instruction sequence of *least cost*
  - Sequence of instructions that takes least amount of time to execute.
  - For single issue fixed-latency machine: fewest number of instructions.
- Suppose each instruction has fixed cost:
  - *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  $\Rightarrow$  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
```

# Codegen

---

```
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

---

```
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]\n",
 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=[],
 jump=NONE})
...
)
```

# Expression Munch

---

```
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

---

```
| munchExp (T.BINOP (T.PLUS, e1, T.CONST (c))) =
 let
 val t = Temp.newtemp()
 in
 emit (Assem.OPER { assem="ADDI 'd0 = 's0 +" ^
 int (c) ^ "\n",
 src=[munchExp (e1)],
 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:

$$c_t + \sum_{\text{all leaves } i \text{ of } t} c_i$$

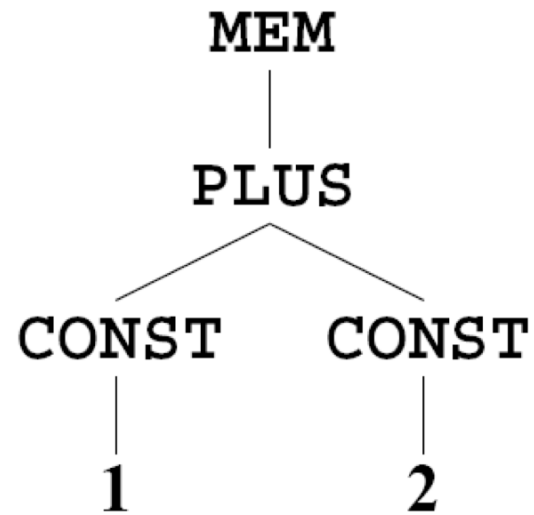
- Tile is chosen which has minimum cost.

# Optimum Instruction Selection – Example

---

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

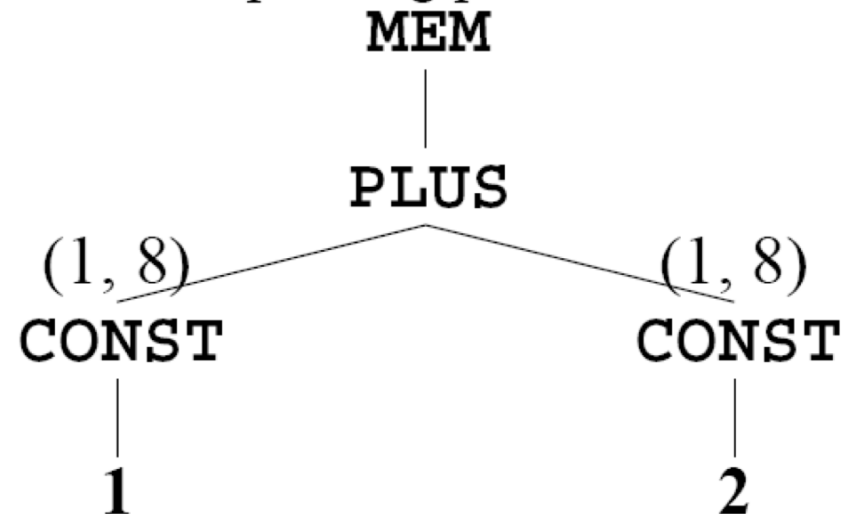
MEM (PLUS (CONST (1), CONST (2)))



# Optimum Instruction Selection – Example

## Step 1: Find cost of root node

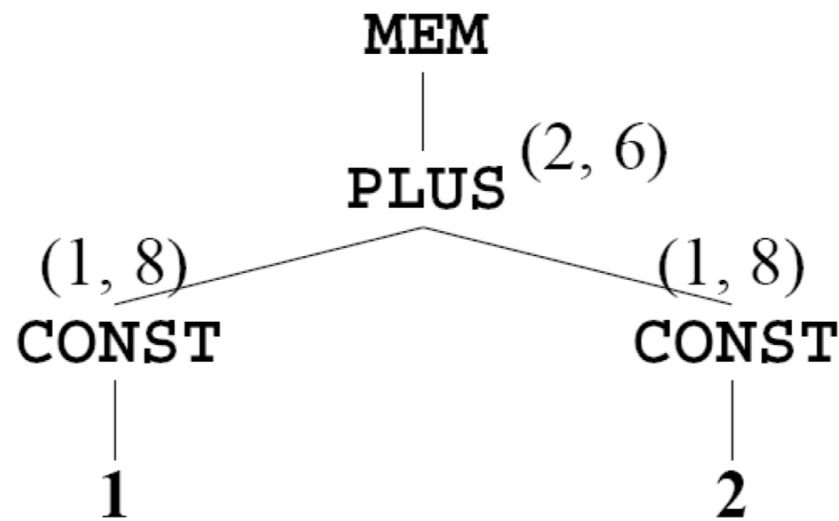
(a,b): a is minimum cost, b is corresponding pattern number



Consider PLUS node:

Pattern	Cost	Leaves Cost	Total
(2) PLUS(e1, e2)	1	2	3
(6) PLUS(CONST(c), e1)	1	1	2
(7) PLUS(e1, CONST(c))	1	1	2

# Optimum Instruction Selection – Example

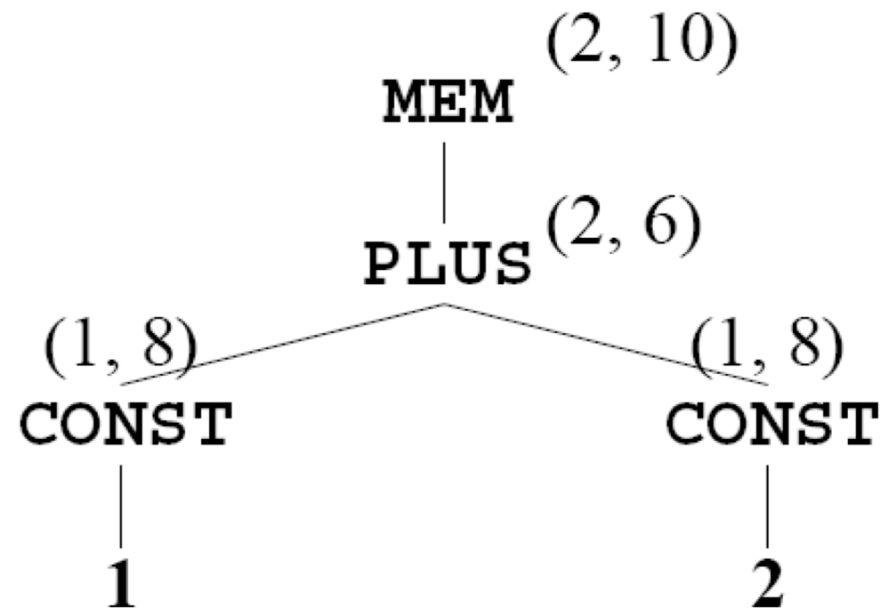


Consider MEM node:

Pattern	Cost	Leaves Cost	Total
(13) MEM(e1)	1	2	3
(10) MEM(PLUS(e1, CONST(c)))	1	1	2
(11) MEM(PLUS(CONST(c), e1))	1	1	2

# Optimum Instruction Selection – Example

---



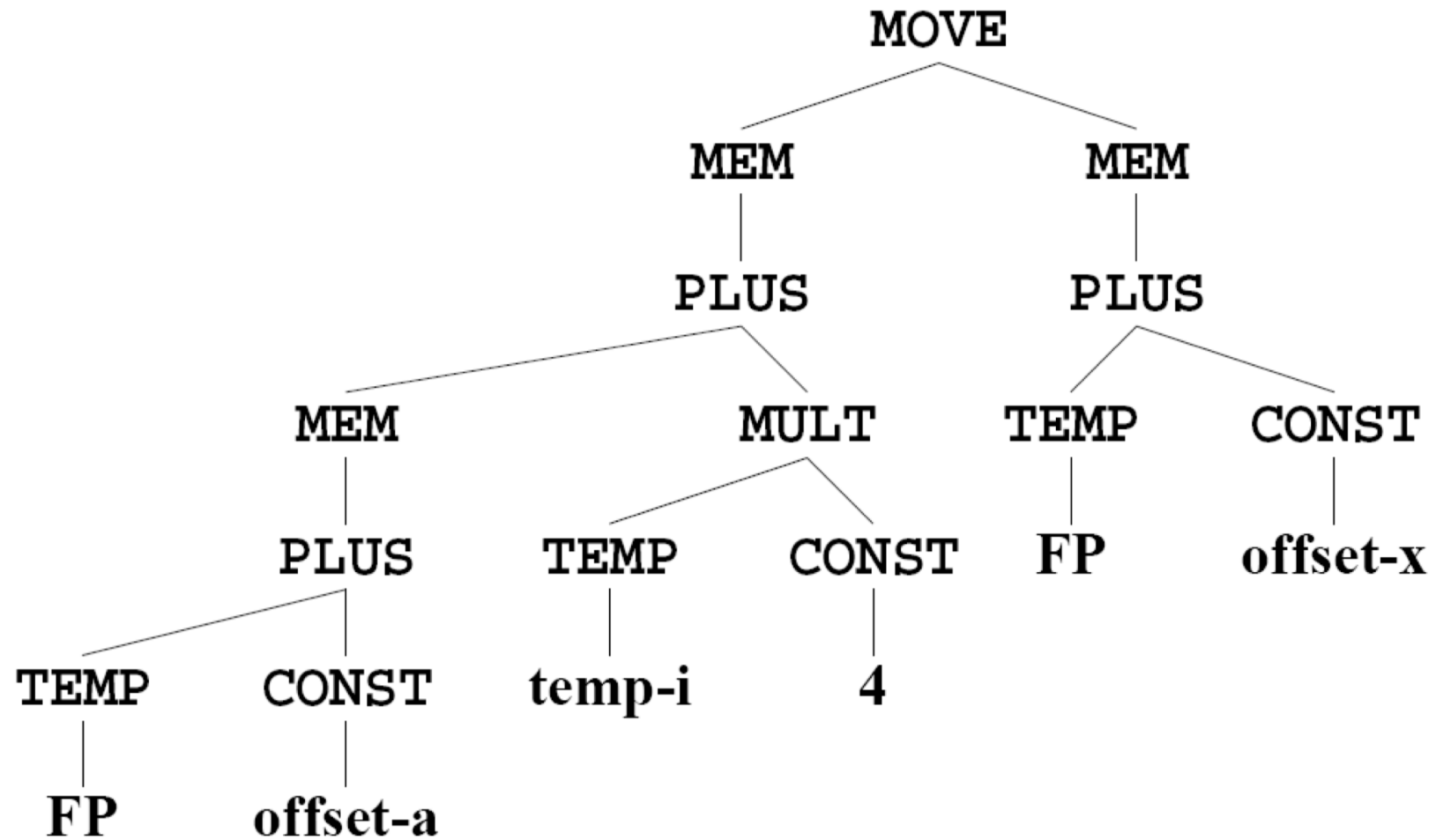
## Step 2: Emit instructions

ADDI r1 = r0 + 1

LOAD r2 = M[r1 + 2]

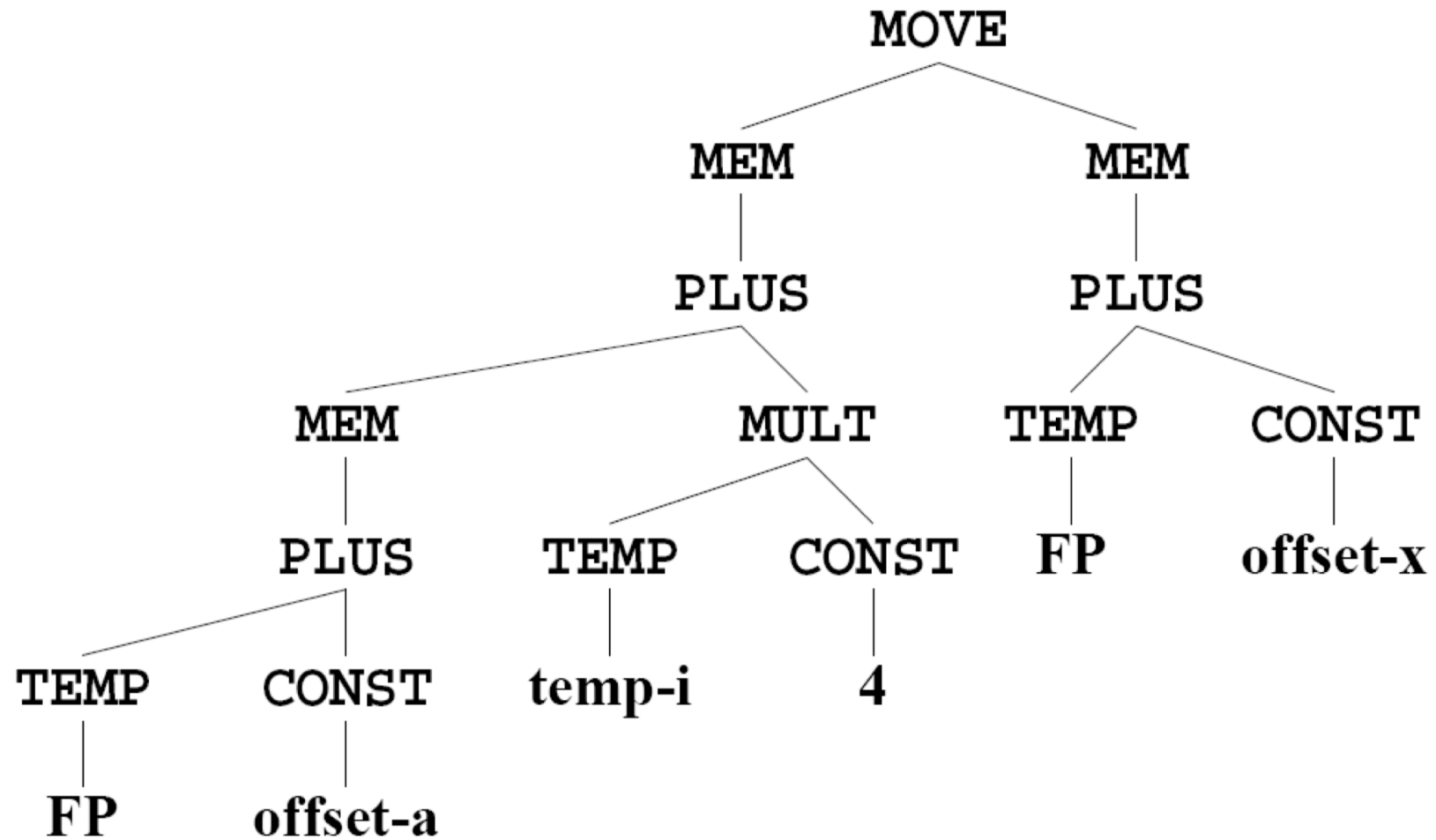
# Optimum Instruction Selection – Big Example

---



# Optimum Instruction Selection – Big Example

---



# Optimum Instruction Selection – Big Example

---

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...



