Lecture 19

Instruction Selection: Tree-pattern matching

(EaC-11.3)

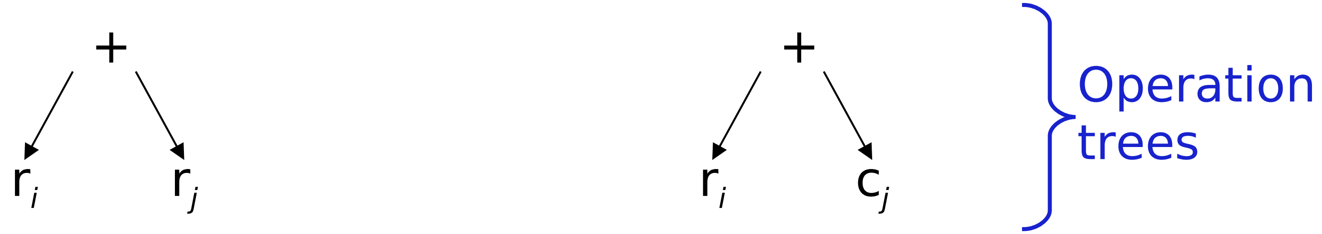
The Concept

Many compilers use tree-structured IRs

- Abstract syntax trees generated in the parser
- Trees or DAGs for expressions

These systems might well use trees to represent target ISA

Consider the add operators

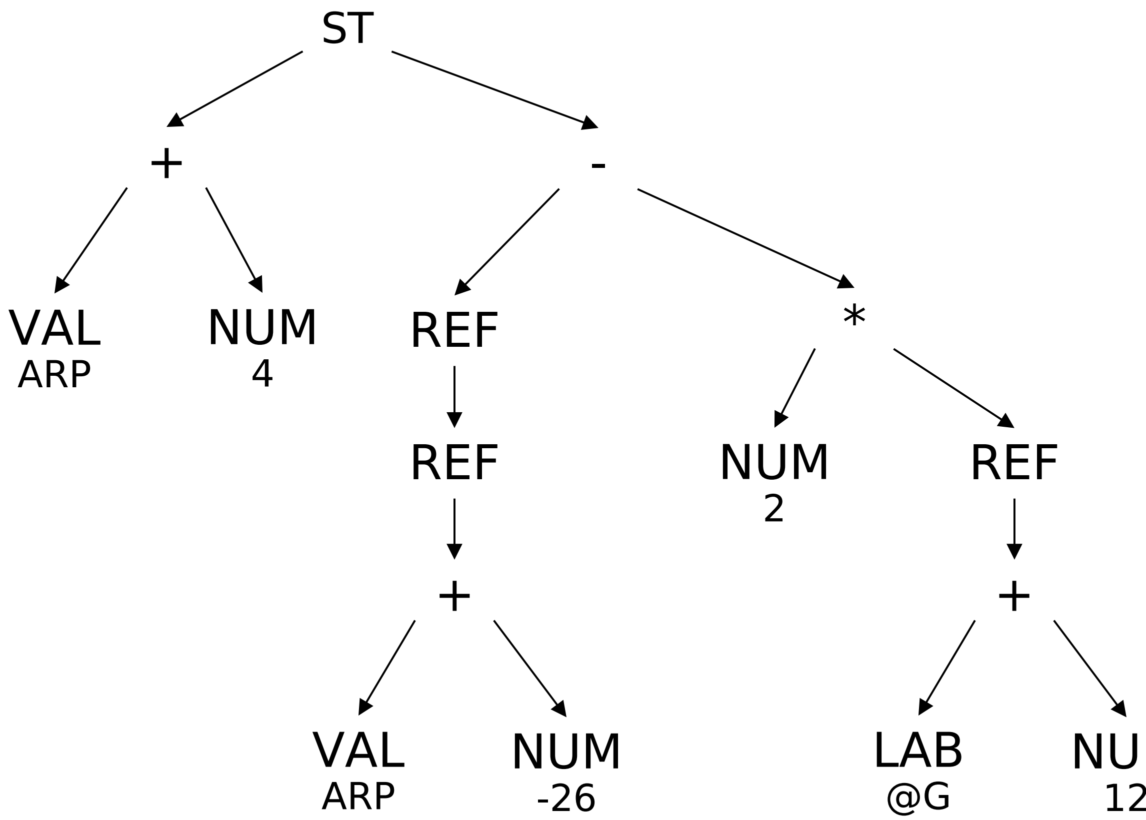


`add` $r_i, r_j \Rightarrow r_k$ `addI` $r_i, c_j \Rightarrow r_k$

If we can match these “pattern trees” against IR trees, …

The Concept

Low-level AST for w ← (*x) - 2 * y



ARP: r_{arp}

NUM: constant

LAB: ASM label

w: at ARP+4

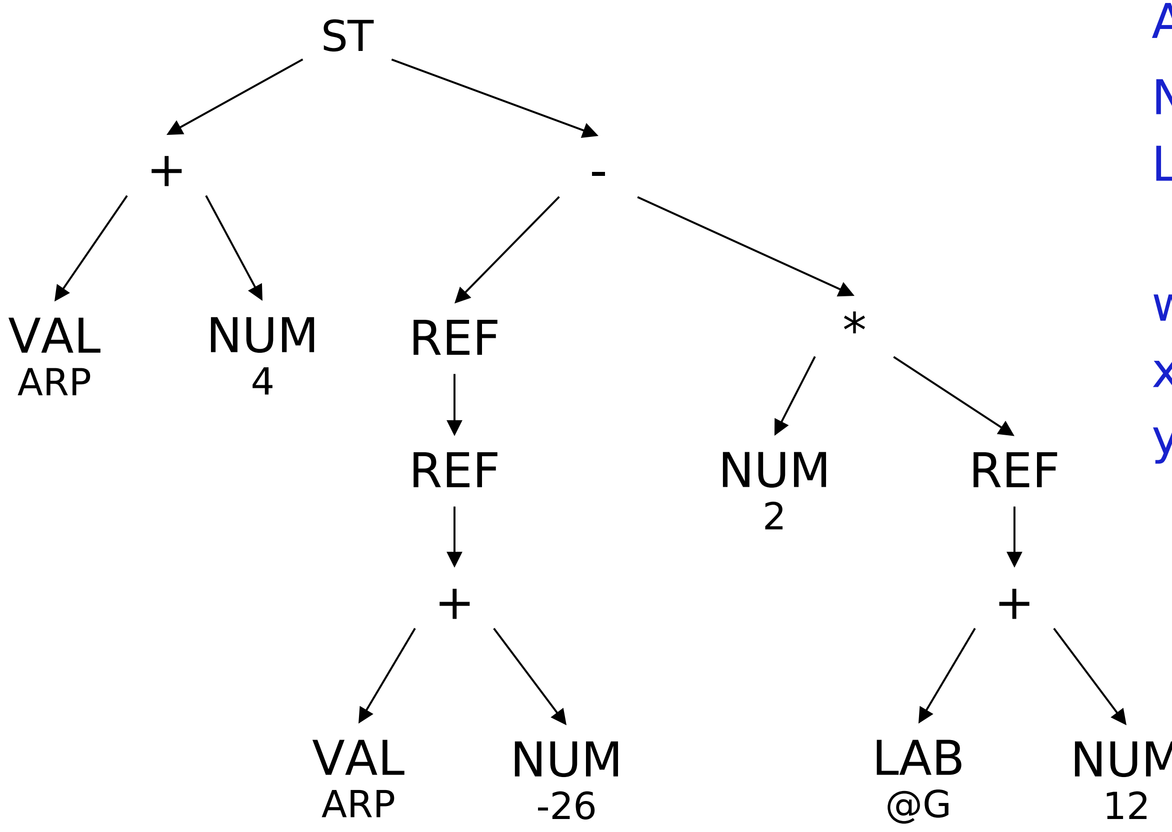
x: at ARP-26

y: at @G+12

(Ref ≈ Load)

The Concept

Low-level AST for w ← (*x) - 2 * y



ARP: r_{arp}

NUM: constant

LAB: ASM label

w: at ARP+4

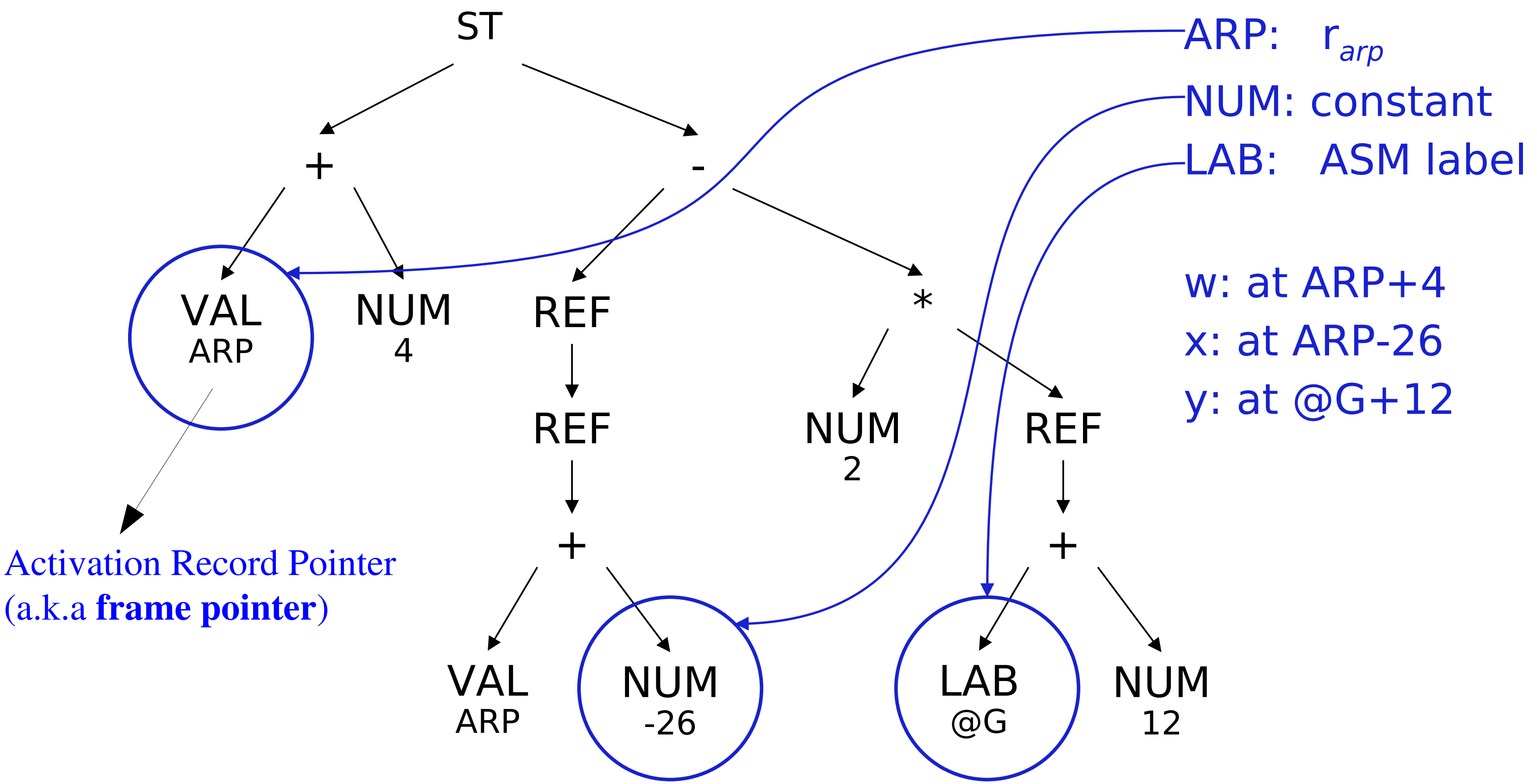
x: at ARP-26

y: at @G+12

(Ref ≈ Load)

The Concept

Low-level AST for w ← (*x) - 2 * y



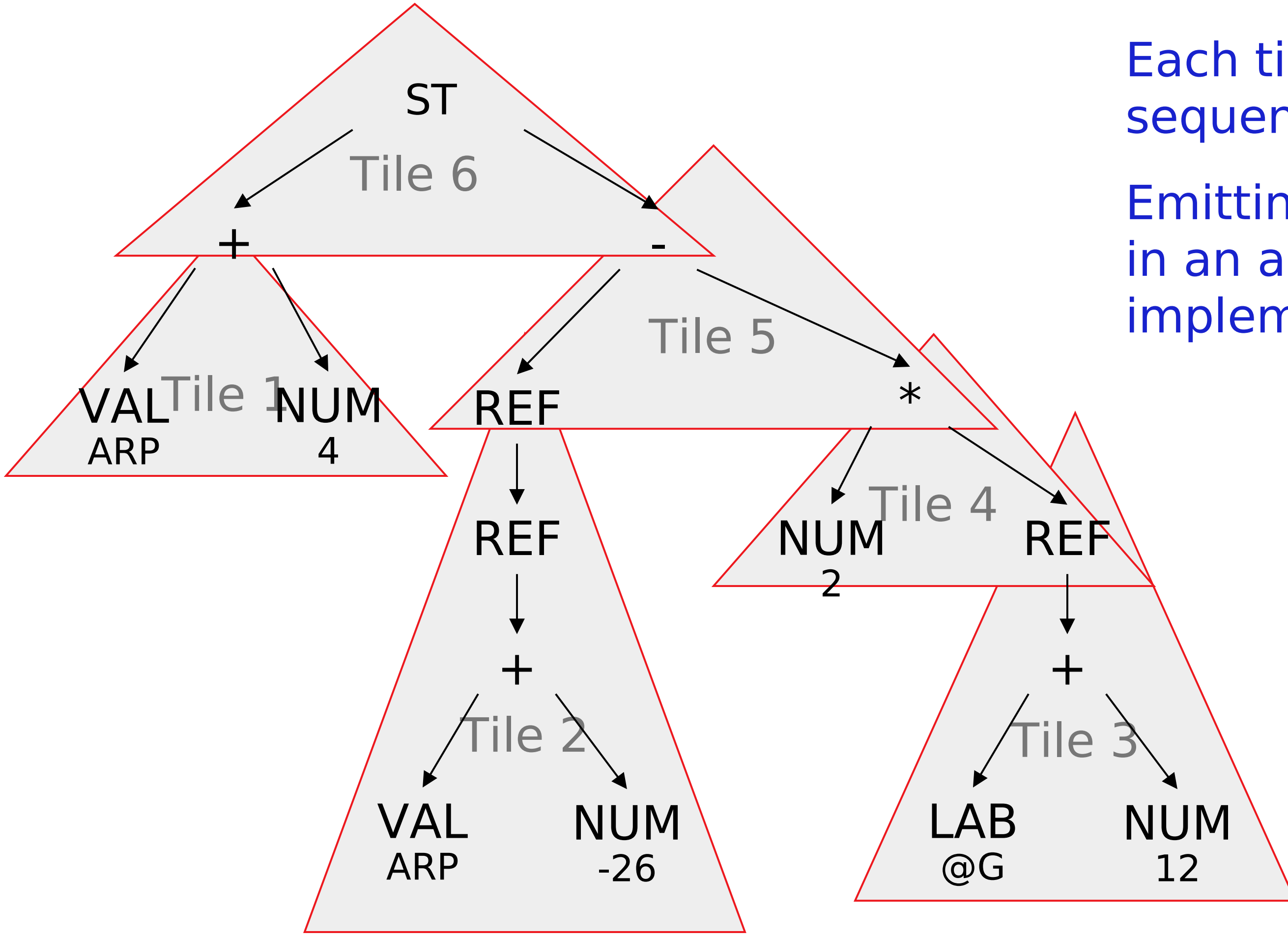
Tree-pattern matching

Goal is to "tile" AST with operation trees

- A tiling is collection of <*ast*,*op* > pairs
  - → *ast* is a node in the AST
  - → *op* is an operation tree
  - → <*ast*, *op* > means that *op* could implement the subtree at *ast*

- A tiling 'implements" an AST if it covers every node in the AST and the overlap between any two trees is limited to a single node
  - → <*ast*, *op*> ∈ tiling means *ast* is also covered by a leaf in another operation tree in the tiling, unless it is the root
  - → Where two operation trees meet, they must be compatible (expect the value in the same location)

Tiling the Tree



Each tile corresponds to a sequence of operations

Emitting those operations in an appropriate order implements the tree.

Generating Code

Given a tiled tree

- Postorder treewalk, with node-dependent order for children
  - → Right child of $\leftarrow$ before its left child
  - → Might impose "most demanding first" rule …
- Emit code sequence for tiles, in order
- Tie boundaries together with register names
  - → Tile 6 uses registers produced by tiles 1 & 5
  - → Tile 6 emits "`store` $r_{\texttt{tile 5}} \Rightarrow r_{\texttt{tile 1}}$"
  - → Can incorporate a "real" register allocator or just use virtual registers

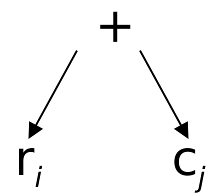
So, What's Hard About This?

Finding the matches to tile the tree

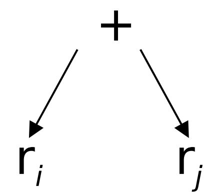
- Compiler writer connects operation trees to AST subtrees
  - → Encode tree syntax, in linear form
  - → Provides a set of rewrite rules
  - → Associated with each is a code template

Notation

To describe these trees, we need a concise notation



$+(r_i,c_j)$

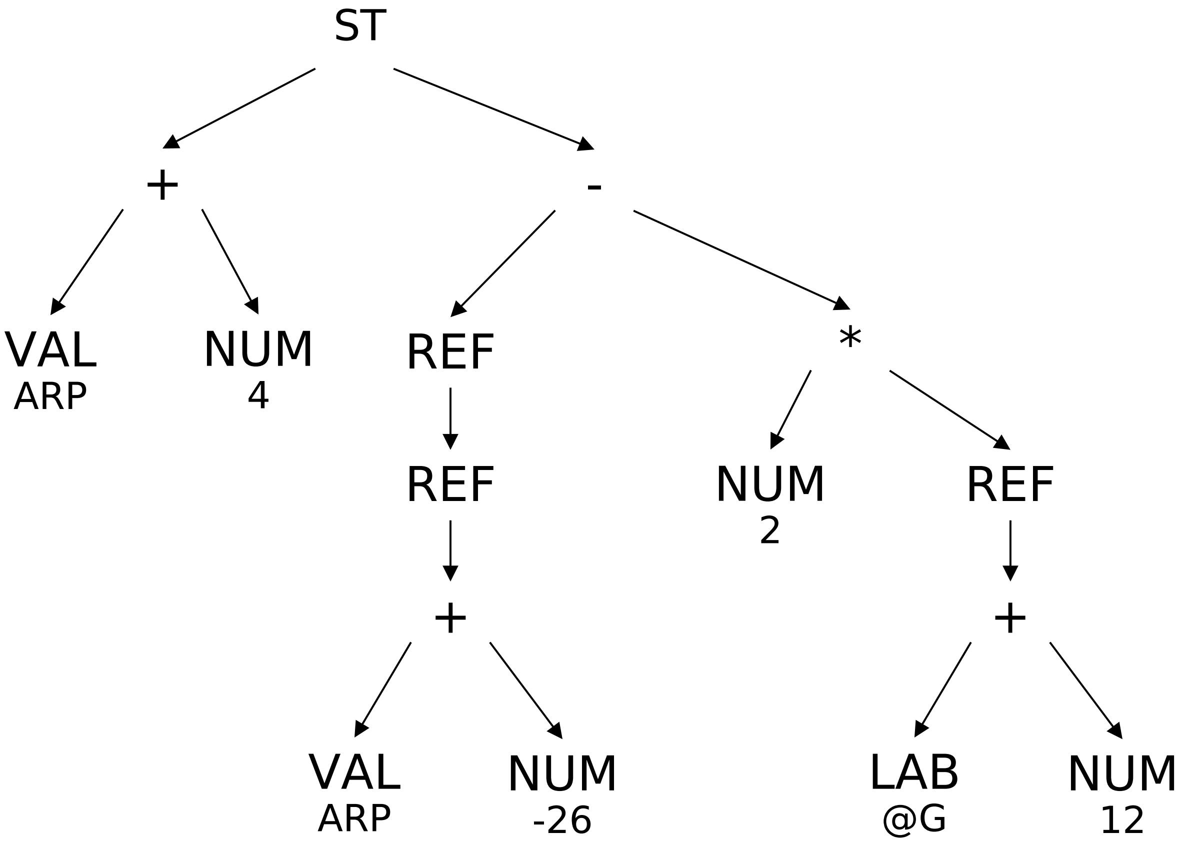


$+(r_i,r_j)$

Linear prefix form

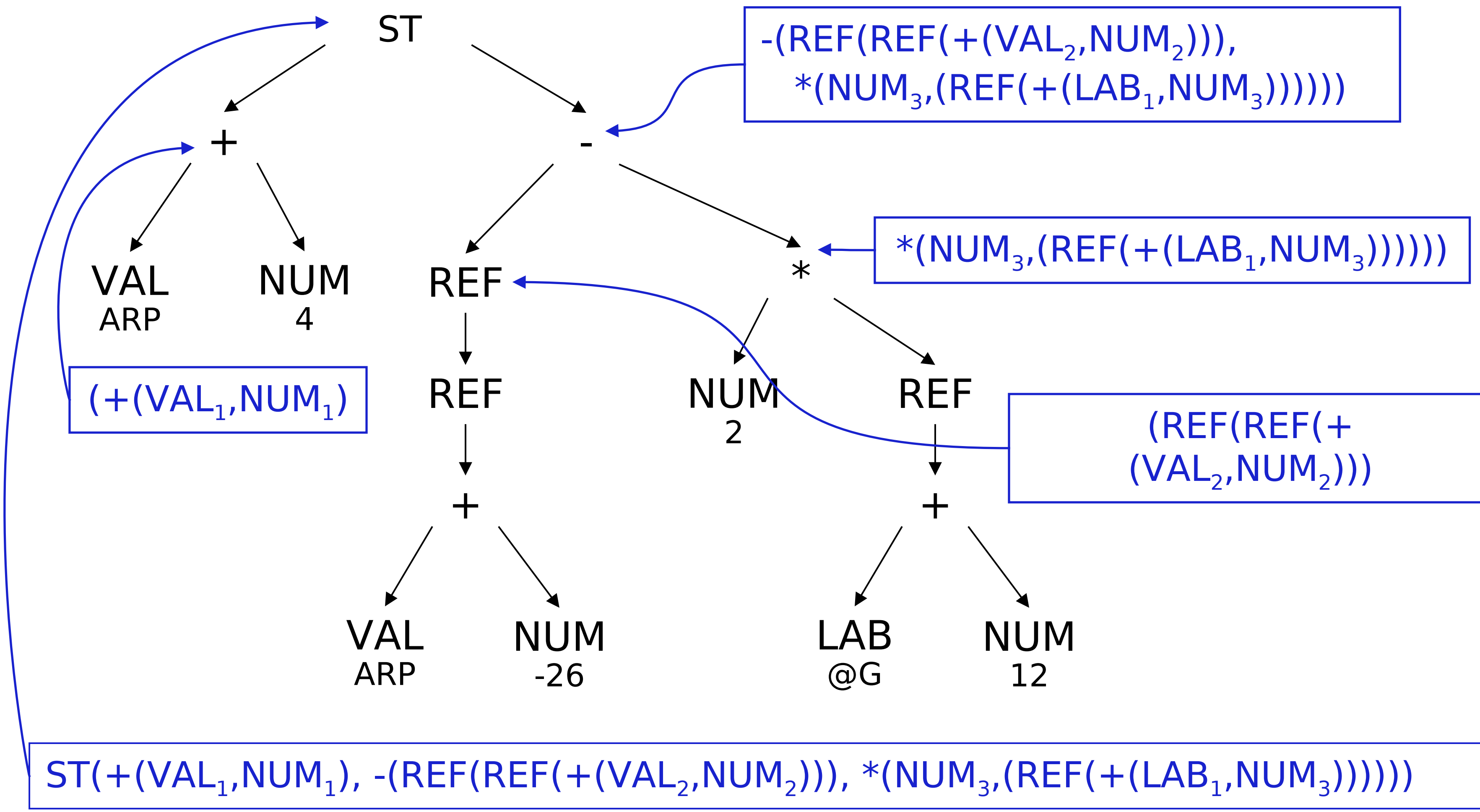
Notation

To describe these trees, we need a concise notation



Notation

To describe these trees, we need a concise notation



Rewrite rules: LL Integer AST into ILOC

	Rule	Cost	Template
1	Goal → Assign	0	
2	Assign → ST(Reg_1,Reg_2)	1	`store` $r_2 \Rightarrow r_1$
3	Assign → ST(+(Reg_1,Reg_2),Reg_3)	1	`storeAO` $r_3 \Rightarrow r_1, r_2$
4	Assign → ST(+(Reg_1,NUM_2),Reg_3)	1	`storeAI` $r_3 \Rightarrow r_1, n_2$
5	Assign → ST(+(NUM_1,Reg_2),Reg_3)	1	`storeAI` $r_3 \Rightarrow r_2, n_1$
6	Reg → LAB_1	1	`loadI` $l_1 \Rightarrow r_{new}$
7	Reg → VAL_1	0	
8	Reg → NUM_1	1	`loadI` $n_1 \Rightarrow r_{new}$
9	Reg → REF(Reg_1)	1	`load` $r_1 \Rightarrow r_{new}$
10	Reg → REF(+ (Reg_1,Reg_2))	1	`loadAO` $r_1, r_2 \Rightarrow r_{new}$
11	Reg → REF(+ (Reg_1,NUM_2))	1	`loadAI` $r_1, n_2 \Rightarrow r_{new}$
12	Reg → REF(+ (NUM_1,Reg_2))	1	`loadAI` $r_2, n_1 \Rightarrow r_{new}$

Rewrite rules: LL Integer AST into ILOC (*part II*)

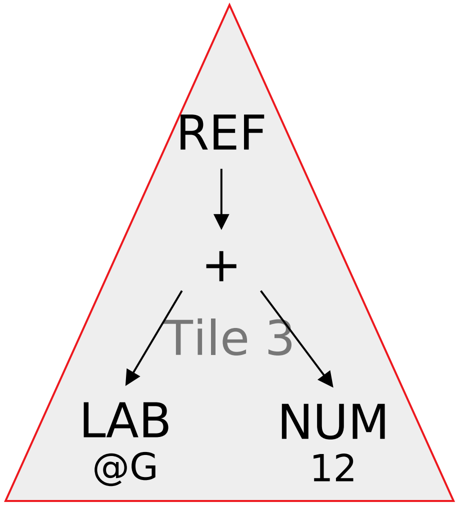
	Rule	Cost	Template
13	Reg → REF(+ (Reg_1,Lab_2))	1	`loadAI` $r_1, l_2 \Rightarrow r_{new}$
14	Reg → REF(+ (Lab_1,Reg_2))	1	`loadAI` $r_2, l_1 \Rightarrow r_{new}$
15	Reg → + (Reg_1,Reg_2)	1	`addI` $r_1, r_2 \Rightarrow r_{new}$
16	Reg → + (Reg_1,NUM_2)	1	`addI` $r_1, n_2 \Rightarrow r_{new}$
17	Reg → + (NUM_1,Reg_2)	1	`addI` $r_2, n_1 \Rightarrow r_{new}$
18	Reg → + (Reg_1,Lab_2)	1	`addI` $r_1, l_2 \Rightarrow r_{new}$
19	Reg → + (Lab_1,Reg_2)	1	`addI` $r_2, l_1 \Rightarrow r_{new}$
20	Reg → - (NUM_1,Reg_2)	1	`rsubI` $r_2, n_1 \Rightarrow r_{new}$
…	…	…	…

A real set of rules would cover more than signed integers …

So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

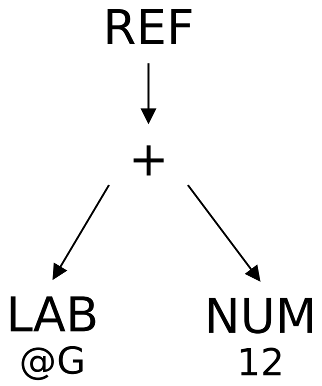
Consider tile 3 in our example



So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

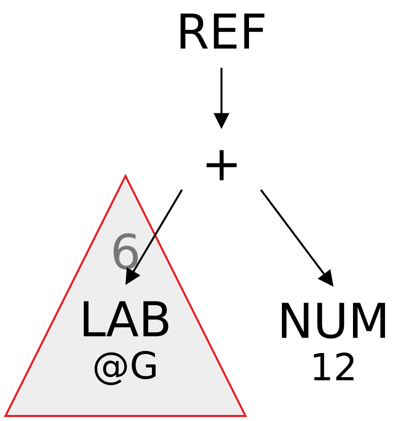


What rules match tile 3?

So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example



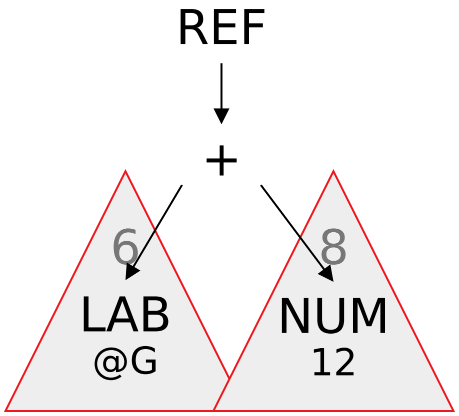
What rules match tile 3?

6: Reg → LAB_1 tiles the lower left node

So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example



What rules match tile 3?

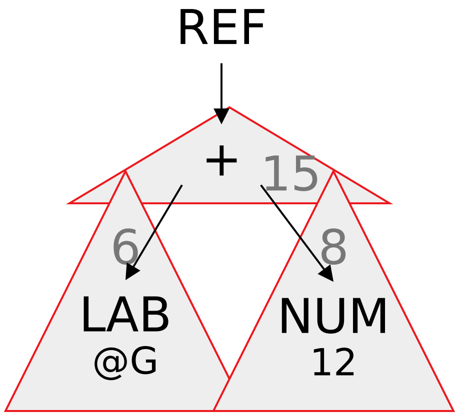
6: Reg $\rightarrow$ LAB_1 tiles the lower left node

8: Reg $\rightarrow$ NUM_1 tiles the bottom right node

So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example



What rules match tile 3?

6: Reg $\rightarrow$ LAB_1 tiles the lower left node

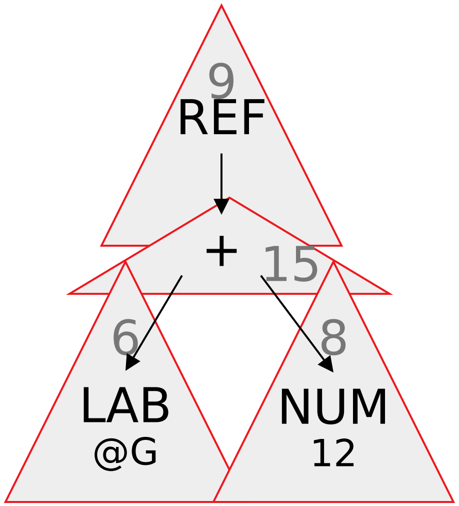
8: Reg $\rightarrow$ NUM_1 tiles the bottom right node

15: Reg $\rightarrow$ + (Reg_1,Reg_2) tiles the + node

So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example



What rules match tile 3?

6: Reg $\rightarrow$ LAB_1 tiles the lower left node

8: Reg $\rightarrow$ NUM_1 tiles the bottom right node

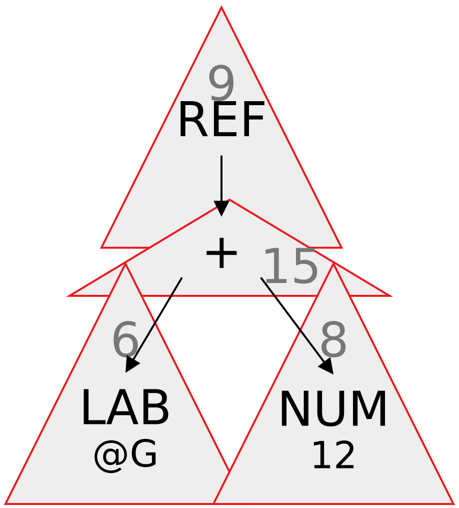
15: Reg $\rightarrow$ + (Reg_1,Reg_2) tiles the + node

9: Reg $\rightarrow$ REF(Reg_1) tiles the REF

So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example



What rules match tile 3?

- 6: Reg → LAB_1 tiles the lower left node
- 8: Reg → NUM_1 tiles the bottom right node
- 15: Reg → + (Reg_1,Reg_2) tiles the + node
- 9: Reg → REF(Reg_1) tiles the REF

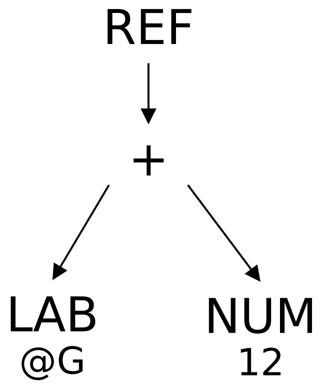
We denote this match as <6,8,15,9>

Of course, it implies <8,6,15,9>

Both have a cost of 4

Finding matches

Many Sequences Match Our Subtree



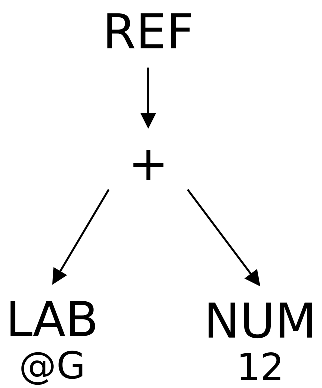
Cost	Sequences			
2	6,11	8,14		
3	6,8,10	8,6,10	6,16,9	8,19,9
4	6,8,15,9	8,6,15,9		

In general, we want the low cost sequence

- Each unit of cost is an operation (1 cycle)
- We should favour short sequences

Finding matches

Low Cost Matches



Sequences with Cost of 2	
6: Reg → LAB_1 11: Reg → REF(+(Reg_1,NUM_2))	`loadI  @G    ⇒ ri` `loadAI ri,12 ⇒ rj`
8: Reg → NUM_1 14: Reg → REF(+(LAB_1,Reg_2))	`loadI 12     ⇒ ri` `loadAI ri,@G ⇒ rj`

These two are equivalent in cost

6,11 might be better, because @G may be longer than the immediate field

Tiling the Tree

Still need an algorithm

- Assume each rule implements one operator
- Assume operator takes 0, 1, or 2 operands

Now, …

Tiling the Tree

```
Tile(n)
  Label(n) ← Ø
  if n has two children then
     Tile (left child of n)
     Tile (right child of n)
     for each rule r that implements n
        if (left(r) ∈ Label(left(n)) and
           (right(r) ∈ Label(right(n))
          then Label(n) ← Label(n) ∪ { r }
```

Match binary nodes against binary rules

```
 else if n has one child
     Tile(child of n)
     for each rule r that implements n
        if (left(r) ∈ Label(child(n))
           then Label(n) ← Label(n) ∪ { r }
```

Match unary nodes against unary rules

```
 else  /* n is a leaf */
     Label(n) ← {all  rules that implement n }
```

Handle leaves with lookup in rule table

Notes:

- left and right refer to the children of the AST node or right-hand sides of a rule
- implements: e.g. rule 9 implements REF

Tiling the Tree

```
Tile(n)
   Label(n) ← Ø
   if n has two children then
       Tile (left child of n)
       Tile (right child of n)
       for each rule r that implements n
           if (left(r) ∈ Label(left(n)) and
              (right(r) ∈ Label(right(n))
             then Label(n) ← Label(n) ∪ { r }
  else if n has one child
       Tile(child of n)
       for each rule r that implements n
           if (left(r) ∈ Label(child(n))
              then Label(n) ← Label(n) ∪ { r }
  else  /* n is a leaf */
       Label(n) ← {all  rules that implement n }
```

This algorithm

- Finds all matches in rule set
- Labels node n with that set
- Can keep lowest cost match at each point
- Leads to a notion of local optimality — lowest cost at each point
- Spends its time in the two matching loops

The Big Picture

- Tree patterns represent AST and ASM
- Can use matching algorithms to find low-cost tiling of AST
- Can turn a tiling into code using templates for matched rules
- Techniques (& tools) exist to do this efficiently

Hand-coded matcher like *Tile*	Avoids large sparse table Lots of work
Encode matching as an automaton	O(1) cost per node Tools like BURS (bottom-up rewriting system), BURG
Use parsing techniques	Uses known technology Very ambiguous grammars
Linearize tree into string and use string searching algorithm (Aho-Corasick)	Finds all matches