Instruction Selection, II
Tree-pattern matching

COMP 412
Fall 2005

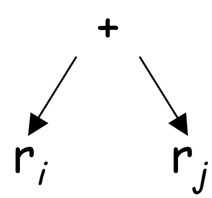
The Concept

Many compilers use tree-structured IRs

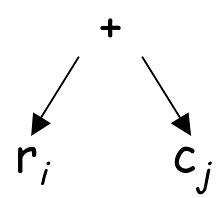
- Abstract syntax trees generated in the parser
- Trees or DAGs for expressions

These systems might well use trees to represent target ISA

Consider the ILOC add operations



`add` $r_i, r_j \Rightarrow r_k$



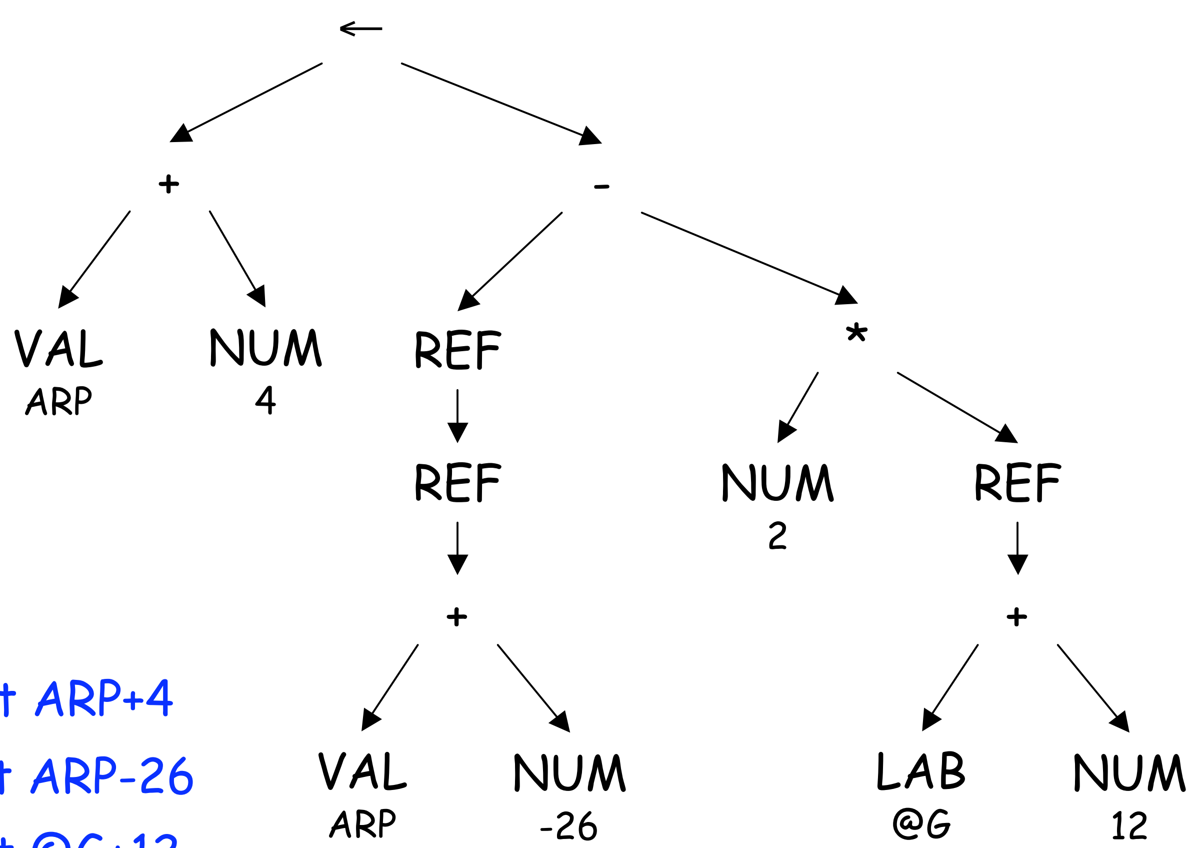
`addI` $r_i, c_j \Rightarrow r_k$

Operation trees or pattern trees

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

The Concept

Low-level AST for w ← x - 2 * y



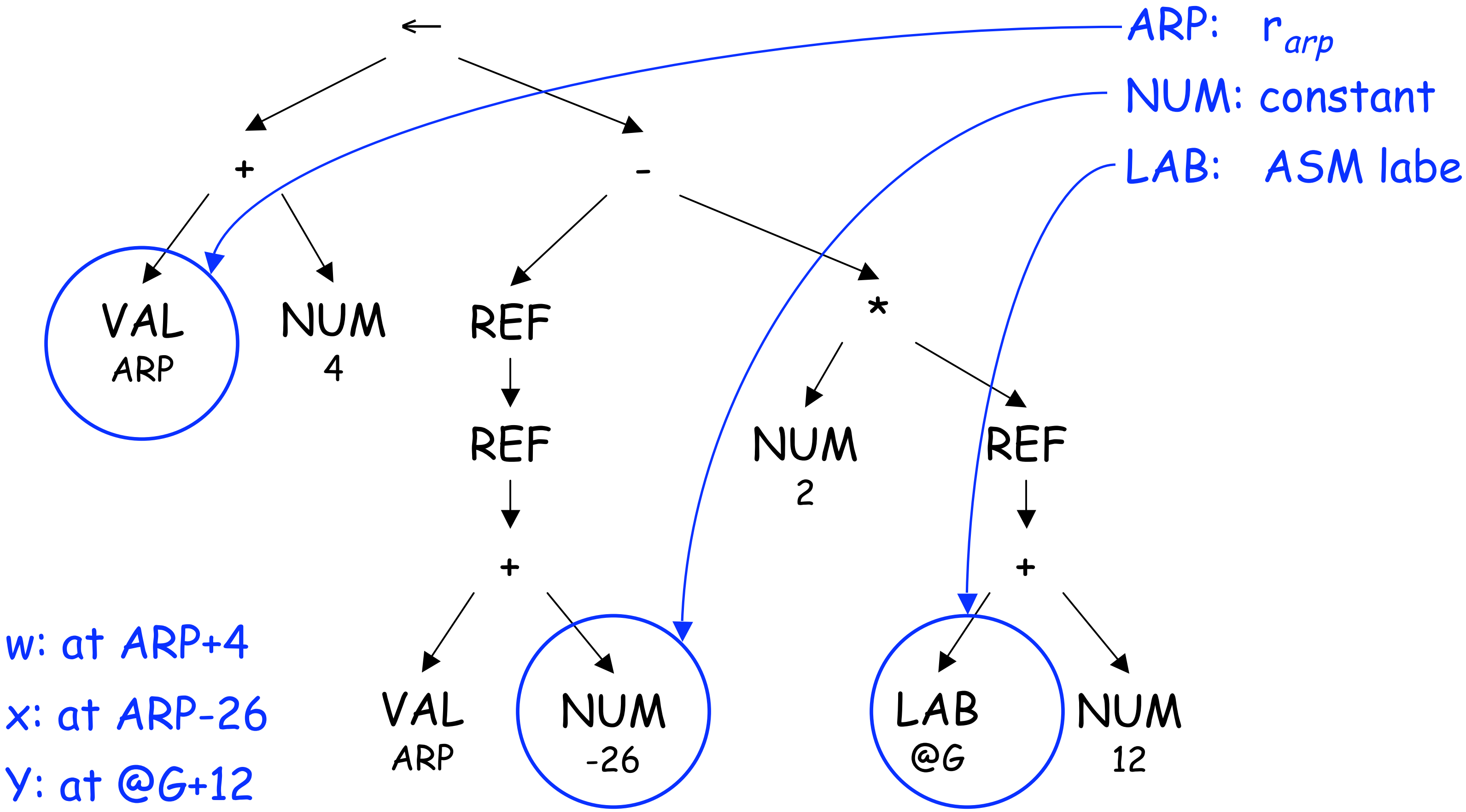
w: at ARP+4

x: at ARP-26

y: at @G+12

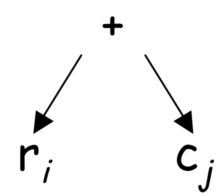
The Concept

Low-level AST for w ← x - 2 * y

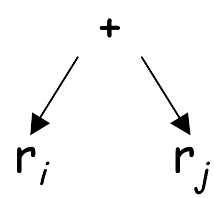


Notation

To describe these trees, we need a concise notation



$+(r_i,c_j)$

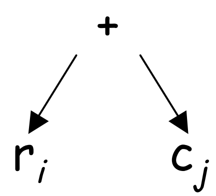


$+(r_i,r_j)$

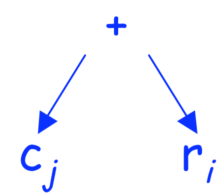
Linear prefix form

Notation

To describe these trees, we need a concise notation

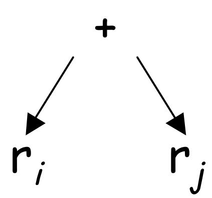


$+(r_i,c_j)$



$+(c_j,r_i)$

Pattern for commutative variant of $+(r_i,c_j)$



$+(r_i,r_j)$

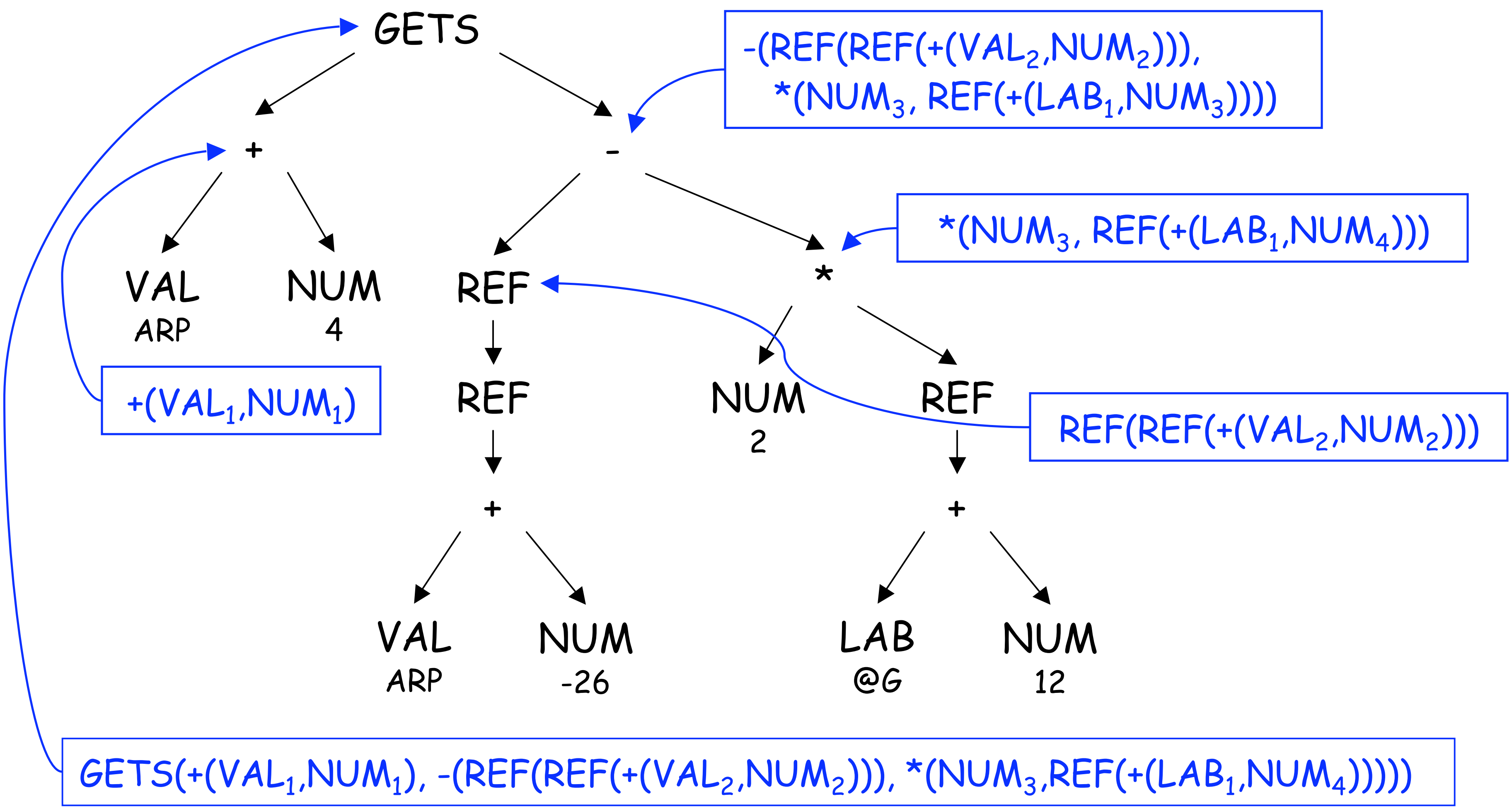
With each tree pattern, we associate a code template and a cost

- Template shows how to implement the subtree
- Cost is used to drive process to low-cost code sequences

Subscripts added to create unique names

Notation

The same notation can describe our low-level AST



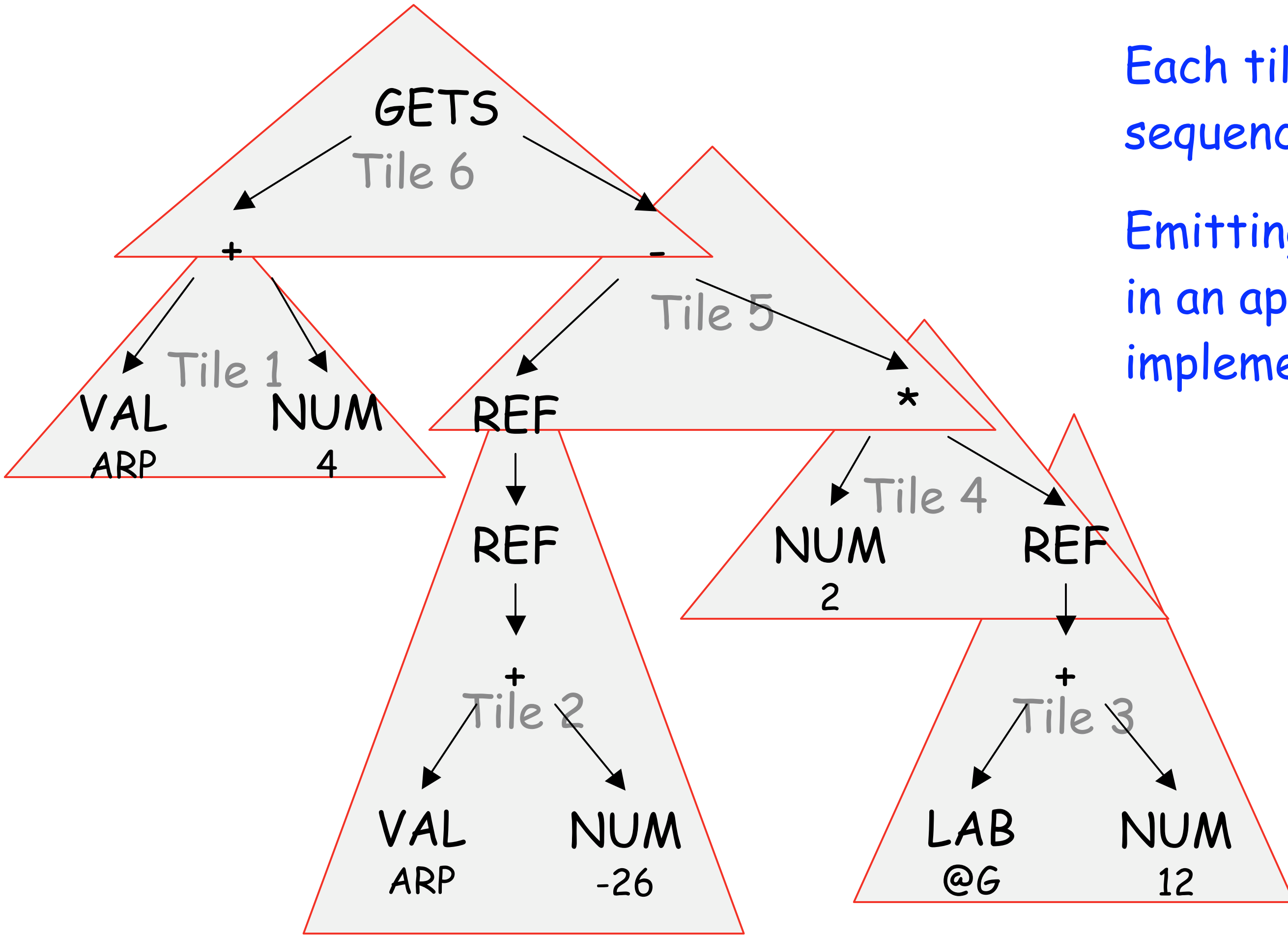
Tree-pattern matching

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 a pattern 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 that the *root* of *ast* is also covered by a leaf in another pattern tree in the tiling, unless it is the root
  - Where two pattern trees overlap, they must be compatible (expect the value in the same location)

A Tiling for our Example Tree

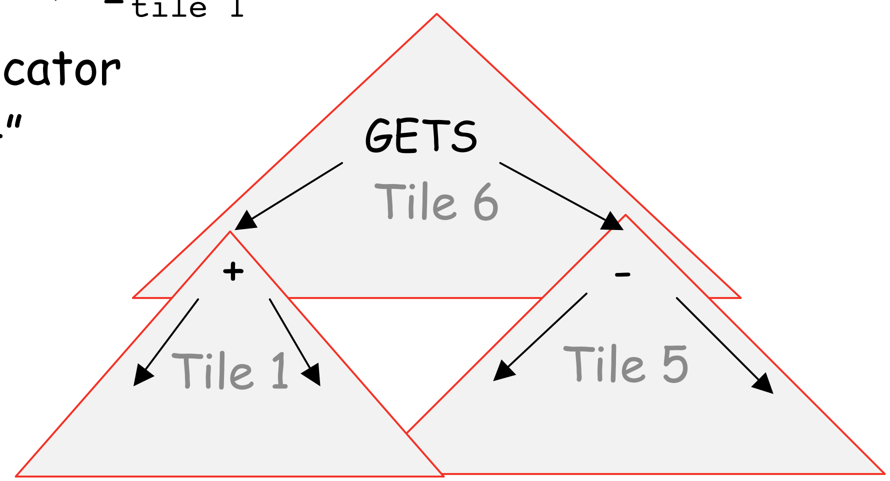


Each tile corresponds to a sequence of operations

Emitting those operations in an appropriate order implements the tree.`

Generating Code from the Tiled Tree

- Postorder treewalk, with node-dependent visitation order
  - Right child of GETS before its left child
  - Might impose "most demanding subtree first" rule … (*Sethi*)
- Emit code sequence for tiles, in traversal order
- Tie boundaries together with names
  - Tile 6 uses registers produced by tiles 1 & 5
  - Tile 6 emits "`store` $\texttt{r}_{\texttt{tile 5}} \Rightarrow \texttt{r}_{\texttt{tile 1}}$"
  - Can incorporate a "real" allocator
    or can use "NextRegister++"



So, What's Hard About This?

Finding the matches to tile the tree

- Compiler writer connects operation trees to AST subtrees
  - Provides a set of rewrite rules
  - Encode tree syntax, in linear form
  - Associate a code template with each rule
  - Give the cost of each template

Rewrite rules: LL Integer AST into ILOC

	Rule	Cost	Template
1	Goal $\rightarrow$ Assign	0	
2	Assign $\rightarrow$ GETS(Reg$_1$,Reg$_2$)	1	store $r_2 \Rightarrow r_1$
3	Assign $\rightarrow$ GETS(+(Reg$_1$,Reg$_2$),Reg$_3$)	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign $\rightarrow$ GETS(+(Reg$_1$,NUM$_2$),Reg$_3$)	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign $\rightarrow$ GETS(+(NUM$_1$,Reg$_2$),Reg$_3$)	1	storeAI $r_3 \Rightarrow r_2, n_1$
6	Reg $\rightarrow$ LAB$_1$	1	loadI $l_1 \Rightarrow r_{new}$
7	Reg $\rightarrow$ VAL$_1$	0	
8	Reg $\rightarrow$ NUM$_1$	1	loadI $n_1 \Rightarrow r_{new}$
9	Reg $\rightarrow$ REF(Reg$_1$)	1	load $r_1 \Rightarrow r_{new}$
10	Reg $\rightarrow$ REF(+ (Reg$_1$,Reg$_2$))	1	loadAO $r_1, r_2 \Rightarrow r_{new}$
11	Reg $\rightarrow$ REF(+ (Reg$_1$,NUM$_2$))	1	loadAI $r_1, n_2 \Rightarrow r_{new}$
12	Reg $\rightarrow$ REF(+ (NUM$_1$,Reg$_2$))	1	loadAI $r_2, n_1 \Rightarrow r_{new}$

From Figure 11.5 in EaC

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

Commutative pair

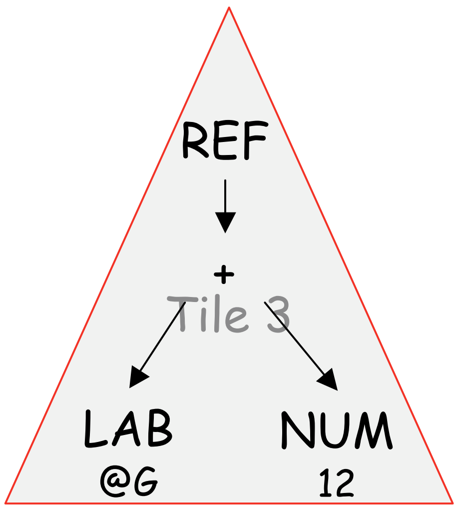
	Rule	Cost	Template
13	Reg → + (Reg_1,Reg_2)	1	`add` $r_1, r_2 \Rightarrow r_{new}$
14	Reg → + (Reg_1,NUM_2)	1	`addI` $r_1, n_2 \Rightarrow r_{new}$
15	Reg → + (NUM_1,Reg_2)	1	`addI` $r_2, n_1 \Rightarrow r_{new}$
16	Reg → - (Reg_1,Reg_2)	1	`sub` $r_1, r_2 \Rightarrow r_{new}$
17	Reg → - (Reg_1,NUM_2)	1	`subI` $r_1, n_2 \Rightarrow r_{new}$
18	Reg → - (NUM_1,Reg_2)	1	`rsubI` $r_2, n_1 \Rightarrow r_{new}$
19	Reg → × (Reg_1,Reg_2)	1	`mult` $r_1, r_2 \Rightarrow r_{new}$
20	Reg → × (Reg_1,NUM_2)	1	`multI` $r_1, n_2 \Rightarrow r_{new}$
21	Reg → × (NUM_1,Reg_2)	1	`multI` $r_2, n_1 \Rightarrow r_{new}$

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

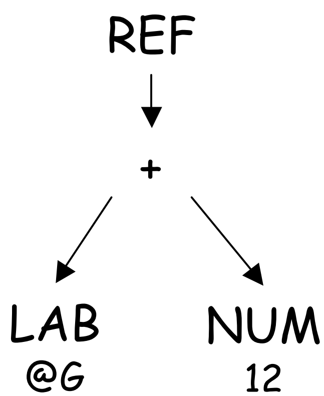


So, What's Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

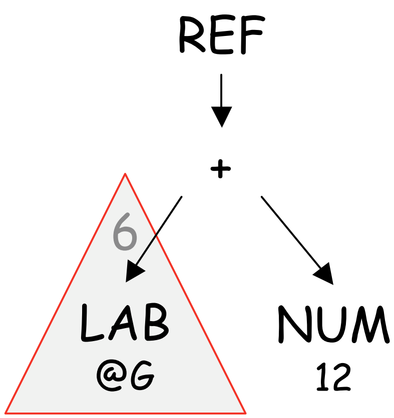
What rules match tile 3?



So, What's Hard About This?

Need an algorithm to match AST subtrees with the rules

Consider tile 3 in our example



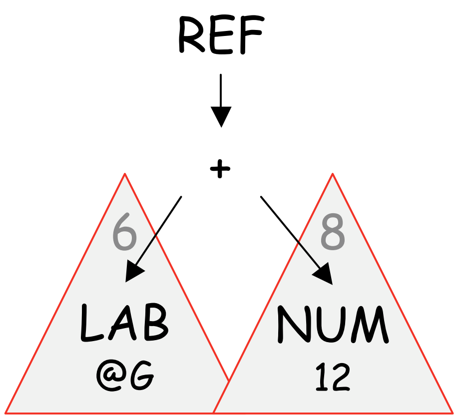
What rules match tile 3?

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?

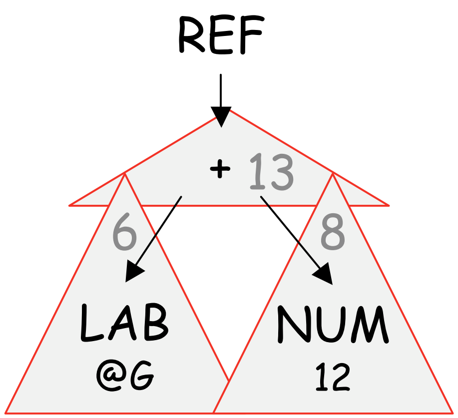
6: Reg → LAB_1 tiles the lower left node

8: Reg → 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

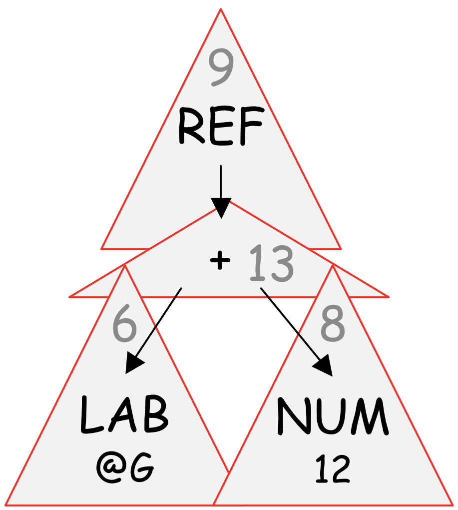
8: Reg $\rightarrow$ NUM_1 tiles the bottom right node

13: 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



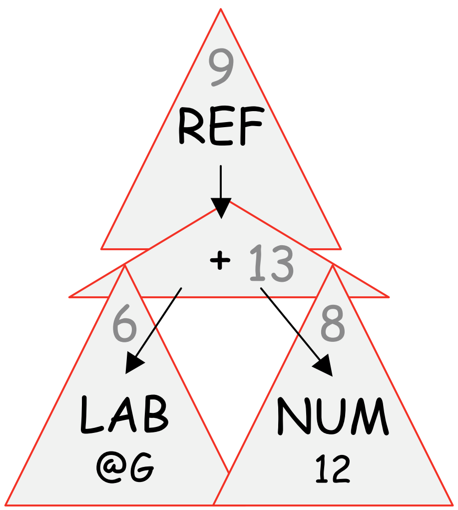
What rules match tile 3?

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

13: Reg → + (Reg_1,Reg_2) tiles the + node

9: Reg → REF(Reg_1) tiles the REF

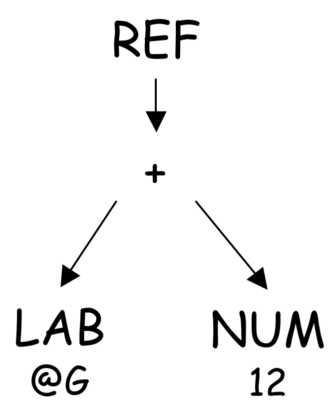
We denote this match as <6,8,13,9>

Of course, it implies <8,6,13,9>

Both have a cost of 4

Finding matches

Many Sequences Match Our Subtree



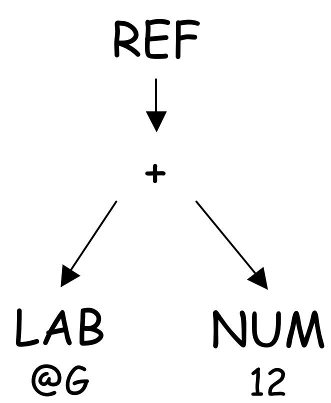
Cost	Sequences			
2	6,11	8,12		
3	6,8,10	8,6,10	6,14,9	8,15,9
4	6,8,13,9	8,6,13,9		

In general, we want the low cost sequence

- We have assumed uniform unit costs
- We favor shorter (lower-cost) sequences
- Accurate costs produce better code

Finding matches

Low Cost Matches



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

These two sequences are equivalent in cost

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

Can encode length restriction on immediate fields into the classification of terminals, at the cost of a little more ambiguity

Tiling the Tree

Still need an algorithm

- Assume each rule implements one operator
  - This is a big assumption because it eliminates rules like 11

    $Reg \rightarrow REF(+\ (Reg_1, NUM_2))$ (corresponds to loadAI)
  - A solution: break into two rules

    $Reg \rightarrow REF(DUM)$ cost: 1

    $DUM \rightarrow +\ (Reg_1, NUM_2)$ cost: 0
- 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 }
  else if n has one child
      Tile(left 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 }
```

Match binary nodes against binary rules

Match unary nodes against unary rules

Handle leaves with lookup in rule table

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(left 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 }
```

Use same notation to navigate through the rules (pattern trees) and the AST

- *left* and *right* have obvious meanings on AST
- *left* and *right* have similar meanings on the rhs of a rule — interpreted as if on the underlying pattern tree

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(left 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

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(left 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 }
```

Oversimplifications

1. Only handles 1 storage class
2. Must track low cost sequence in each class
3. Must choose lowest cost for subtree, across all classes

The extensions to handle these complications are pretty straightforward.

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(left 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 }
```

Can turn matching code (inner loop) into a table lookup

Table can get huge and sparse

|op trees| x |labels| x |labels|
200 x 1000 x 1000
leads to 200,000,000 entries

Fortunately, they are quite sparse & have reasonable encodings (e.g., Chase's work)

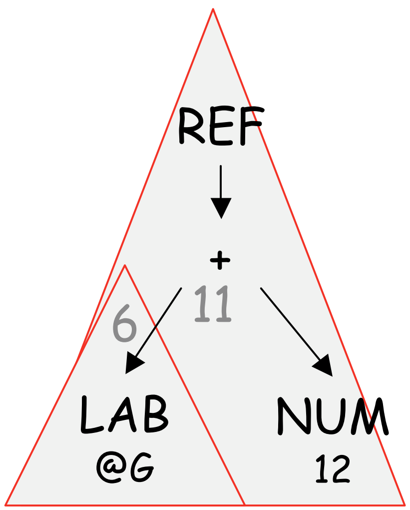
The Big Picture

- 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, BURG
Use parsing techniques	Uses known technology Very ambiguous grammars
Linearize tree into string and use Aho-Corasick	Finds all matches

Extra Slides Start Here

Other Sequences for Tile 3



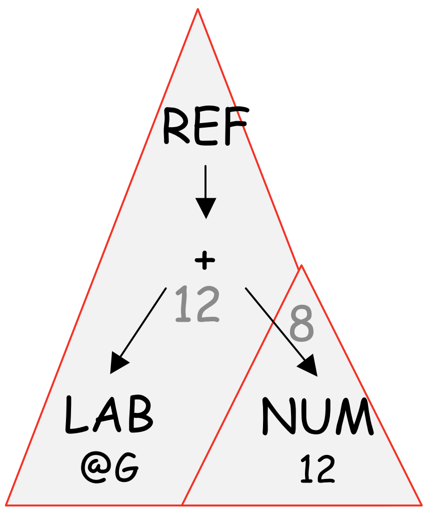
6,11

6: Reg → LAB_1

11: Reg → REF(+ (Reg_1,NUM_2))

Two operator rule

Other Sequences for Tile 3



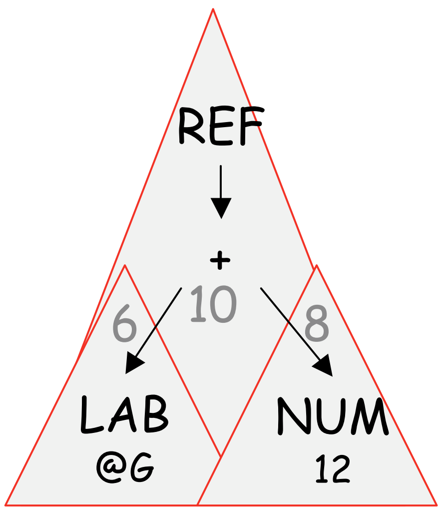
8,12

8: $Reg \rightarrow NUM_1$

12: $Reg \rightarrow REF(+ (NUM_1,Reg_2))$

Two operator rule

Other Sequences for Tile 3



6,8,10

6: Reg → LAB_1

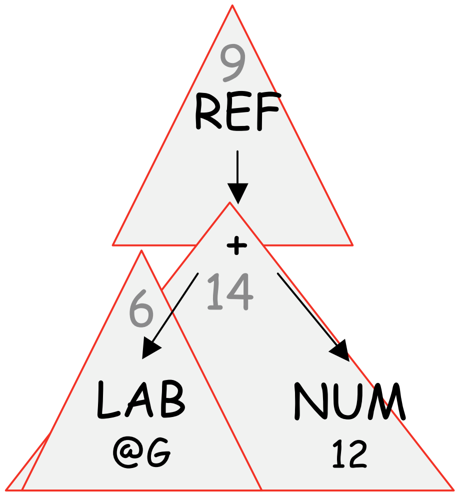
8: Reg → NUM_1

11: Reg → REF(+ (Reg_1,Reg_2))

Two operator rule

8,6,10 looks the same

Other Sequences for Tile 3



6,14,9

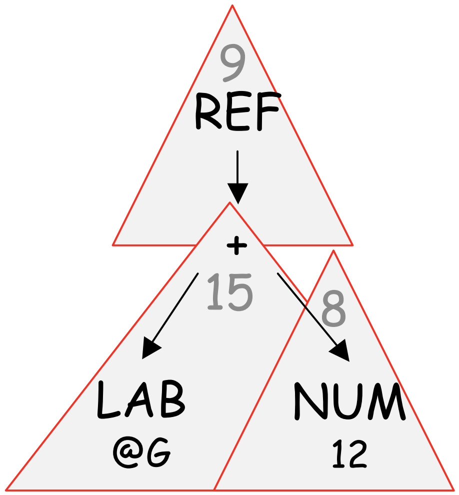
6: Reg $\rightarrow$ LAB_1

14: Reg $\rightarrow$ + (Reg_1,NUM_2)

9: Reg $\rightarrow$ REF(Reg_1)

All single operator rules

Other Sequences for Tile 3



8,15,9

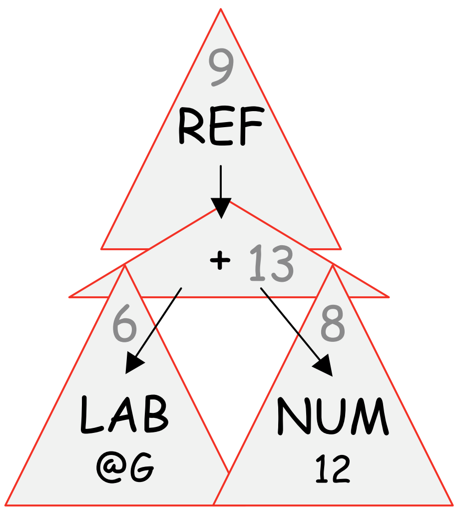
8: $Reg \rightarrow NUM_1$

15: $Reg \rightarrow + (NUM_1, Reg_2)$

9: $Reg \rightarrow REF(Reg_1)$

All single operator rules

Other Sequences for Tile 3



6,8,13,9

6: $Reg \rightarrow LAB_1$

8: $Reg \rightarrow NUM_1$

13: $Reg \rightarrow + (Reg_1,Reg_2)$

9: $Reg \rightarrow REF(Reg_1)$

All single operator rules

8,6,13,9 looks the same