Instruction Selection via Tree-Pattern Matching

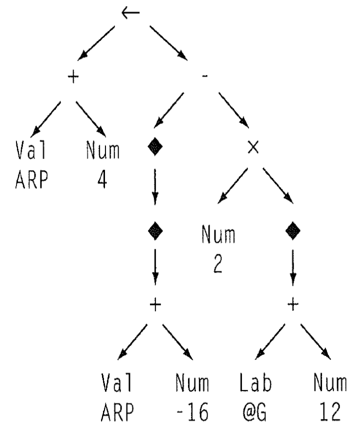
Objective

Annotate AST so as to produce close to optimal code.

Produce potential sequences of code and let the compiler schedule them at a later time.

This is an alternative to first generating code and then optimizing it through peep-hole optimization.

Annotated AST



a = b – 2 x c

A diamond means an address
Traverse tree and apply re-write rules

	Production			Cost	Code Template
1	Goal	→	Assign	0	
2	Assign	→	← (Reg_1,Reg_2)	1	store r_2 ⇒ r_1
3	Assign	→	← (+ (Reg_1,Reg_2),Reg_3)	1	storeAO r_3 ⇒ r_1,r_2
4	Assign	→	← (+ (Reg_1,Num_2),Reg_3)	1	storeAI r_3 ⇒ r_1,n_2
5	Assign	→	← (+ (Num_1,Reg_2),Reg_3)	1	storeAI r_3 ⇒ r_2,n_1
6	Reg	→	Lab_1	1	loadI l_1 ⇒ r_{new}
7	Reg	→	Val_1	0	
8	Reg	→	Num_1	1	loadI n_1 ⇒ r_{new}
9	Reg	→	◆ (Reg_1)	1	load r_1 ⇒ r_{new}
10	Reg	→	◆ (+ (Reg_1,Reg_2))	1	loadAO r_1,r_2 ⇒ r_{new}
11	Reg	→	◆ (+ (Reg_1,Num_2))	1	loadAI r_1,n_2 ⇒ r_{new}
12	Reg	→	◆ (+ (Num_1,Reg_2))	1	loadAI r_2,n_1 ⇒ r_{new}
13	Reg	→	◆ (+ (Reg_1,Lab_2))	1	loadAI r_1,l_2 ⇒ r_{new}
14	Reg	→	◆ (+ (Lab_1,Reg_2))	1	loadAI r_2,l_1 ⇒ r_{new}
15	Reg	→	+ (Reg_1,Reg_2)	1	add r_1,r_2 ⇒ r_{new}
16	Reg	→	+ (Reg_1,Num_2)	1	addI r_1,n_2 ⇒ r_{new}
17	Reg	→	+ (Num_1,Reg_2)	1	addI r_2,n_1 ⇒ r_{new}
18	Reg	→	+ (Reg_1,Lab_2)	1	addI r_1,l_2 ⇒ r_{new}
19	Reg	→	+ (Lab_1,Reg_2)	1	addI r_2,l_1 ⇒ r_{new}
20	Reg	→	- (Reg_1,Reg_2)	1	sub r_1,r_2 ⇒ r_{new}
21	Reg	→	- (Reg_1,Num_2)	1	subI r_1,n_2 ⇒ r_{new}
22	Reg	→	- (Num_1,Reg_2)	1	rsubI r_2,n_1 ⇒ r_{new}
23	Reg	→	× (Reg_1,Reg_2)	1	mult r_1,r_2 ⇒ r_{new}
24	Reg	→	× (Reg_1,Num_2)	1	multI r_1,n_2 ⇒ r_{new}
25	Reg	→	× (Num_1,Reg_2)	1	multI r_2,n_1 ⇒ r_{new}

Example

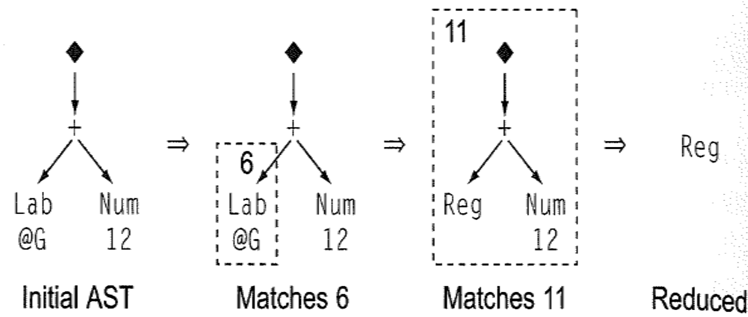
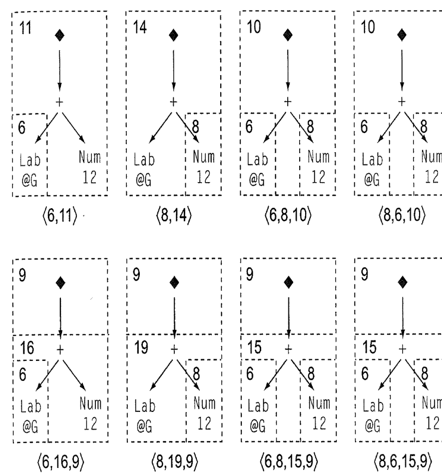


FIGURE 11.5 A Simple Tree Rewrite Sequence.

But Wait! There is more!



The Scream by Edvard Munch

Some Code to Calm you Down

```
loadI  @G      => ri
loadAI ri,12   => rj
```
⟨6,11⟩

```
loadI  12      => ri
loadAI ri,@G   => rj
```
⟨8,14⟩

```
loadI  @G      => ri
loadI  12      => rj
loadAO ri,rj   => rk
```
⟨6,8,10⟩

```
loadI  12      => ri
loadI  @G      => rj
loadAO ri,rj   => rk
```
⟨8,6,10⟩

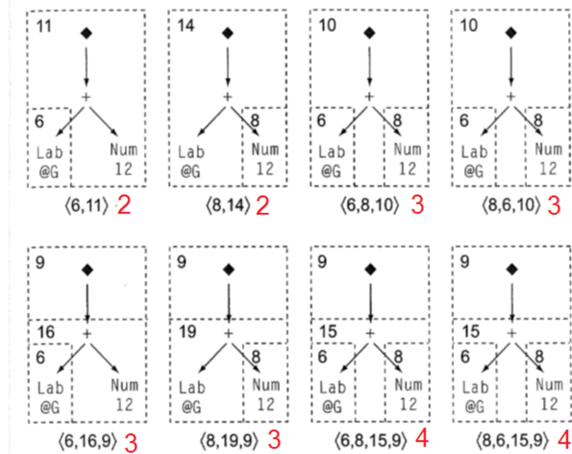
```
loadI @G       => ri
addI  ri,12    => rj
load  rj       => rk
```
⟨6,16,9⟩

```
loadI 12       => ri
addI  ri,@G    => rj
load  rj       => rk
```
⟨8,19,9⟩

```
loadI @G       => ri
loadI 12       => rj
add   ri,rj    => rk
load  rk       => rl
```
⟨6,8,15,9⟩

```
loadI 12       => ri
loadI @G       => rj
add   ri,rj    => rk
load  rk       => rl
```
⟨8,6,15,9⟩

Use Cost Function to Find Low-cost Solution



Simple Tree-walk Algorithm

```
Tile(n)
  Label(n) ← Ø
  if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n)) and right(r) ∈ Label(right(n))
        then Label(n) ← Label(n) ∪ {r}
  else if n is a unary node then
    Tile(left(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n))
        then Label(n) ← Label(n) ∪ {r}
  else /* n is a leaf */
    Label(n) ← {all rules that match the operation in n}
```

Simple Tree-walk Algorithm

The algorithm labels each node with sets of patterns.

The compiler will can choose from them.

The algorithm uses a post-order traversal of the AST.

For assignment statements, generate code for rhs before lhs.