Compiler Construction

Chapter 11: Instruction Selection

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Overview

Back-end optimization

- Instruction Selection
- Instruction scheduling
- Register allocation

Convert IR into target machine codes

- Machine-independent: most languages share some common features
- Machine-dependent: number of target instructions, types of registers, cost of each Instruction

Code generation

- Syntax-directed translation: Naive without any optimization
- Hand-crafted mapping rules between IR and target machine codes
- An automatic tool to choose the best choice among several possibilities

E.g. choices for $r_j = r_i$ in ILOC are `i2i r_j <= r_i` and the followings

addI	r_i,0	⇒	r_j	subI	r_i,0	⇒	r_j	multI	r_i,1	⇒	r_j
divI	r_i,1	⇒	r_j	lshiftI	r_i,0	⇒	r_j	rshiftI	r_i,0	⇒	r_j
and	r_i,r_i	⇒	r_j	orI	r_i,0	⇒	r_j	xorI	r_i,0	⇒	r_j

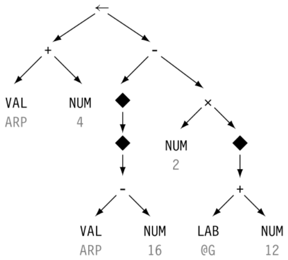
Target machine codes

- Faster runtime (lower instruction cost)
- Less energy consumption
- Shorter codes (smaller size)

Motivating example

Op	Arg_1	Arg_2	Result
×	2	c	t_1
-	b	t_1	a

(a) Quadruples



(b) Low-Level AST

FIGURE 11.1 Low-Level IRs for $a \leftarrow b - 2 \times c$.

The low-level detail in the AST allows the instruction selector to tailor its decisions to specific context.

- Although the high-level AST look similar, the translation will be different.

Low-level abstract syntax tree

Node types

- `NUM` for a constant that fits to the immediate field of a 3-address code
- `CON` for a constant that is larger than `NUM` but fits into the immediate field of `loadI`
- `LAB` represents relocatable symbol, i.e. an assembly-level label
- `VAL` represents a value in a register
- $\diamond$ signifies a level of indirection

Ad-hoc matching

Template-based matching

- For a tree-based IR, the approach is a post-order walk, similar to the syntax-driven translation
- For a linear IR, the approach takes a linear scan over the code and emits codes for each operation

In this chapter

- Peephole optimization as a selector
- Tree-pattern matching as a selector

Peephole optimization

- Peephole: a small window (scope) that is moved over the code
- We are looking for a specific pattern in the translated code

E.g.

storeAI $r_1 \Rightarrow r_{arp},8$ loadAI $r_{arp},8 \Rightarrow r_{15}$	$\Rightarrow$	storeAI $r_1 \Rightarrow r_{arp},8$ i2i $r_1 \Rightarrow r_{15}$
addI $r_2,0 \Rightarrow r_7$ mult $r_4,r_7 \Rightarrow r_{10}$	$\Rightarrow$	mult $r_4,r_2 \Rightarrow r_{10}$
jumpI $\rightarrow l_{10}$ l_{10}: jumpI $\rightarrow l_{11}$	$\Rightarrow$	jumpI $\rightarrow l_{11}$ l_{10}: jumpI $\rightarrow l_{11}$

Components

1. The expander will rewrite the input IR into low-level IR (LLIR)
2. The simplifier with find specific patterns in the LLIR and improve them
3. The matcher will match (map) the LLIR into machine instructions (assembly - ASM)

Example: Peephole optimization

Op	Arg$_1$	Arg$_2$	Result
×	2	c	t_1
-	b	t_1	a

Expand ⇒

$r_1 \leftarrow 2$
$r_2 \leftarrow \texttt{@CP}$
$r_3 \leftarrow \texttt{@G}$
$r_4 \leftarrow r_2 + r_3$
$r_5 \leftarrow M(r_4)$
$r_6 \leftarrow 12$
$r_7 \leftarrow r_5 + r_6$
$r_8 \leftarrow M(r_7)$
$r_9 \leftarrow r_1 \times r_8$
$r_{10} \leftarrow 16$
$r_{11} \leftarrow r_{arp} - r_{10}$
$r_{12} \leftarrow M(r_{11})$
$r_{13} \leftarrow M(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$
$r_{15} \leftarrow 4$
$r_{16} \leftarrow r_{arp} + r_{15}$
$M(r_{16}) \leftarrow r_{14}$

⇓ *Simplify*

$r_1 \leftarrow 2$
$r_5 \leftarrow M(\texttt{@CP} + \texttt{@G})$
$r_8 \leftarrow M(r_5 + 12)$
$r_9 \leftarrow r_1 \times r_8$
$r_{12} \leftarrow M(r_{arp} - 16)$
$r_{13} \leftarrow M(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$
$M(r_{arp} + 4) \leftarrow r_{14}$

Match ⇐

```
loadI   2          => r1
loadI   @CP        => rt1
loadAI  rt1, @G    => r5
loadAI  r5, 12     => r8
mult    r1, r8     => r9
subI    rarp, 16   => rt2
load    rt2        => r12
load    r12        => r13
sub     r13, r9    => r14
storeAI r14        => rarp, 4
```

■ **FIGURE 11.2** *Expand*, *Simplify*, and *Match* Applied to the Example.

Sequences produced by the simplifier

$r_1 \leftarrow 2$ $r_2 \leftarrow @CP$ $r_3 \leftarrow @G$	$r_2 \leftarrow @CP$ $r_3 \leftarrow @G$ $r_4 \leftarrow r_2 + r_3$	$r_4 \leftarrow @CP + @G$ $r_5 \leftarrow M(r_4)$ $r_6 \leftarrow 12$	$r_5 \leftarrow M(@CP + @G)$ $r_6 \leftarrow 12$ $r_7 \leftarrow r_5 + r_6$
Sequence 1	Sequence 2	Sequence 3	Sequence 4
$r_5 \leftarrow M(@CP + @G)$ $r_7 \leftarrow r_5 + 12$ $r_8 \leftarrow M(r_7)$	$r_5 \leftarrow M(@CP + @G)$ $r_8 \leftarrow M(r_5 + 12)$ $r_9 \leftarrow r_1 \times r_8$	$r_8 \leftarrow M(r_5 + 12)$ $r_9 \leftarrow r_1 \times r_8$ $r_{10} \leftarrow 16$	$r_9 \leftarrow r_1 \times r_8$ $r_{10} \leftarrow 16$ $r_{11} \leftarrow r_{arp} - r_{10}$
Sequence 5	Sequence 6	Sequence 7	Sequence 8
$r_9 \leftarrow r_1 \times r_8$ $r_{11} \leftarrow r_{arp} - 16$ $r_{12} \leftarrow M(r_{11})$	$r_9 \leftarrow r_1 \times r_8$ $r_{12} \leftarrow M(r_{arp} - 16)$ $r_{13} \leftarrow M(r_{12})$	$r_{12} \leftarrow M(r_{arp} - 16)$ $r_{13} \leftarrow M(r_{12})$ $r_{14} \leftarrow r_{13} - r_9$	$r_{13} \leftarrow M(r_{12})$ $r_{14} \leftarrow r_{13} - r_9$ $r_{15} \leftarrow 4$
Sequence 9	Sequence 10	Sequence 11	Sequence 12
$r_{14} \leftarrow r_{13} - r_9$ $r_{15} \leftarrow 4$ $r_{16} \leftarrow r_{arp} + r_{15}$	$r_{14} \leftarrow r_{13} - r_9$ $r_{16} \leftarrow r_{arp} + 4$ $M(r_{16}) \leftarrow r_{14}$	$r_{14} \leftarrow r_{13} - r_9$ $M(r_{arp} + 4) \leftarrow r_{14}$	$M(r_{arp} + 4) \leftarrow r_{14}$
Sequence 13	Sequence 14	Sequence 15	Sequence 16

■ **FIGURE 11.3** Sequences Produced by the Simplifier.

Result from the simplifier

1	r_1	$\leftarrow 2$
6	r_5	$\leftarrow M(@CP + @G)$
7	r_8	$\leftarrow M(r_5 + 12)$
10	r_9	$\leftarrow r_1 \times r_8$
11	r_{12}	$\leftarrow M(r_{arp} - 16)$
12	r_{13}	$\leftarrow M(r_{12})$
15	r_{14}	$\leftarrow r_{13} - r_9$
16	$M(r_{arp} + 4)$	$\leftarrow r_{14}$

Code Emitted by the Simplifier

Recognizing dead values

```
r12 <= 2
r14 <= r12 + r12
```

- We may fold `r12 + 12` into 4
- However, we cannot eliminate `r12 = 2` unless we know that the variable will NOT be used after these operations
  - We may perform LiveOut analysis or find list of names that are used in more than one block

Control-flow operations

Eliminating branches and jumps by tracking dead labels

- Combine blocks
- Eliminating unreachable block

Physical versus logical windows

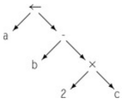
Physical windows: adjacent operations

- Lack of instruction-level parallelism

Logical windows: connected operation in the flow

- E.g. operations that define and use the same value
- Next use analysis

Simple tree walk



Op	Arg_1	Arg_2	Result
×	2	c	t
-	b	t	a

Arithmetic Operations				Memory Operations			
add	r_1, r_2	⇒	r_3	store	r_1	⇒	r_2
addI	r_1, c_2	⇒	r_3	storeAO	r_1	⇒	r_2, r_3
sub	r_1, r_2	⇒	r_3	storeAI	r_1	⇒	r_2, c_3
subI	r_1, c_2	⇒	r_3	loadI	c_1	⇒	r_3
rsubI	r_2, c_1	⇒	r_3	load	r_1	⇒	r_3
mult	r_1, r_2	⇒	r_3	loadAO	r_1, r_2	⇒	r_3
multI	r_1, c_2	⇒	r_3	loadAI	r_1, c_2	⇒	r_3

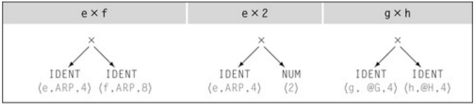
Simple tree walk

Reading a variable and a constant

```
case IDENT:                              case NUM:
  t1 ← base(node);                         result ← NextRegister( );
  t2 ← offset(node);                       emit (loadI, val(node),
  result ← NextRegister( );                           none, result);
  emit (loadAO, t1, t2, result);           break;
  break;
```

- Loading a variables into a register relies on two routines, base and offset
  - A call by reference needs an additional load from the computed address
- Loading a constant
  - If a constant is fit into the immediate field, then use the immediate form
  - Reuse values already stored in a register

Example: Variables on multiply



Generated Code

$e \times f$	$e \times 2$	$g \times h$
		loadI @G ⇒ r_5
		loadI 4 ⇒ r_6
loadI 4 ⇒ r_5		loadAO r_5, r_6 ⇒ r_7
loadAO r_{arp}, r_5 ⇒ r_6	loadI 4 ⇒ r_5	loadI @H ⇒ r_8
loadI 8 ⇒ r_7	loadAO r_{arp}, r_5 ⇒ r_6	loadI 4 ⇒ r_9
loadAO r_{arp}, r_7 ⇒ r_8	loadI 2 ⇒ r_7	loadAO r_8, r_9 ⇒ r_{10}
mult r_6, r_8 ⇒ r_9	mult r_6, r_7 ⇒ r_8	mult r_7, r_{10} ⇒ r_{11}

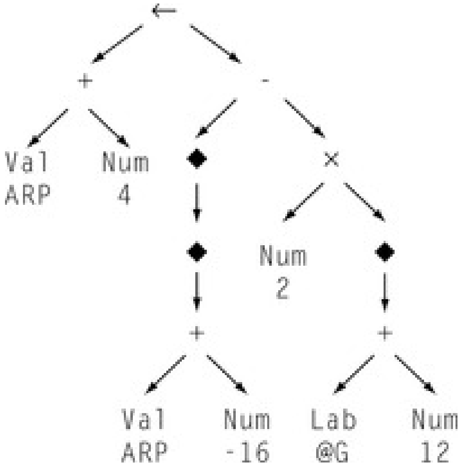
Desired Code

$e \times f$	$e \times 2$	$g \times h$
loadAI $r_{arp}, 4$ ⇒ r_5		loadI 4 ⇒ r_5
loadAI $r_{arp}, 8$ ⇒ r_6	loadAI $r_{arp}, 4$ ⇒ r_5	loadAI r_5, @G ⇒ r_6
mult r_5, r_6 ⇒ r_7	multI $r_5, 2$ ⇒ r_6	loadAI r_5, @H ⇒ r_7
		mult r_6, r_7 ⇒ r_8

The simple abstract syntax tree cannot see the redundancy in the machine codes

Example: Variables on multiply

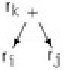
```
a = b - 2 * c
```



We may change the tree into a low-level abstract syntax tree.

Tree-pattern matching

- We may also represent the operation (machine instruction) using a tree.
  - We call this an op-tree.
- We can also linearize an abstract syntax tree in to a prefix form



add $r_i, r_j \Rightarrow r_k$

addI $r_i, c_j \Rightarrow r_k$]

$\leftarrow(+(\mathrm{Val}_1, \mathrm{Num}_1),$
$-(\blacklozenge(\blacklozenge(+(\mathrm{Val}_2, \mathrm{Num}_2))),$
$\times(\mathrm{Num}_3, \blacklozenge(+(\mathrm{Lab}_1, \mathrm{Num}_4)))))$

Then, we are trying to map an AST subtree rooted at an **<ast-node>** with an **<op-tree>**.

Rewrite rules

	Production			Cost	Code Template	
0	Goal	$\rightarrow$	Goal Stmt	0		
1	Goal	$\rightarrow$	Stmt	0		
2	Stmt	$\rightarrow$	$\leftarrow$ (Reg_1,Reg_2)	3	store	$r_2 \Rightarrow r_1$
3	Stmt	$\rightarrow$	$\leftarrow$ ($T1_1$,Reg_2)	3	storeAO	$r_2 \Rightarrow$ T1.r_1,T1.r_2
4	Stmt	$\rightarrow$	$\leftarrow$ ($T2_1$,Reg_2)	3	storeAI	$r_2 \Rightarrow$ T2.r,T2.n
5	Reg	$\rightarrow$	◆ (Reg_1)	3	load	$r_1 \Rightarrow r_{new}$
6	Reg	$\rightarrow$	◆ ($T1_1$)	3	loadAO	T1.r_1,T1.$r_2 \Rightarrow r_{new}$
7	Reg	$\rightarrow$	◆ ($T2_1$)	3	loadAI	T2.r,T2.n $\Rightarrow r_{new}$
8	Reg	$\rightarrow$	+ (Reg_1,Reg_2)	1	add	r_1,$r_2 \Rightarrow r_{new}$
9	Reg	$\rightarrow$	+ (Reg_1,$T3_2$)	1	addI	r_1,T3 $\Rightarrow r_{new}$
10	Reg	$\rightarrow$	+ ($T3_1$,Reg_2)	1	addI	r_2,T3 $\Rightarrow r_{new}$
11	Reg	$\rightarrow$	- (Reg_1,Reg_2)	1	sub	r_1,$r_2 \Rightarrow r_{new}$
12	Reg	$\rightarrow$	- (Reg_1,$T3_2$)	1	subI	r_1,T3 $\Rightarrow r_{new}$
13	Reg	$\rightarrow$	- ($T3_1$,Reg_2)	1	rsubI	r_2,T3 $\Rightarrow r_{new}$
14	Reg	$\rightarrow$	× (Reg_1,Reg_2)	2	mult	r_1,$r_2 \Rightarrow r_{new}$
15	Reg	$\rightarrow$	× (Reg_1,$T3_2$)	2	multI	r_1,T3 $\Rightarrow r_{new}$
16	Reg	$\rightarrow$	× ($T3_1$,Reg_2)	2	multI	r_2,T3 $\Rightarrow r_{new}$
17	Reg	$\rightarrow$	CON_1	1	loadI	$CON_1 \Rightarrow r_{new}$
18	Reg	$\rightarrow$	NUM_1	1	loadI	$NUM_1 \Rightarrow r_{new}$
19	Reg	$\rightarrow$	LAB_1	4	loadI loadAI	@CP $\Rightarrow r_{new_1}$ r_{new_1},@L $\Rightarrow r_{new_2}$
20	Reg	$\rightarrow$	VAL_1	0		
21	T1	$\rightarrow$	+ (Reg_1,Reg_2)	0		
22	T2	$\rightarrow$	+ (Reg_1,$T3_2$)	0		
23	T2	$\rightarrow$	+ ($T3_1$,Reg_2)	0		
24	T3	$\rightarrow$	NUM_1	0		

■ **FIGURE 11.5** Rewrite Rules for Tiling the Low-Level Tree with ILOC.

Restriction on the rule set

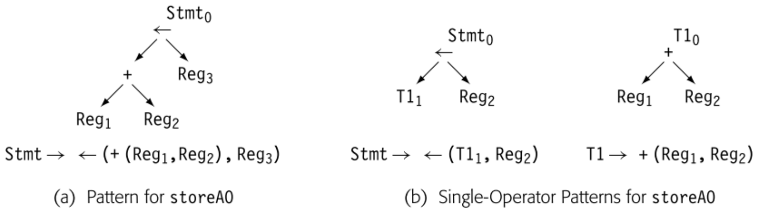
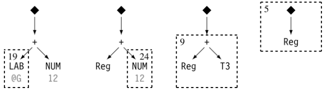


FIGURE 11.6 Single-Operator Patterns Versus Multioperator Patterns.

- Each pattern includes at most one operator
- Leaves in the tree appear only in singleton rules (Rule 17, 20, 24)

Rewrite sequence

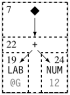
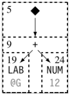
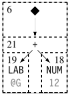
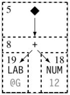


(a) Matches 19 (b) Matches 24 (c) Matches 9 (d) Matches 5

■ **FIGURE 11.7** A Simple Tree Rewrite Sequence.

Potential matches for the variable @G+12

- @G is a label (memory address)

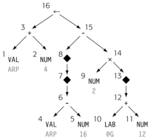
Tiling				
Code	`loadI @CP` $\Rightarrow r_i$ `loadAI` r_i`,@G` $\Rightarrow r_j$ `loadAI` r_j`,12` $\Rightarrow r_k$ Cost: 7 cycles	`loadI @CP` $\Rightarrow r_i$ `loadAI` r_i`,@G` $\Rightarrow r_j$ `addI` r_j`,12` $\Rightarrow r_k$ `load` $r_k \Rightarrow r_l$ Cost: 8 cycles	`loadI @CP` $\Rightarrow r_i$ `loadAI` r_i`,@G` $\Rightarrow r_j$ `loadI 12` $\Rightarrow r_k$ `loadAO` $r_j, r_k \Rightarrow r_l$ Cost: 8 cycles	`loadI @CP` $\Rightarrow r_i$ `loadAI` r_i`,@G` $\Rightarrow r_j$ `loadI 12` $\Rightarrow r_k$ `add` $r_j, r_k \Rightarrow r_l$ `load` $r_l \Rightarrow r_m$ Cost: 9 cycles
Sequences	⟨19,24,22,7⟩ ⟨24,19,22,7⟩	⟨19,24,9,5⟩ ⟨24,19,9,5⟩	⟨19,18,21,6⟩ ⟨18,19,21,6⟩	⟨19,18,8,5⟩ ⟨18,19,8,5⟩

■ **FIGURE 11.8** The Set of All Tilings for the Example Subtree.

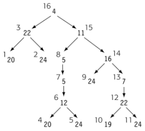
Example

	Production			Cost	Code Template	
0	Goal	→	Goal Stmt	0		
1	Goal	→	Stmt	0		
2	Stmt	→	← (Reg_1,Reg_2)	3	store	$r_2 \Rightarrow r_1$
3	Stmt	→	← ($T1_1$,Reg_2)	3	storeAO	$r_2 \Rightarrow$ T1.r_1,T1.r_2
4	Stmt	→	← ($T2_1$,Reg_2)	3	storeAI	$r_2 \Rightarrow$ T2.r,T2.n
5	Reg	→	◆ (Reg_1)	3	load	$r_1 \Rightarrow r_{new}$
6	Reg	→	◆ ($T1_1$)	3	loadAO	T1.r_1,T1.$r_2 \Rightarrow r_{new}$
7	Reg	→	◆ ($T2_1$)	3	loadAI	T2.r,T2.n $\Rightarrow r_{new}$
8	Reg	→	+ (Reg_1,Reg_2)	1	add	r_1,$r_2 \Rightarrow r_{new}$
9	Reg	→	+ (Reg_1,$T3_2$)	1	addI	r_1,T3 $\Rightarrow r_{new}$
10	Reg	→	+ ($T3_1$,Reg_2)	1	addI	r_2,T3 $\Rightarrow r_{new}$
11	Reg	→	- (Reg_1,Reg_2)	1	sub	r_1,$r_2 \Rightarrow r_{new}$
12	Reg	→	- (Reg_1,$T3_2$)	1	subI	r_1,T3 $\Rightarrow r_{new}$
13	Reg	→	- ($T3_1$,Reg_2)	1	rsubI	r_2,T3 $\Rightarrow r_{new}$
14	Reg	→	× (Reg_1,Reg_2)	2	mult	r_1,$r_2 \Rightarrow r_{new}$
15	Reg	→	× (Reg_1,$T3_2$)	2	multI	r_1,T3 $\Rightarrow r_{new}$
16	Reg	→	× ($T3_1$,Reg_2)	2	multI	r_2,T3 $\Rightarrow r_{new}$
17	Reg	→	CON_1	1	loadI	$CON_1 \Rightarrow r_{new}$
18	Reg	→	NUM_1	1	loadI	$NUM_1 \Rightarrow r_{new}$
19	Reg	→	LAB_1	4	loadI loadAI	@CP $\Rightarrow r_{new_1}$ r_{new_1},@L $\Rightarrow r_{new_2}$
20	Reg	→	VAL_1	0		
21	T1	→	+ (Reg_1,Reg_2)	0		
22	T2	→	+ (Reg_1,$T3_2$)	0		
23	T2	→	+ ($T3_1$,Reg_2)	0		
24	T3	→	NUM_1	0		

FIGURE 11.5 Rewrite Rules for Tiling the Low-Level Tree with ILOC.



(a) Low-Level AST for a ← b - 2 × c



(b) Top-down Assignment of Rules

Node	Rule	Emitted Code			
6	12	subI	r_{arp}, 16	⇒	r_a
7	5	load	r_a	⇒	r_b
8	5	load	r_b	⇒	r_c
10	19	loadI	@CP	⇒	r_d
		loadAI	r_d, @G	⇒	r_e
13	7	loadAI	r_e, 12	⇒	r_f
14	16	multI	r_f, 2	⇒	r_g
15	11	sub	r_c, r_g	⇒	r_h
16	4	storeAI	r_h	⇒	r_{arp}, 4

(c) Code Emitted by Rules

Node	Reg	Stmt	T1	T2	T3
1	**20^{0}**				
2	18^{1}				**24^{0}**
3	8^{2} 9^{1}		21^{1}	**22^{0}**	
4	**20^{0}**				
5	18^{1}				**24^{0}**
6	11^{2} **12^{1}**				
7	**5^{4}**				
8	**5^{7}**				
9	18^{1}				**24^{0}**
10	**19^{4}**				
11	18^{1}				**24^{0}**
12	8^{6} 9^{5}		21^{5}	**22^{4}**	
13	5^{8} 6^{8} **7^{7}**				
14	14^{10} **16^{9}**				
15	**11^{17}**				
16		2^{21} 3^{21} **4^{20}**			

(d) Full Set of Matches and Costs

FIGURE 11.10 Results of Running *Tile* on the Low-Level AST for a ← b - 2 × c.

Tiles and their costs

```
Tile(n)                     /* n is an AST node */
    if n is a leaf then
        Match(n,*) ← { rules that implement n }
    else if n is a unary node then
        Tile(child(n))
        Match(n,*) ← Ø        /* Clear n's Match sets */
        for each rule r where operator(r) = operator(n) do
            if (child(r), child(n)) are compatible then
                add r to Match(n, class(r))
    else if n is a binary node then
        Tile(left(n))
        Tile(right(n))
        Match(n,*) ← Ø        /* Clear n's Match sets */
        for each rule r where operator(r) = operator(n) do
            if (left(r), left(n)) and (right(r), right(n)) are compatible then
                add r to Match(n, class(r))
```

FIGURE 11.9 Compute All Matches to Tile an AST.

```
Tile(n)    /* n is a node in an AST */
    if n is a leaf node then
        Match(n,*).rule ← { low-cost rule that matches n, in each class }
        Match(n,*).cost ← { corresponding cost }
    else if n is a unary node then
        Tile(child(n))
        Match(n,*).rule ← invalid
        Match(n,*).cost ← largest integer
        for each rule r where operator(r) = operator(n) do
            if (child(r), child(n)) are compatible then
                NewCost ← RuleCost(r) + Match(child(n),class(child(r))).cost
                if (Match(n,class(r)).cost > NewCost) then
                    Match(n,class(r)).rule ← r
                    Match(n,class(r)).cost ← NewCost
    else if n is a binary node then
        Tile(left(n))
        Tile(right(n))
        Match(n,*).rule ← invalid
        Match(n,*).cost ← largest integer
        for each rule r where operator(r) = operator(n) do
            if (left(r), left(n)) and (right(r), right(n)) are compatible then
                NewCost ← RuleCost(r) + Match(left(n), class(left(r))).cost
                          + Match(right(n), class(right(r))).cost
                if (Match(n,class(r)).cost > NewCost) then
                    Match(n,class(r)).rule ← r
                    Match(n,class(r)).cost ← NewCost
```

FIGURE 11.11 Compute Low-Cost Matches to Tile an AST.

Tools

Finding the optimal (lowest possible) cost matches

1. Write rules to match an <ast-node> with <op-tree>. This method is suitable for a small instruction set
2. Bottom-up rewrite systems (BURS)
3. Parsing techniques
4. String matching