8 Instruction Selection

Chapter Contents

Instruction Selection

The process of mapping IR operations into target machine operations is called **instruction selection**.

- elaborates details that are hidden in the IR program
- combines multiple ir operations into a single machine instruction

Instruction selection is a **pattern matching** problem. For small instruction sets it can be solved by a hand-coded pass. Larger instruction sets require systematic approaches.

- tree-pattern matching
- peephole optimization

Systematic approaches to code generation make it easier to **retarget** a compiler.

- the compiler writer provides a description of the target ISA
- a tool then constructs a selector for use at compile time

Code Generation

Recall The back end must solve three problems to generate code for a program in IR form: **instruction selection**, **instruction scheduling**, and **register allocation**.

Even though most compilers handle these three processes separately, they are often collectively referred to as **code generation**.

Complexity

- **instruction selection**: while it is not clear how to define optimal instruction selection, the problem involves a huge number of alternatives
- **instruction scheduling**: NP-complete for a basic block under most realistic execution models
- **register allocation**: in its general form, NP-complete in procedures with control flow

Code Generation

The **level** of exposed detail in the IR program matters.

- an IR with a **higher level** of abstraction than the ISA requires the instruction selector to supply additional detail
- an IR with a **lower level** of abstraction than the ISA allows the selector to tailor its selections accordingly

Note Compilers that perform little or no optimization generate code directly from the IR produced by the front end.

Instruction Selection

The complexity of instruction selection arises from the fact that a typical processor provides **many** distinct ways to perform the same computation.

Example Consider an IR construct that copies a value from one general-purpose register `r_i` to another `r_j`.

The obvious implementation uses `i2i r_i ⇒ r_j`, but **each** of the following operations also works.

```
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
```

Still more possibilities exist, *e.g.*, two-operation sequences, including a store followed by a load, also works.

Instruction Selection

Each alternate sequence has its own **costs**.

- modern machines implement simple operations, such as `i2i`, `add`, and `lshift`, so that they execute in a single cycle
- operations, like integer multiplication and division, typically take longer
- the speed of a memory operation depends on many factors, including the detailed current state of the computer's memory system

The **actual cost** can depend on context, *e.g.*, memory state, utilization of the CPU's different functional units, *etc*.

Example If the ALU of an Intel x86-64 CPU is currently busy, the compiler might decide to perform arithmetics using the addressing unit, *i.e.*, use `lea` instead of `imult` and `add`.

Instruction Selection

When the abstraction levels of IR and target ISA differ significantly or the underlying computation models differ, instruction selection plays a critical role in **bridging that gap**.

Consider three scenarios for generating code from an ILOC-like IR

- **Simple, scalar RISC machine** Mapping is straightforward: the code generator might consider only one or two assembly-language sequences for each ir operation.
- **CISC processor** To make effective use of a CISC instruction set, the compiler may need to aggregate several IR operations into one target-machine operation.
- **Stack machine** Code generator must translate the register-to-register style of ILOC into a stack-based style with implicit names and, possibly, destructive operations.

As the gap in abstraction between the IR and the target ISA grows, so does the need for **tools** to help build code generators.

Extending the Treewalk Scheme

On Slide 332, we introduced a **simple treewalk routine** to generate code from the AST of an expression. While this produces correct code, it never capitalizes on the opportunity to tailor the code to specific circumstances and context.

Variables

- AST does not encode their **storage classes**
- routine would need to be extended to deal with variables with different-sized representations, call-by-value and call-by-reference parameters, and variables that already reside in registers

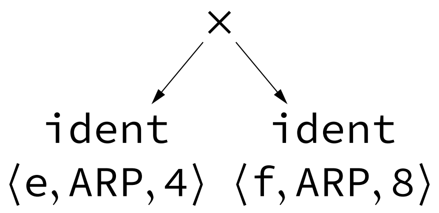
Numbers

- loads number constant into register in every case
- some instructions have immediate forms that can handle the constant directly

```
else if $n$ = ident then
    $t_1 \leftarrow$ base($n$)
    $t_2 \leftarrow$ offset($n$)
    $r \leftarrow$ NextRegister()
    emit(loadAO, $t_1$, $t_2$, $r$)
else if $n$ = num then
    $r \leftarrow$ NextRegister()
    emit(loadI, $n$, _, $r$)
```

Extending the Treewalk Scheme

e × f



Generated Code

```
loadI   4           ⇒ r1
loadAO  rarp, r1    ⇒ r2
loadI   8           ⇒ r3
loadAO  rarp, r3    ⇒ r4
mult    r2, r4      ⇒ r5
```

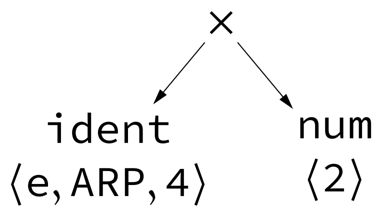
Desired Code

```
loadAI  rarp, 4 ⇒ r1
loadAI  rarp, 8 ⇒ r2
mult    r1, r2  ⇒ r3
```

The inefficiency comes from the fact that the treewalk scheme does **not** generate `loadAI` operations. More complicated code in the `ident` case can cure this problem.

Extending the Treewalk Scheme

$e \times 2$



Generated Code

```
loadI   4         ⇒ r1
loadAO  rarp, r1  ⇒ r2
loadI   2         ⇒ r3
mult    r2, r3    ⇒ r4
```

Desired Code

```
loadAI  rarp, 4  ⇒ r1
multI   r1, 2    ⇒ r2
```

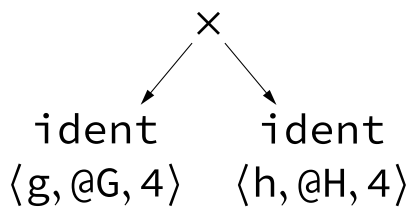
To implement the multiply with a `multI` the code generator must look **beyond** the local context

- the case for × might recognize that one subtree evaluates to a constant
- the code that handles the `num` node might determine that its parent can be implemented with an immediate operation

Either way, it requires non-local context that **violates** the simple treewalk paradigm.

Extending the Treewalk Scheme

$g \times h$



Generated Code

```
loadI   @G       ⇒  r1
loadI   4        ⇒  r2
loadAO  r1, r2   ⇒  r3
loadI   @H       ⇒  r4
loadI   4        ⇒  r5
loadAO  r4, r5   ⇒  r6
mult    r5, r6   ⇒  r7
```

Desired Code

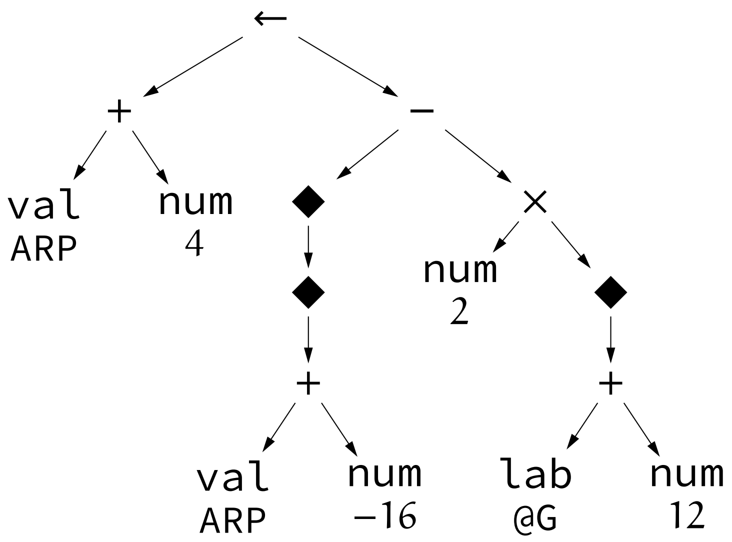
```
loadI   4        ⇒  r1
loadAI  r1, @G   ⇒  r2
loadAI  r1, @H   ⇒  r3
mult    r2, r3   ⇒  r4
```

The fundamental problem with this example lies in the fact that the final code contains a **common subexpression** that was hidden in the AST.

To discover and handle this redundancy, the code generator needs to check the base address and offset values of subtrees and generate appropriate sequences for all cases.

Extending the Treewalk Scheme

A better way of catching this kind of redundancy is to **expose** the redundant details in the IR and let the optimizer eliminate them.



Example Low-level AST for `a ← b − 2 × c`

- `a` is a local variable stored at offset 4 from the ARP
- `b` is a call-by-reference parameter
- `c` is stored at offset 12 from label `@G`

Furthermore, the additions that are implicit in `loadAI` and `storeAI` operations appear explicitly in the tree.

Note Exposing more detail in the AST should lead to better code.

Extending the Treewalk Scheme

The simple treewalk scheme has **one option** for each AST node type. To make effective use of the instruction set, the code generator should consider as many options as possible.

When the code generator considers multiple possible matches for a given subtree, it needs a way to choose among them.

- if the compiler writer can **associate a cos**t with each pattern, then the matching scheme can select patterns in a way that minimizes the costs
- if the costs **truly reflect** performance, this sort of cost-driven instruction selection should lead to good code

Rather than writing code that explicitly navigates the IR and tests the applicability of each operation, the compiler writer should specify rules, and the **tools** should produce the code required to match those rules with the IR form of the code.

Tree-Pattern Matching

To generate code using **tree-pattern matching**, both the IR form of the program and the target machine's instruction set must be expressed as trees.

Example ILOC's addition operations might be modelled by operation trees as shown on the right.

$$\begin{tikzcd} & r_k + \arrow[dl] \arrow[dr] & \\ r_i & & r_j \end{tikzcd}$$

$\texttt{add}\ r_i,\ r_j \Rightarrow r_k$

$$\begin{tikzcd} & r_k + \arrow[dl] \arrow[dr] & \\ r_i & & c_j \end{tikzcd}$$

$\texttt{addI}\ r_i,\ c_j \Rightarrow r_k$

By **systematically** matching operation trees with subtrees of an AST, the compiler can discover all the potential implementations for the subtree.

Tree-Pattern Matching

To work with tree patterns, we need a more convenient notation for describing them.

Using a **prefix notation**, we can write the operation tree for `add` as $+(r_i, r_j)$ and `addI` as $+(r_i, c_j)$, where the symbol `r` denotes an operand in a register and the symbol `c` denotes a known constant operand. Subscripts ensure uniqueness.

Example In prefix notation the low-level AST from Slide 440 can be rewritten as follows.

$$
\begin{aligned}
\leftarrow(&+(\mathtt{Val}_1, \mathtt{Num}_1),\\
&-(\blacklozenge(\blacklozenge(+(\mathtt{Val}_2, \mathtt{Num}_2))),\\
&\times(\mathtt{Num}_3, \blacklozenge(+(\mathtt{Lab}_1, \mathtt{Num}_4)))))
\end{aligned}
$$

Tree-Pattern Matching

Given an AST and a collection of operation trees, the goal is to map the AST to operations by constructing a tiling of the AST with operation trees.

Tiling

A **tiling** is a collection of $\langle \textit{ast-node}, \textit{op-tree} \rangle$ pairs, where *ast-node* is a node in the AST and *op-tree* is an operation tree.

The presence of an $\langle \textit{ast-node}, \textit{op-tree} \rangle$ pair in the tiling means that the target-machine operation represented by *op-tree* **could** implement *ast-node*.

A tiling **implements** the AST if it implements every operation and each tile connects with its neighbors. A tile, $\langle \textit{ast-node}, \textit{op-tree} \rangle$, **connects** with its neighbors if *ast-node* is covered by a leaf in another *op-tree* in the tiling, unless *ast-node* is the root of the AST.

Rewrite Rules

Given a tiling that implements an AST, the compiler can easily generate assembly code in a **bottom-up walk**. Thus, the key to making this approach practical lies in algorithms that quickly find good tilings for an AST.

The relationships between operation trees and subtrees in the AST is encoded as a set of **rewrite rules**. The rule set includes **one or more** rules for every kind of node in the AST.

Rewrite Rules

A rewrite rule consists of a production in a tree grammar, a code template, and an associated cost.

Examples I

	Production			Cost	Code Template			
1	Goal	$\rightarrow$	Assign	0	–			
2	Assign	$\rightarrow$	$\leftarrow(\mathtt{Reg}_1, \mathtt{Reg}_2)$	1	store	r_2	$\Rightarrow$	r_1
3	Assign	$\rightarrow$	$\leftarrow(+(\mathtt{Reg}_1, \mathtt{Reg}_2), \mathtt{Reg}_3)$	1	storeAO	r_3	$\Rightarrow$	r_1, r_2
4	Assign	$\rightarrow$	$\leftarrow(+(\mathtt{Reg}_1, \mathtt{Num}_2), \mathtt{Reg}_3)$	1	storeAI	r_3	$\Rightarrow$	r_1, n_2
5	Assign	$\rightarrow$	$\leftarrow(+(\mathtt{Num}_1, \mathtt{Reg}_2), \mathtt{Reg}_3)$	1	storeAI	r_3	$\Rightarrow$	r_2, n_1
6	Reg	$\rightarrow$	$\mathtt{Lab}_1$	1	load	l_1	$\Rightarrow$	r_{new}
7	Reg	$\rightarrow$	$\mathtt{Val}_1$	0	–			
8	Reg	$\rightarrow$	$\mathtt{Num}_1$	1	load	n_1	$\Rightarrow$	r_{new}

Examples II

9	Reg	$\rightarrow$	$\blacklozenge(\texttt{Reg}_1)$	1	`load`	r_1		$\Rightarrow$	r_{new}
10	Reg	$\rightarrow$	$\blacklozenge(+(\texttt{Reg}_1, \texttt{Reg}_2))$	1	`loadAO`	r_1,	r_2	$\Rightarrow$	r_{new}
11	Reg	$\rightarrow$	$\blacklozenge(+(\texttt{Reg}_1, \texttt{Num}_2))$	1	`loadAI`	r_1,	n_2	$\Rightarrow$	r_{new}
12	Reg	$\rightarrow$	$\blacklozenge(+(\texttt{Num}_1, \texttt{Reg}_2))$	1	`loadAI`	r_2,	n_1	$\Rightarrow$	r_{new}
13	Reg	$\rightarrow$	$\blacklozenge(+(\texttt{Reg}_1, \texttt{Lab}_2))$	1	`loadAI`	r_1,	l_2	$\Rightarrow$	r_{new}
14	Reg	$\rightarrow$	$\blacklozenge(+(\texttt{Lab}_1, \texttt{Reg}_2))$	1	`loadAI`	r_2,	l_1	$\Rightarrow$	r_{new}

Examples III

15	Reg	$\rightarrow$	$+(\mathtt{Reg}_1, \mathtt{Reg}_2)$	1	add	$r_1, r_2 \Rightarrow r_{new}$
16	Reg	$\rightarrow$	$+(\mathtt{Reg}_1, \mathtt{Num}_2)$	1	addI	$r_1, n_2 \Rightarrow r_{new}$
17	Reg	$\rightarrow$	$+(\mathtt{Num}_1, \mathtt{Reg}_2)$	1	addI	$r_2, n_1 \Rightarrow r_{new}$
18	Reg	$\rightarrow$	$+(\mathtt{Reg}_1, \mathtt{Lab}_2)$	1	addI	$r_1, l_2 \Rightarrow r_{new}$
19	Reg	$\rightarrow$	$+(\mathtt{Lab}_1, \mathtt{Reg}_2)$	1	addI	$r_2, l_1 \Rightarrow r_{new}$
20	Reg	$\rightarrow$	$-(\mathtt{Reg}_1, \mathtt{Reg}_2)$	1	sub	$r_1, r_2 \Rightarrow r_{new}$
21	Reg	$\rightarrow$	$-(\mathtt{Reg}_1, \mathtt{Num}_2)$	1	subI	$r_1, n_2 \Rightarrow r_{new}$
22	Reg	$\rightarrow$	$-(\mathtt{Num}_1, \mathtt{Reg}_2)$	1	rsubI	$r_2, n_1 \Rightarrow r_{new}$
23	Reg	$\rightarrow$	$\times(\mathtt{Reg}_1, \mathtt{Reg}_2)$	1	mult	$r_1, r_2 \Rightarrow r_{new}$
24	Reg	$\rightarrow$	$\times(\mathtt{Reg}_1, \mathtt{Num}_2)$	1	multI	$r_1, n_2 \Rightarrow r_{new}$
25	Reg	$\rightarrow$	$\times(\mathtt{Num}_1, \mathtt{Reg}_2)$	1	multI	$r_2, n_1 \Rightarrow r_{new}$

Rewrite Rules

Note The rewrite rules form a **tree grammar** similar to the grammars that we used to specify the syntax of programming languages.

Each rewrite rule, or production, has a **nonterminal** symbol as its left-hand side

- nonterminals in the grammar allow for abstraction
- they serve to connect the rules in the grammar
- they encode knowledge about where a value is stored and what form it takes

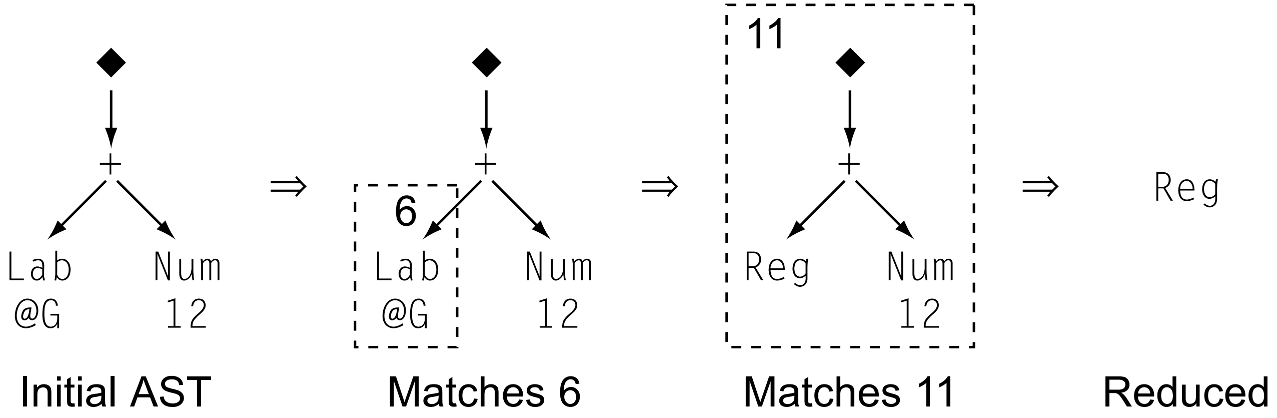
Tree patterns can capture **context** in a way that the simple treewalk code generator cannot. For example, Rules 10 to 14 match two operators ($\blacklozenge$, $+$) to describe in which situations `loadAO` and `loadAI` can be used.

The tree grammar uses **ambiguity** to express the different ways a particular subtree can be implemented. For example, a subtree that matches Rule 10 can also be tiled with the combination of Rules 15 and 9.

Rewrite Rules

To apply these rules to a tree, we look for a **sequence of rewriting steps** that reduces the tree to a single symbol.

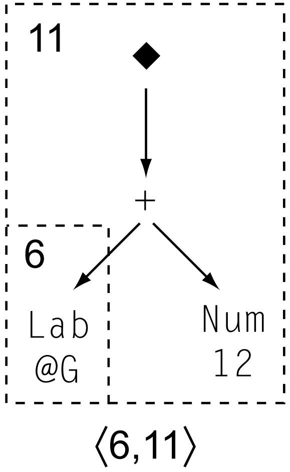
Example Below a rewrite sequence for the subtree that references the variable c from the tree on Slide 440 is shown.



Rewrite Rules

The diagram on the right summarizes this sequence

- dashed boxes show the specific right-hand sides that matched
- the rule number is recorded in the upper left corner of each box
- the list of rule numbers at the bottom indicates the sequence in which the rules were applied
- the rewrite sequence replaces the boxed subtree with the final rule's left-hand side.



Note Nonterminals ensure that the operation trees connect appropriately at the points where they overlap.

- Rule 6 rewrites a `Lab` as a `Reg`
- the left leaf in Rule 11 is a `Reg`

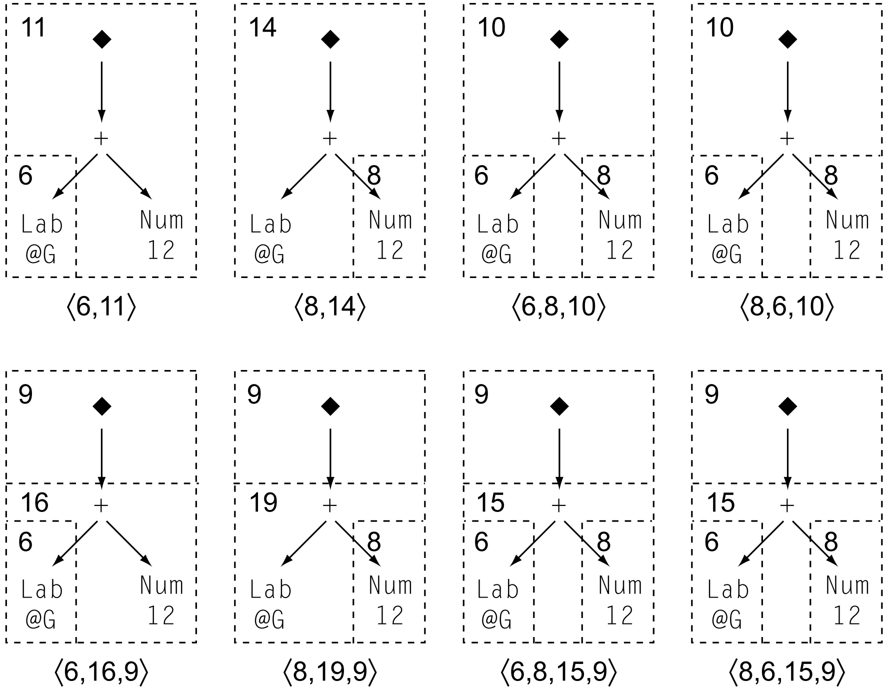
Rewrite Rules

For this trivial subtree, the rules generate **many** rewrite sequences, reflecting the grammar's ambiguity.

To produce assembly code, the selector uses the **code templates** associated with each rule in a bottom-up walk.

The **cost model** drives the code generator to select one of the better sequences.

As none of the sequences use Rule 1 or 7, their cost is equal to their length.



Rewrite Rules

```
loadI   @G      ⇒  r_i
loadAI  r_i,12  ⇒  r_j
```

$\langle 6,11\rangle$

```
loadI   12      ⇒  r_i
loadAI  r_i,@G  ⇒  r_j
```

$\langle 8,14\rangle$

```
loadI   @G        ⇒  r_i
loadI   12        ⇒  r_j
loadAO  r_i,r_j   ⇒  r_k
```

$\langle 6,8,10\rangle$

```
loadI   12        ⇒  r_i
loadI   @G        ⇒  r_j
loadAO  r_i,r_j   ⇒  r_k
```

$\langle 8,6,10\rangle$

```
loadI  @G       ⇒  r_i
addI   r_i,12   ⇒  r_j
load   r_j      ⇒  r_k
```

$\langle 6,16,9\rangle$

```
loadI  12       ⇒  r_i
addI   r_i,@G   ⇒  r_j
load   r_j      ⇒  r_k
```

$\langle 8,19,9\rangle$

```
loadI  @G         ⇒  r_i
loadI  12         ⇒  r_j
add    r_i,r_j    ⇒  r_k
load   r_k        ⇒  r_l
```

$\langle 6,8,15,9\rangle$

```
loadI  12         ⇒  r_i
loadI  @G         ⇒  r_j
add    r_i,r_j    ⇒  r_k
load   r_k        ⇒  r_l
```

$\langle 8,6,15,9\rangle$

Both $\langle 6,11\rangle$ and $\langle 8,14\rangle$ produce the lowest cost. They lead to different, but equivalent code sequences and the selector is **free to choose** between them.

Finding a Tiling

To apply these ideas to code generation, we need an algorithm that can construct a good tiling, *i.e.*, a tiling that produces efficient code.

To simplify the algorithm, we make two assumptions about the form of the rewrite rules

- each operation has, at most, **two** operands
  extending the algorithm to handle the general case is straightforward
- a rule's right-hand side contains at most **one** operation
  transforming the the unrestricted case to this simpler case is simple

The goal of tiling is to label each node in the AST with a **set of patterns** that the compiler can use to implement it.

The algorithm `Tile(n)` (*cf.* next slide) finds tilings for a tree rooted at node `n` in the AST. It computes the LABEL sets in a **postorder traversal**.

```
procedure Tile(n)
    LABEL(n) ← ∅
    if n is a binary node then
        Tile(left(n))
        Tile(right(n))
        foreach rule r that matches n's operation do
            if left(r) ∈ LABEL(left(n)) ∧ right(r) ∈ LABEL(right(n)) then
                LABEL(n) ← LABEL(n) ∪ {r}
    else if n is a unary node then
        Tile(left(n))
        foreach rule r that matches n's operation do
            if left(r) ∈ LABEL(left(n)) then
                LABEL(n) ← LABEL(n) ∪ {r}
    else
        LABEL(n) ← {all rules that match operation in n}
```

Finding the Low-Cost Matches

Among all possible matches, the code generator should find the **lowest-cost match**.

Conceptually, the code generator can discover the lowest-cost match for each subtree in a **bottom-up pass** over the AST. In practice, the process is slightly more complex.

If an operand may be in different **storage classes**, *i.e.*, register, memory, or constant, the code generator needs to track the lowest-cost sequences for each storage class.

This adds a small amount of space and time to the process, but the increase is **bounded** by a factor equal to the number of storage classes.

Note Bottom-up accumulation of costs implements a **dynamic-programming solution** to finding the minimal-cost tiling.

Tools

There are several ways that the compiler writer can implement code generators based on these principles.

1. **Hand-code** a matcher, similar to `Tile`, that explicitly checks for matching rules as it tiles the tree.
2. Since the problem is finite, encode it as a **finite automaton**, *i.e.*, a tree-matching automaton, and obtain the low-cost behavior of a DFA.
3. The grammar-like form of the rules suggests using **parsing techniques** with extensions to handle highly ambiguous grammars
4. By linearizing the tree into a prefix string, the problem can be translated to a **string-matching problem**.

Tools are available that implement each of the last three approaches.

Peephole Optimization

Another technique to perform the matching operations at the heart of instruction selection builds on a technology for late-stage optimization, called **peephole optimization**.

This approach combines **systematic local optimization** on a low-level IR with a **simple scheme for matching** the IR to target-machine operations.

Peephole Optimization

- move a sliding window, or “peephole”, over the code
- examine the operations in the window, looking for specific patterns
- if a pattern is recognized, rewrite it with a better instruction sequence

The **basic premise** of peephole optimization is simple: the compiler can efficiently find local improvements by examining short sequences of adjacent operations.

Peephole Optimization

Example A classic pattern is a store followed by a load from the same location. The load can be replaced by a copy.

$$\begin{array}{llll} \texttt{storeAI} & r_1 & \Rightarrow & r_{arp}, 8 \\ \texttt{loadAI} & r_{arp}, 8 & \Rightarrow & r_{15} \end{array} \longrightarrow \begin{array}{llll} \texttt{storeAI} & r_1 & \Rightarrow & r_{arp}, 8 \\ \texttt{i2i} & r_1 & \Rightarrow & r_{15} \end{array}$$

If the peephole optimizer recognized that this rewrite made the store operation **dead**, it could also eliminate the store operation.

In general, however, recognizing dead stores requires global analysis that is beyond the scope of a peephole optimizer.

Peephole Optimization

Example Simple algebraic identities are also amenable to peephole optimization.

$$\begin{array}{l} \texttt{addI}\ r_2,\ 0\ \Rightarrow\ r_7 \\ \texttt{mult}\ r_4,\ r_7\ \Rightarrow\ r_{10} \end{array} \longrightarrow \texttt{mult}\ r_4,\ r_2\ \Rightarrow\ r_{10}$$

Example Branches for which the target is also a branch can be redirected.

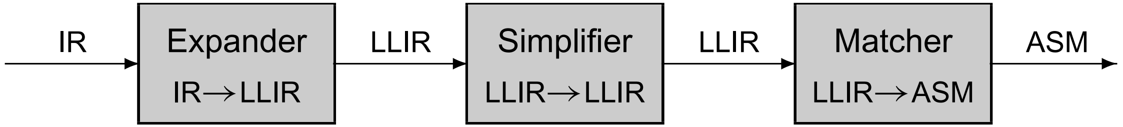
$$\begin{array}{ll} & \texttt{jumpI}\ \rightarrow\ l_{10} \\ l_{10}: & \texttt{jumpI}\ \rightarrow\ l_{11} \end{array} \longrightarrow \begin{array}{ll} & \texttt{jumpI}\ \rightarrow\ l_{11} \\ l_{10}: & \texttt{jumpI}\ \rightarrow\ l_{11} \end{array}$$

If this eliminates the last branch to l_{10}, the basic block beginning at l_{10} becomes unreachable and can be eliminated. Unfortunately, proving that the operation at l_{10} is unreachable takes more analysis than is typically available during peephole optimization.

Peephole Optimization

Early peephole optimizers used a limited set of **hand-coded patterns**.

A modern peephole optimizer breaks the process into three distinct tasks: **expansion**, **simplification**, and **matching**.



Structurally, this looks like a compiler

- if the input and output languages are the same, this system is a peephole optimizer
- with different input and output languages, the same algorithms can perform instruction selection

Peephole Optimization

The **expander** rewrites the IR into a sequence of lower-level IR (LLIR) operations that represents all the direct effects of an operation.

The **simplifier** makes a pass over the LLIR, examining the operations in a small window on the LLIR and systematically trying to improve them. The basic mechanisms are

- forward substitution,
- algebraic simplification,
- evaluating constant-valued expressions, and
- liminating useless effects, *e.g.*, creation of unused condition codes.

The **matcher** compares the simplified LLIR against the pattern library, looking for the pattern that best captures all the effects in the LLIR.

Example The next slides show these three steps for the low-level AST from Slide 440.

Peephole Optimization

expand $\longrightarrow$

$$
\begin{array}{lcl}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{12} & \leftarrow & 12 \\
r_{13} & \leftarrow & r_{11} + r_{12} \\
r_{14} & \leftarrow & M(r_{13}) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{16} & \leftarrow & -16 \\
r_{17} & \leftarrow & r_{arp} + r_{16} \\
r_{18} & \leftarrow & M(r_{17}) \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r_{15} \\
r_{21} & \leftarrow & 4 \\
r_{22} & \leftarrow & r_{arp} + r_{21} \\
M(r_{22}) & \leftarrow & r_{20}
\end{array}
$$

simplify $\longrightarrow$

$$
\begin{array}{lcl}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{14} & \leftarrow & M(r_{11} + 12) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{18} & \leftarrow & M(r_{arp} - 16) \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r_{15} \\
M(r_{arp} + 4) & \leftarrow & r_{20}
\end{array}
$$

Peephole Optimization

$$
\begin{array}{lll}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{14} & \leftarrow & M(r_{11} + 12) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{18} & \leftarrow & M(r_{arp} - 16) \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r_{15} \\
M(r_{arp} + 4) & \leftarrow & r_{20}
\end{array}
\quad \xrightarrow{\text{match}} \quad
\begin{array}{llll}
\texttt{loadI} & 2 & \Rightarrow & r_{10} \\
\texttt{loadI} & @G & \Rightarrow & r_{11} \\
\texttt{loadAI} & r_{11}, r_{12} & \Rightarrow & r_{14} \\
\texttt{mult} & r_{10}, r_{14} & \Rightarrow & r_{15} \\
\texttt{loadAI} & r_{arp}, -16 & \Rightarrow & r_{18} \\
\texttt{load} & r_{18} & \Rightarrow & r_{19} \\
\texttt{sub} & r_{19}, r_{15} & \Rightarrow & r_{20} \\
\texttt{storeAI} & r_{20} & \Rightarrow & r_{arp}, 4
\end{array}
$$

The next slides show the successive sequences that the peephole optimizer has in its window as it processes the low-level IR. Assume that it has a **three-operation window**.

Peephole Optimization

```
r10  ←  2
r11  ←  @G
r12  ←  12
```

Sequence 1

- no simplification is possible
- move window forward

```
r11  ←  @G
r12  ←  12
r13  ←  r11 + r12
```

Sequence 2

- forward substitute `12` into definition of r_{13}
- discard definition of dead register r_{12}

```
r11  ←  @G
r13  ←  r11 + 12
r14  ←  M(r13)
```

Sequence 3

- forward substitute r_{11} + `12` into definition of r_{14}
- discard definition of dead register r_{13}

Peephole Optimization

$$
\boxed{\begin{array}{lll} r_{11} & \leftarrow & @G \\ r_{14} & \leftarrow & M(r_{11} + 12) \\ r_{15} & \leftarrow & r_{10} \times r_{14} \end{array}}
$$

Sequence 4

- no simplification is possible
- move window forward

$$
\boxed{\begin{array}{lll} r_{14} & \leftarrow & M(r_{11} + 12) \\ r_{15} & \leftarrow & r_{10} \times r_{14} \\ r_{16} & \leftarrow & -16 \end{array}}
$$

Sequence 5

- no simplification is possible
- move window forward

$$
\boxed{\begin{array}{lll} r_{15} & \leftarrow & r_{10} \times r_{14} \\ r_{16} & \leftarrow & -16 \\ r_{17} & \leftarrow & r_{arp} + r_{16} \end{array}}
$$

Sequence 6

- forward substitute -16 into definition of r_{17}
- discard definition of dead register r_{16}

Peephole Optimization

$$\boxed{\begin{array}{lll} r_{15} & \leftarrow & r_{10} \times r_{14} \\ r_{17} & \leftarrow & r_{arp} - 16 \\ r_{18} & \leftarrow & M(r_{17}) \end{array}}$$

Sequence 7

- forward substitute $r_{arp} - 16$ into definition of r_{18}
- discard definition of dead register r_{17}

$$\boxed{\begin{array}{lll} r_{15} & \leftarrow & r_{10} \times r_{14} \\ r_{18} & \leftarrow & M(r_{arp} - 16) \\ r_{19} & \leftarrow & M(r_{18}) \end{array}}$$

Sequence 8

- no simplification is possible
- move window forward

$$\boxed{\begin{array}{lll} r_{18} & \leftarrow & M(r_{arp} - 16) \\ r_{19} & \leftarrow & M(r_{18}) \\ r_{20} & \leftarrow & r_{19} - r_{15} \end{array}}$$

Sequence 9

- no simplification is possible
- move window forward

Peephole Optimization

$$\begin{array}{lll} r_{19} & \leftarrow & M(r_{18}) \\ r_{20} & \leftarrow & r_{19} - r_{15} \\ r_{21} & \leftarrow & 4 \end{array}$$

Sequence 10

- no simplification is possible
- move window forward

$$\begin{array}{lll} r_{20} & \leftarrow & r_{19} - r_{15} \\ r_{21} & \leftarrow & 4 \\ r_{22} & \leftarrow & r_{arp} + r_{21} \end{array}$$

Sequence 11

- forward substitute 4 into definition of r_{22}
- discard definition of dead register r_{21}

$$\begin{array}{lll} r_{20} & \leftarrow & r_{19} - r_{15} \\ r_{22} & \leftarrow & r_{arp} + 4 \\ M(r_{22}) & \leftarrow & r_{22} \end{array}$$

Sequence 12

- forward substitute $r_{arp} + 4$ into use of r_{22}
- discard definition of dead register r_{22}

Peephole Optimization

$$\boxed{\begin{array}{l} r_{20} \leftarrow r_{19} - r_{15} \\ M(r_{arp} + 4) \leftarrow r_{22} \end{array}}$$

Sequence 13

- no simplification is possible
- halt as there is no further code to bring into the window

Note Several design issues affect the ability of a peephole optimizer to improve code

- the ability to detect **dead values** plays a critical role in simplification
- the handling of **control-flow** operations determines what happens at block boundaries
- the **size** of the peephole window limits the optimizer's ability to combine related operations