Code Generation

As with parsing, methods for code generation can be classified:

Ad hoc.

As with semantic processing, code could be generated after the parse by traversing the AST. Typically, a combined pre- and post-order traversal suffices, where a node's type prompts the code generator to emit a parameterized template of code.

Systematic

- **Grammar-based (21).**
- **Grammars with attributes (20).**
- **Tree pattern-matching (17, 19).**
- **By peephole processing (11).**

These methods spend more time on instruction selection than can be afforded or managed by *ad hoc.* methods.

In a compiler course, the choice of code generation strategy is key to a successful experience. Many courses stop just before code generation, in which case the students do not experience the elation of watching their compilers actually work.

If the target of translation is reasonably high-level (*e.g.*, a LISP-like intermediate langauge), then *ad hoc.* methods are feasible. In this case, an interpreter should be provided to execute the translated programs.

Otherwise, experience with an automatic code generator is more beneficial. Watch for developments in the `lcc` system (18), which can be obtained by contacting Dave Hanson (`drh@princeton.edu`). If the MIPS instruction set were targeted, then Larus's SPIM simulator (30)(Appendix A) can greatly facilitate debugging the generated code.

Example of a high-level intermediate language

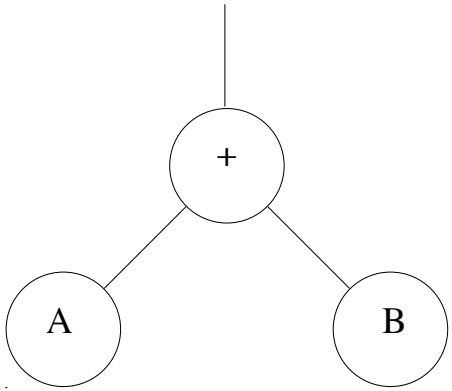
The language FRIL (10) was developed to ease code generation, primarily by resembling LISP and by offering a declarative mechanism for storage association. Each symbol FRIL is declared at most once as any procedure's local or parameter. Each "expression" declares the static depth of its frame, and provides a pointer to its outer scope.

```
int a1;
extern int a2;
int one;
void main() {
   int i;

   int factorial(X)
   int X;
   {
     int Y;
     Y = X;
     if (Y > 0) Y*factorial(X-1);
     else one;
   }

   one = 1;
   a1 = factorial(i=5);
   a2 = factorial(3);
```

```
(Expression 1  /* factorial */
  (PushLevel 5 (LinkExpressionID 2)
    (Args   (SymbolID 7)   /* X */)
    (Locals (SymbolID 8)   /* Y */)
  )
 (Def (SymbolID 8)   /* Y */
  (Use (SymbolID 7)   /* X */)
 )
 (-> 0
  (CHOOSE
    (
      (NE 0
         (GT (Use (SymbolID 8) /* Y */) 0)
      )
        (TIMES
          (Use (SymbolID 8) /* Y */)
          (-> 1
            (MINUS
              (Use (SymbolID 7) /* X */)
              1
            )
          )
        )
    )
    (1 (Use (SymbolID "one")))
  )
 )
)
```

Ad hoc. methods

For example, for a binary $+$ node, the code generator would be called recursively to place the result of the left and right subtrees in two known locations (say, registers R_1 and R_2). Code would then be emitted to form the sum, placing the result in yet another known location.




```
(PLUS
   /* Code for A */
   /* Code for B */
)
```

I usually provide procedures for generating FRIL's symbol table, for generating a PushLevel, and for indenting and formatting the output. The students must decide what constitutes an expression. For example, FRIL has only one control transfer operator: the procedure call. Thus, the body of an iterative loop must be invoked recursively to achieve iteration.

Students write some 200 lines of code to complete the *ad hoc.* code generator for FRIL.

A systematic method — Tree Rewriting

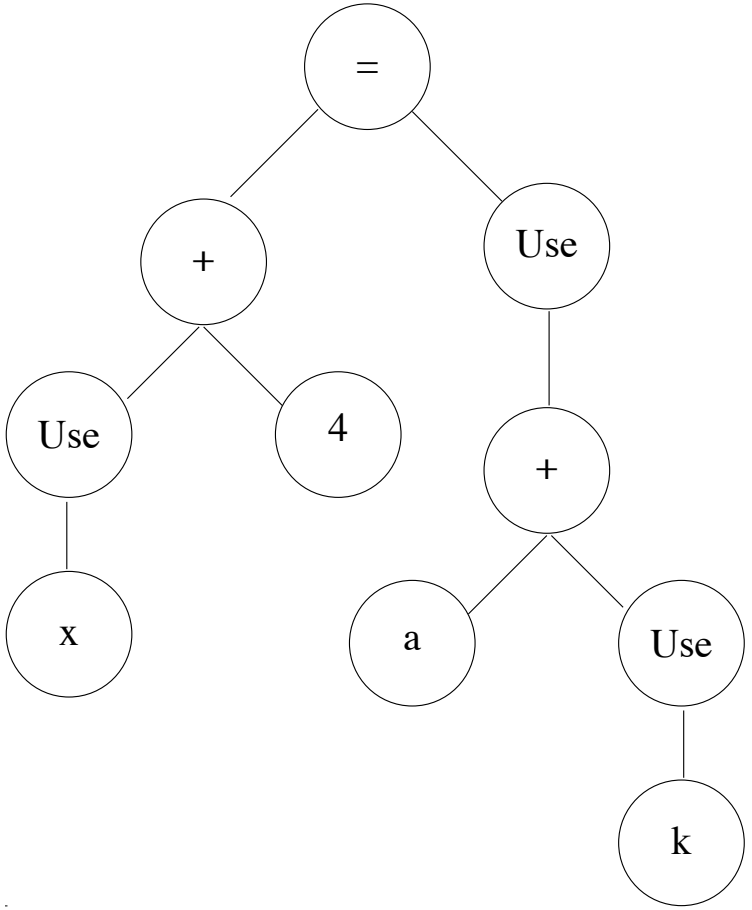
While the *ad hoc.* method services the AST a node at a time, tree rewriting systems can examine larger subtrees and searching for more optimal instruction sequences.

The AST shown to the right is representative of the code fragment

```
*(x+4)=a[k];
```

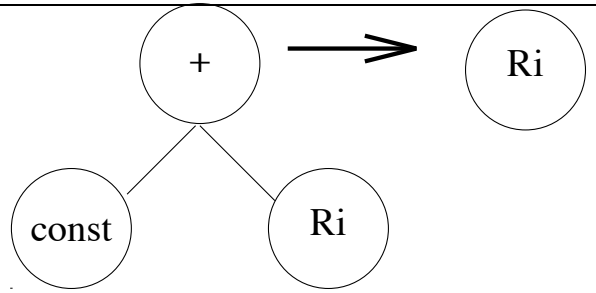
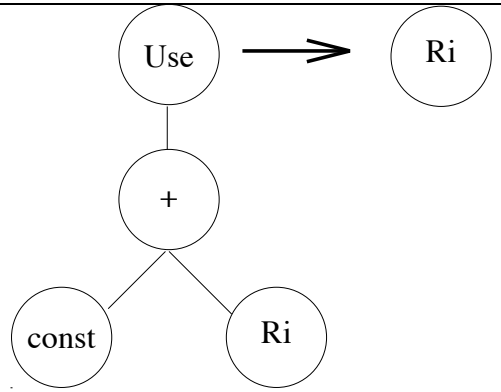
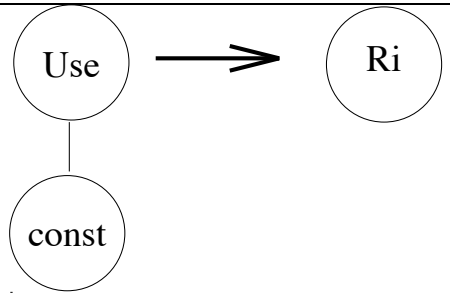
Note that left and right value analysis has already taken place.

Let's assume that from the perspective of code generation, the nodes `x`, `a`, and `k` represent constants. This would be the case had the compiler assigned storage to these variables. If not, then the AST should reflect a level of indirection (probably off a popular register) to reach those variables.



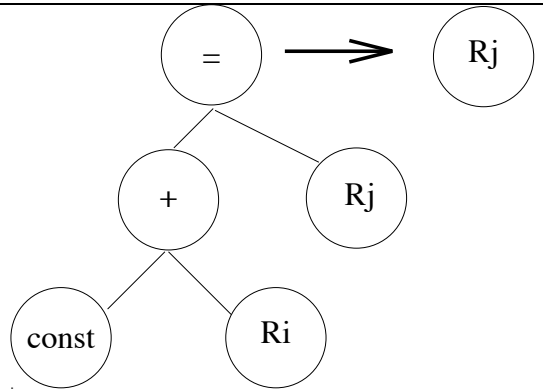
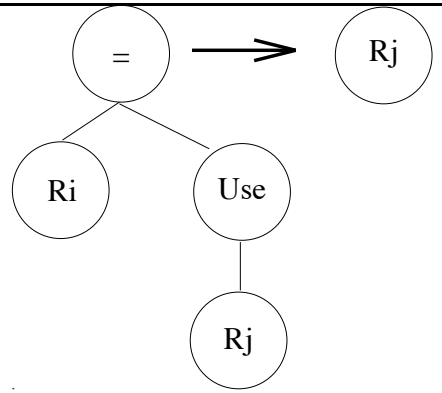
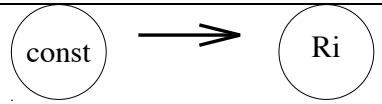
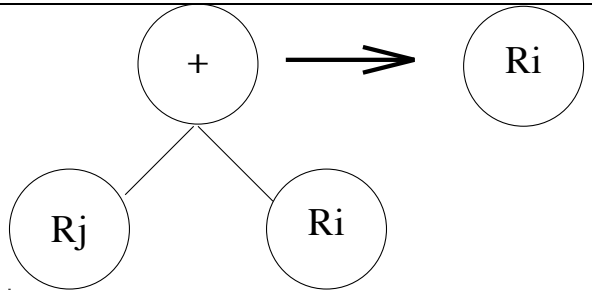
Given the richness of most instruction sets, trying all combinations of instructions to cover the tree would be prohibitively expensive. Most tree matching algorithms use *dynamic programming*, so that results previously holding for some subtree can be reused without additional cost.

Tree rewriting

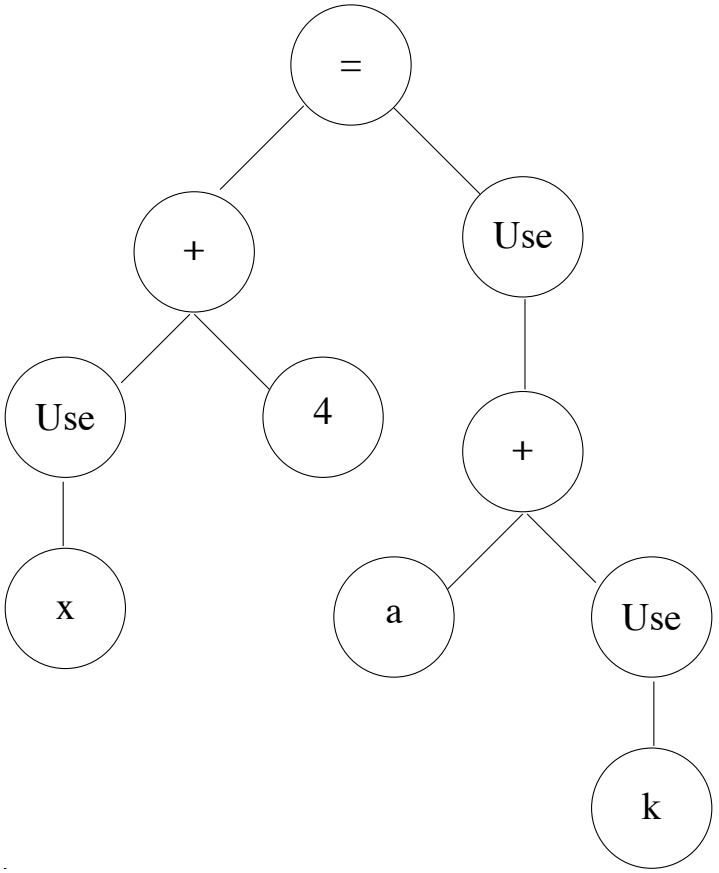
Rule	Rewrite	Instruction	Cost
1		$R_i \leftarrow R_i$+const	1
2		$R_i \leftarrow$ M(R_i+const)	5
3		$R_i \leftarrow$ M(const)	3

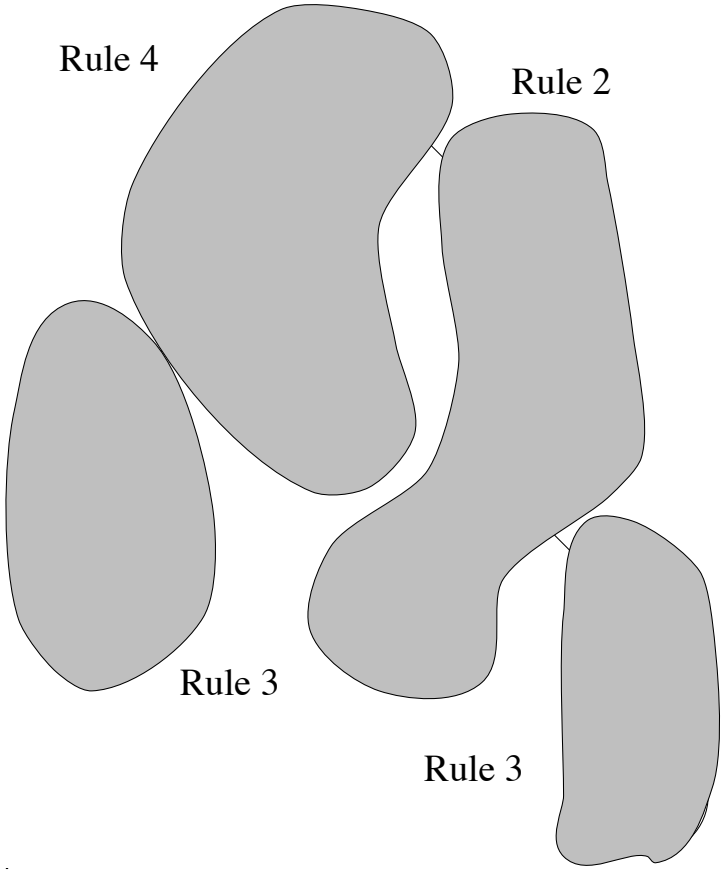
Not shown are the rules that account for the symmetry of addition.

Tree rewriting

Rule	Rewrite	Instruction	Cost
4		M(R_i+const) $\leftarrow R_j$	5
5		M(R_i) $\leftarrow$ M(R_j)	6
6		$R_i \leftarrow$ const	1
7		$R_i \leftarrow R_i$+R_j	1

Example – one way to cover the nodes

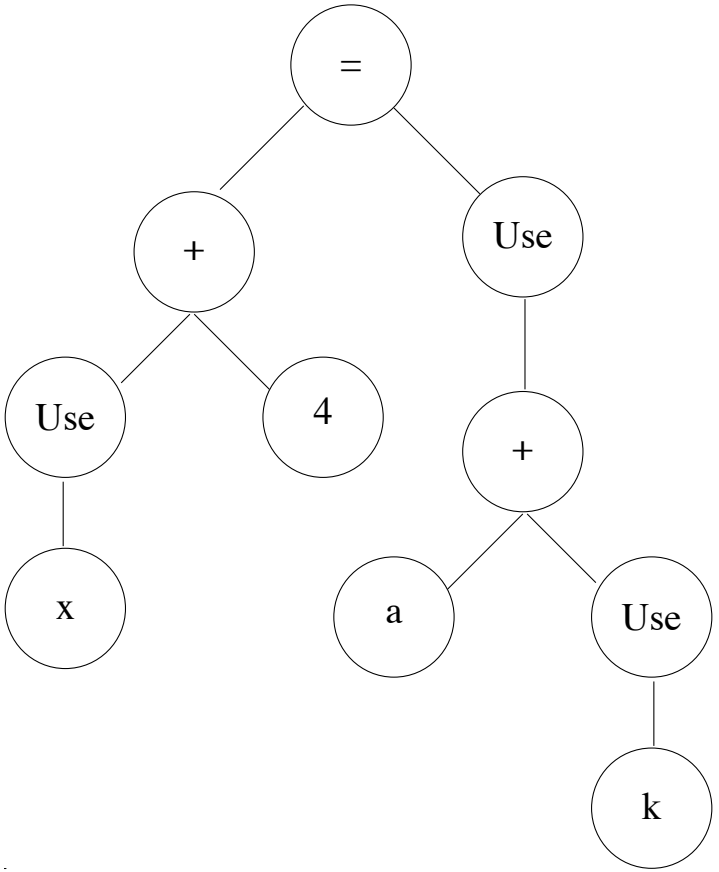


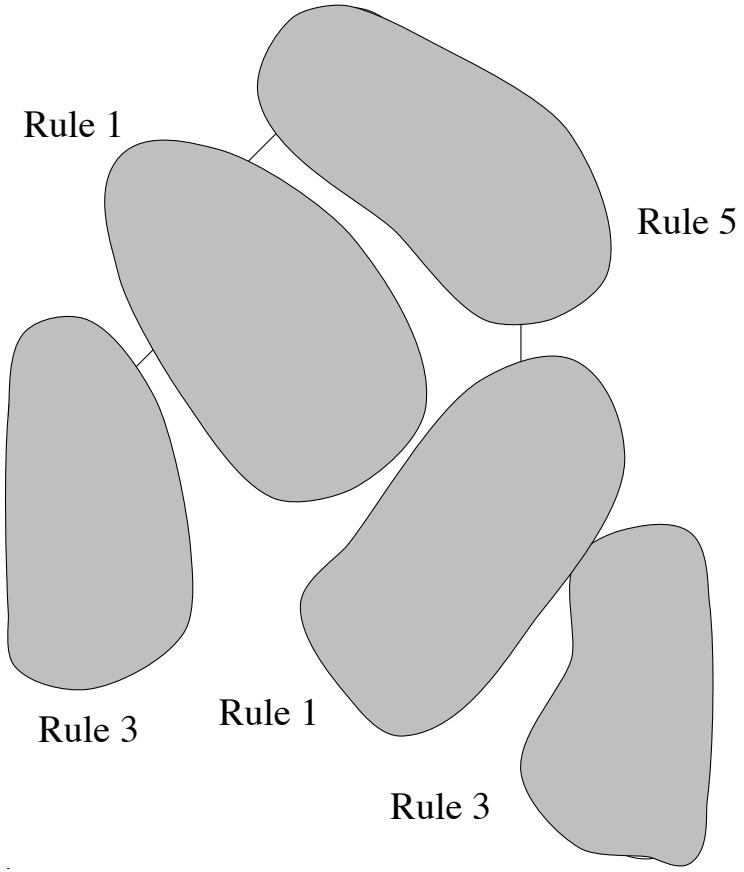


Rule	Instr	Cost
2	$R_i \rightarrow$M(R_i+const)	5
3	$R_i \rightarrow$M(const)	3
4	M(R_i+const)$\rightarrow R_j$	5

Rule	Cost
3	3
2	5
3	3
4	5
	16

Example – another way to cover the nodes





Rule	Instr	Cost
1	$R_i \rightarrow R_i$+const	1
3	$R_i \rightarrow$M(const)	3
5	M(R_i)$\rightarrow$M(R_j)	6

Rule	Cost
3	3
1	1
3	3
1	1
5	6
	14