7

Translation to Intermediate Code

trans-late: to turn into one's own or another language

Webster's Dictionary

The semantic analysis phase of a compiler must translate abstract syntax into abstract machine code. It can do this after type-checking, or at the same time.

Though it is possible to translate directly to real machine code, this hinders portability and modularity. Suppose we want compilers for N different source languages, targeted to M different machines. In principle this is $N \cdot M$ compilers (Figure 7.1a), a large implementation task.

An *intermediate representation* (IR) is a kind of abstract machine language that can express the target-machine operations without committing to too much machine-specific detail. But it is also independent of the details of the source language. The *front end* of the compiler does lexical analysis, parsing, semantic analysis, and translation to intermediate representation. The *back end* does optimization of the intermediate representation and translation to machine language.

A portable compiler translates the source language into IR and then translates the IR into machine language, as illustrated in Figure 7.1b. Now only N front ends and M back ends are required. Such an implementation task is more reasonable.

Even when only one front end and one back end are being built, a good IR can modularize the task, so that the front end is not complicated with machine-specific details, and the back end is not bothered with information specific to one source language. Many different kinds of IR are used in compilers; for this compiler we have chosen simple expression trees.

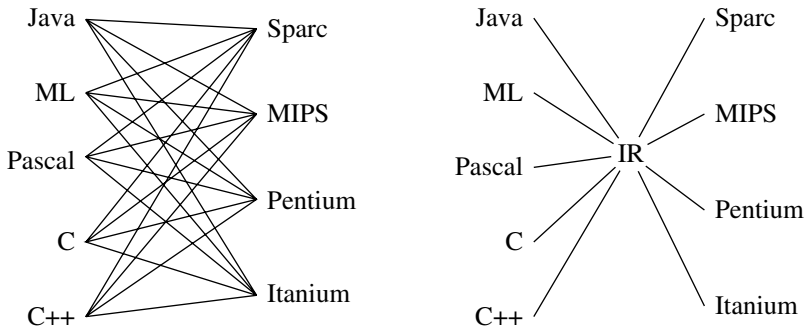


FIGURE 7.1. Compilers for five languages and four target machines: (a) without an IR, (b) with an IR.

7.1 INTERMEDIATE REPRESENTATION TREES

The intermediate representation tree language is defined by the package `Tree`, containing abstract classes `Stm` and `Exp` and their subclasses, as shown in Figure 7.2.

A good intermediate representation has several qualities:

- It must be convenient for the semantic analysis phase to produce.
- It must be convenient to translate into real machine language, for all the desired target machines.
- Each construct must have a clear and simple meaning, so that optimizing transformations that rewrite the intermediate representation can easily be specified and implemented.

Individual pieces of abstract syntax can be complicated things, such as array subscripts, procedure calls, and so on. And individual "real machine" instructions can also have a complicated effect (though this is less true of modern RISC machines than of earlier architectures). Unfortunately, it is not always the case that complex pieces of the abstract syntax correspond exactly to the complex instructions that a machine can execute.

Therefore, the intermediate representation should have individual components that describe only extremely simple things: a single fetch, store, add, move, or jump. Then any "chunky" piece of abstract syntax can be translated into just the right set of abstract machine instructions; and groups of abstract machine instructions can be clumped together (perhaps in quite different clumps) to form "real" machine instructions.

```
package Tree;

abstract class Exp
 CONST(int value)
 NAME(Label label)
 TEMP(Temp.Temp temp)
 BINOP(int binop, Exp left, Exp right)
 MEM(Exp exp)
 CALL(Exp func, ExpList args)
 ESEQ(Stm stm, Exp exp)

abstract class Stm
 MOVE(Exp dst, Exp src)
 EXP(Exp exp)
 JUMP(Exp exp, Temp.LabelList targets)
 CJUMP(int relop, Exp left, Exp right, Label iftrue, Label iffalse)
 SEQ(Stm left, Stm right)
 LABEL(Label label)

 other classes:
 ExpList(Exp head, ExpList tail)
 StmList(Stm head, StmList tail)

 other constants:
final static int BINOP.PLUS, BINOP.MINUS, BINOP.MUL, BINOP.DIV, BINOP.AND,
    BINOP.OR, BINOP.LSHIFT, BINOP.RSHIFT, BINOP.ARSHIFT, BINOP.XOR;

final static int CJUMP.EQ, CJUMP.NE, CJUMP.LT, CJUMP.GT, CJUMP.LE,
        CJUMP.GE, CJUMP.ULT, CJUMP.ULE, CJUMP.UGT, CJUMP.UGE;
```

FIGURE 7.2. Intermediate representation trees.

Here is a description of the meaning of each tree operator. First, the expressions (Exp), which stand for the computation of some value (possibly with side effects):

CONST(i) The integer constant i.

NAME(n) The symbolic constant n (corresponding to an assembly language label).

TEMP(t) Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

BINOP(o, e_1, e_2) The application of binary operator o to operands e_1, e_2. Subexpression e_1 is evaluated before e_2. The integer arithmetic operators are PLUS, MINUS, MUL, DIV; the integer bitwise logical operators are AND, OR, XOR; the integer logical shift operators are LSHIFT, RSHIFT; the integer arithmetic

right-shift is ARSHIFT. The MiniJava language has only one logical operator, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of MiniJava.

MEM(e) The contents of *wordSize* bytes of memory starting at address e (where *wordSize* is defined in the `Frame` module). Note that when MEM is used as the left child of a MOVE, it means "store," but anywhere else it means "fetch."

CALL(f, l) A procedure call: the application of function f to argument list l. The subexpression f is evaluated before the arguments which are evaluated left to right.

ESEQ(s, e) The statement s is evaluated for side effects, then e is evaluated for a result.

The statements (`stm`) of the tree language perform side effects and control flow:

MOVE(TEMP t, e) Evaluate e and move it into temporary t.

MOVE(MEM(e_1), e_2) Evaluate e_1, yielding address a. Then evaluate e_2, and store the result into *wordSize* bytes of memory starting at a.

EXP(e) Evaluate e and discard the result.

JUMP($e, labs$) Transfer control (jump) to address e. The destination e may be a literal label, as in NAME(*lab*), or it may be an address calculated by any other kind of expression. For example, a C-language `switch(i)` statement may be implemented by doing arithmetic on i. The list of labels `labs` specifies all the possible locations that the expression e can evaluate to; this is necessary for dataflow analysis later. The common case of jumping to a known label l is written as JUMP(NAME l, `new LabelList(`l, `null))`, but the JUMP class has an extra constructor so that this can be abbreviated as JUMP(l).

CJUMP(o, e_1, e_2, t, f) Evaluate e_1, e_2 in that order, yielding values a, b. Then compare a, b using the relational operator o. If the result is `true`, jump to t; otherwise jump to f. The relational operators are EQ and NE for integer equality and nonequality (signed or unsigned); signed integer inequalities LT, GT, LE, GE; and unsigned integer inequalities ULT, ULE, UGT, UGE.

SEQ(s_1, s_2) The statement s_1 followed by s_2.

LABEL(n) Define the constant value of name n to be the current machine code address. This is like a label definition in assembly language. The value NAME(n) may be the target of jumps, calls, etc.

It is almost possible to give a formal semantics to the `Tree` language. However, there is no provision in this language for procedure and function definitions – we can specify only the body of each function. The procedure entry and exit sequences will be added later as special "glue" that is different for each target machine.

7.2 TRANSLATION INTO TREES

Translation of abstract syntax expressions into intermediate trees is reasonably straightforward; but there are many cases to handle. We will cover the translation of various language constructs, including many from MiniJava.

KINDS OF EXPRESSIONS

The MiniJava grammar has clearly distinguished statements and expressions. However, in languages such as C, the distinction is blurred; for example, an assignment in C can be used as an expression. When translating such languages, we will have to ask the following question. What should the representation of an abstract-syntax expression be in the `Tree` language? At first it seems obvious that it should be `Tree.Exp`. However, this is true only for certain kinds of expressions, the ones that compute a value. Expressions that return no value are more naturally represented by `Tree.Stm`. And expressions with boolean values, such as $a > b$, might best be represented as a conditional jump – a combination of `Tree.Stm` and a pair of destinations represented by `Temp.Labels`.

It is better instead to ask, "how might the expression be used?" Then we can make the right kind of *methods* for an object-oriented interface to expressions. Both for MiniJava and other languages, we end up with `Translate.Exp`, not the same class as `Tree.Exp`, having three methods:

```
package Translate;
public abstract class Exp {
   abstract Tree.Exp unEx();
   abstract Tree.Stm unNx();
   abstract Tree.Stm unCx(Temp.Label t, Temp.Label f);
}
```

`Ex` stands for an "expression," represented as a `Tree.Exp`.

`Nx` stands for "no result," represented as a `Tree` statement.

`Cx` stands for "conditional," represented as a function from label-pair to statement. If you pass it a true destination and a false destination, it will make a statement that evaluates some conditionals and then jumps to one of the destinations (the statement will never "fall through").

For example, the MiniJava statement

```
if (a<b && c<d) {
       // true block
}
else {
       // false block
}
```

might translate to a `Translate.Exp` whose `unCx` method is roughly like

```
Tree.Stm unCx(Label t, Label f) {
  Label z = new Label();
  return new SEQ(new CJUMP(CJUMP.LT,a,b,z,f),
                 new SEQ(new LABEL(z),
                     new CJUMP(CJUMP.LT,c,d,t,f)));
}
```

The abstract class `Translate.Exp` can be instantiated by several subclasses: `Ex` for an ordinary expression that yields a single value, `Nx` for an expression that yields no value, and `Cx` for a "conditional" expression that jumps to either t or f:

```
class Ex extends Exp {
   Tree.Exp exp;
   Ex(Tree.Exp e) {exp=e;}
   Tree.Exp unEx() {return exp;}
   Tree.Stm unNx() { ···?··· }
   Tree.Stm unCx(Label t, Label f) { ···?··· }
}
class Nx extends Exp {
   Tree.Stm stm;
   Nx(Tree.Stm s) {stm=s;}
   Tree.Exp unEx() { ···?··· }
   Tree.Stm unNx() {return stm;}
   Tree.Stm unCx(Label t, Label f) { ···?··· }
}
```

But what does the `unNx` method of an `Ex` do? We have a simple `Tree.Exp` that yields a value, and we are asked to produce a `Tree.Stm` that produces no value. There is a conversion operator `Tree.EXP`, and `unNx` must apply it:

```
class Ex extends Exp {
  Tree.Exp exp;
  ⋮
  Tree.Stm unNx() {return new Tree.EXP(exp); }
  ⋮
}
```

```
abstract class Cx extends Exp {
  Tree.Exp unEx() {
    Temp r = new Temp();
    Label t = new Label();
    Label f = new Label();

    return new Tree.ESEQ(
            new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r),
                                       new Tree.CONST(1)),
                         new Tree.SEQ(unCx(t,f),
                          new Tree.SEQ(new Tree.LABEL(f),
                           new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r),
                                                      new Tree.CONST(0)),
                            new Tree.LABEL(t)))))),
                        new Tree.TEMP(r));
  }

  abstract Tree.Stm unCx(Label t, Label f);

  Tree.Stm unNx() { ··· }
}
```

PROGRAM 7.3. The `Cx` class.

Each kind of `Translate.Exp` class must have similar conversion methods. For example, the MiniJava statement

```
flag = (a<b && c<d);
```

requires the `unEx` method of a `Cx` object so that a 1 (for true) or 0 (for false) can be stored into `flag`.

Program 7.3 shows the class `Cx`. The `unEx` method is of particular interest. To convert a "conditional" into a "value expression," we invent a new temporary r and new labels t and f. Then we make a `Tree.Stm` that moves the value 1 into r, and a conditional jump `unCx`(t, f) that implements the conditional. If the condition is false, then 0 is moved into r; if it is true, then execution proceeds at t and the second move is skipped. The result of the whole thing is just the temporary r containing zero or one.

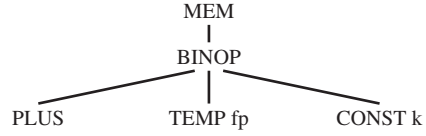
The `unCx` method is still abstract: We will discuss this later, with the translation of comparison operators. But the `unEx` and `unNx` methods can still be implemented in terms of the `unCx` method. We have shown `unEx`; we will leave `unNx` (which is simpler) as an exercise.

The `unCx` method of class `Ex` is left as an exercise. It's helpful to have `unCx` treat the cases of CONST 0 and CONST 1 specially, since they have par-

ticularly simple and efficient translations. Class `Nx`'s `unEx` and `unCx` methods need not be implemented, since these cases should never occur in compiling a well-typed MiniJava program.

SIMPLE VARIABLES

For a simple variable v declared in the current procedure's stack frame, we translate it as



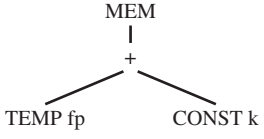
MEM(BINOP(PLUS, TEMP `fp`, CONST k))

where k is the offset of v within the frame and TEMP `fp` is the frame-pointer register. For the MiniJava compiler we make the simplifying assumption that all variables are the same size – the natural word size of the machine.

The `Frame` class holds all machine-dependent definitions; here we add to it a frame-pointer register `FP` and a constant whose value is the machine's natural word size:

```
package Frame;
public class Frame {
   ⋮
   abstract public Temp FP();
   abstract public int wordSize();
}
public abstract class Access {
    public abstract Tree.Exp exp(Tree.Exp framePtr);
}
```

In this and later chapters, we will abbreviate BINOP(PLUS, e_1, e_2) as $+(e_1, e_2)$, so the tree above would be shown as



$+$(TEMP `fp`, CONST k)

The `exp` method of `Frame.Access` is used by `Translate` to turn a `Frame.Access` into the `Tree` expression. The `Tree.Exp` argument is the address of the stack frame that the `Access` lives in. Thus, for an access *a* such as `InFrame`(*k*), we have

```
a.exp(new TEMP(frame.FP())) =
              MEM(BINOP(PLUS,TEMP(frame.FP()),CONST(k)))
```

If *a* is a register access such as `InReg`(t_{832}), then the frame-address argument to *a*`.exp()` will be discarded, and the result will be simply TEMP t_{832}.

An *l*-value such as *v* or *a*[*i*] or *p.next* can appear either on the left side or the right side of an assignment statement – *l* stands for *left*, to distinguish from *r*-values, which can appear only on the right side of an assignment. Fortunately, both MEM and TEMP nodes can appear on the left of a MOVE node.

ARRAY VARIABLES

For the rest of this chapter we will not specify all the interface functions of `Translate`, as we have done for `simpleVar`. But the rule of thumb just given applies to all of them: There should be a `Translate` function to handle array subscripts, one for record fields, one for each kind of expression, and so on.

Different programming languages treat array-valued variables differently.

In Pascal, an array variable stands for the contents of the array – in this case all 12 integers. The Pascal program

```
var a,b : array[1..12] of integer
begin
      a := b
end;
```

copies the contents of array *a* into array *b*.

In C, there is no such thing as an array variable. There are pointer variables; arrays are like "pointer constants." Thus, this is illegal:

```
{int a[12], b[12];
 a = b;
}
```

but this is quite legal:

```
{int a[12], *b;
 b = a;
}
```

The statement `b = a` does not copy the elements of *a*; instead, it means that *b* now points to the beginning of the array *a*.

In MiniJava (as in Java and ML), array variables behave like pointers. MiniJava has no named array constants as in C, however. Instead, new array values are created (and initialized) by the construct `new int[n];` where *n* is the number of elements, and 0 is the initial value of each element. In the program

```
int [] a;
a = new int[12];
b = new int[12];
a = b;
```

the array variable *a* ends up pointing to the same 12 zeros as the variable *b*; the original 12 zeros allocated for *a* are discarded.

MiniJava objects are also pointers. Object assignment, like array assignment, is pointer assignment and does not copy all the fields (see below). This is also true of other object-oriented and functional programming languages, which try to blur the syntactic distinction between pointers and objects. In C or Pascal, however, a record value is "big," and record assignment means copying all the fields.

STRUCTURED *L*-VALUES

An *l*-value is the result of an expression that can occur on the *left* of an assignment statement, such as `x` or `p.y` or `a[i+2]`. An *r*-value is one that can only appear on the *right* of an assignment, such as `a+3` or `f(x)`. That is, an *l*-value denotes a *location* that can be assigned to, and an *r*-value does not.

Of course, an *l*-value can occur on the right of an assignment statement; in this case the *contents* of the location are implicitly taken.

We say that an integer or pointer value is a "scalar," since it has only one component. Such a value occupies just one word of memory and can fit in a register. All the variables and *l*-values in MiniJava are scalar. Even a MiniJava array or object variable is really a pointer (a kind of scalar); the *Java Language Reference Manual* may not say so explicitly, because it is talking about Java semantics instead of Java implementation.

In C or Pascal there are structured *l*-values – structs in C, arrays and records in Pascal – that are not scalar. To implement a language with "large" variables such as the arrays and records in C or Pascal requires a bit of extra work. In a C compiler, the `access` type would require information about the size of the variable. Then, the MEM operator of the TREE intermediate language would

need to be extended with a notion of size:

```
package Tree;
abstract class Exp
 MEM(Exp exp, int size)
```

The translation of a local variable into an IR tree would look like

MEM(+(TEMP fp, CONST k_n), S)

where the S indicates the size of the object to be fetched or stored (depending on whether this tree appears on the left or right of a MOVE).

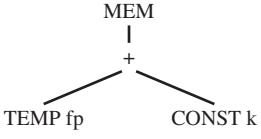
Leaving out the size on MEM nodes makes the MiniJava compiler easier to implement, but limits the generality of its intermediate representation.

SUBSCRIPTING AND FIELD SELECTION

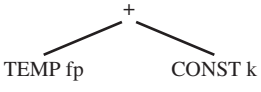
To subscript an array in Pascal or C (to compute $a[i]$), just calculate the address of the ith element of a: $(i-l)\times s+a$, where l is the lower bound of the index range, s is the size (in bytes) of each array element, and a is the base address of the array elements. If a is global, with a compile-time constant address, then the subtraction $a-s\times l$ can be done at compile time.

Similarly, to select field f of a record l-value a (to calculate $a.f$), simply add the constant field offset of f to the address a.

An array variable a is an l-value; so is an array subscript expression $a[i]$, even if i is not an l-value. To calculate the l-value $a[i]$ from a, we do arithmetic on the address of a. Thus, in a Pascal compiler, the translation of an l-value (particularly a structured l-value) should *not* be something like



but should instead be the `Tree` expression representing the base address of the array:

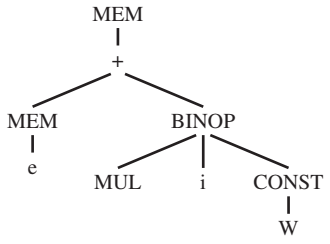


What could happen to this l-value?

- A particular element might be subscripted, yielding a (smaller) l-value. A "+" node would add the index times the element size to the l-value for the base of the array.
- The l-value (representing the entire array) might be used in a context where an r-value is required (e.g., passed as a by-value parameter, or assigned to another array variable). Then the l-value is *coerced* into an r-value by applying the MEM operator to it.

In the MiniJava language, there are no structured, or "large," l-values. This is because all object and array values are really pointers to object and array structures. The "base address" of the array is really the contents of a pointer variable, so MEM is required to fetch this base address.

Thus, if a is a memory-resident array variable represented as MEM(e), then the contents of address e will be a one-word pointer value p. The contents of addresses p, $p + W$, $p + 2W$, ... (where W is the word size) will be the elements of the array (all elements are one word long). Thus, $a[i]$ is just



MEM(+(MEM(e), BINOP(MUL, i, CONST W)))

***l*-values and MEM nodes.** Technically, an l-value (or *assignable variable*) should be represented as an *address* (without the top MEM node in the diagram above). Converting an l-value to an r-value (when it is used in an expression) means *fetching* from that address; assigning to an l-value means *storing* to that address. We are attaching the MEM node to the l-value before knowing whether it is to be fetched or stored; this works only because in the `Tree` intermediate representation, MEM means both *store* (when used as the left child of a MOVE) and *fetch* (when used elsewhere).

A SERMON ON SAFETY

Life is too short to spend time chasing down irreproducible bugs, and money is too valuable to waste on the purchase of flaky software. When a program has a bug, it should detect that fact as soon as possible and announce that fact (or take corrective action) before the bug causes any harm.

Some bugs are very subtle. But it should not take a genius to detect an out-of-bounds array subscript: If the array bounds are $L..H$, and the subscript is i, then $i < L$ or $i > H$ is an array bounds error. Furthermore, computers are well-equipped with hardware able to compute the condition $i > H$. For several decades now, we have known that compilers can automatically emit the code to test this condition. There is no excuse for a compiler that is unable to emit code for checking array bounds. Optimizing compilers can often *safely* remove the checks by compile-time analysis; see Section 18.4.

One might say, by way of excuse, "but the language in which I program has the kind of address arithmetic that makes it impossible to know the bounds of an array." Yes, and the man who shot his mother and father threw himself upon the mercy of the court because he was an orphan.

In some rare circumstances, a portion of a program demands blinding speed, and the timing budget does not allow for bounds checking. In such a case, it would be best if the optimizing compiler could analyze the subscript expressions and prove that the index will always be within bounds, so that an explicit bounds check is not necessary. If that is not possible, perhaps it is reasonable in these rare cases to allow the programmer to explicitly specify an unchecked subscript operation. But this does not excuse the compiler from checking all the other subscript expressions in the program.

Needless to say, the compiler should check pointers for `nil` before dereferencing them, too.[1]

ARITHMETIC

Integer arithmetic is easy to translate: Each arithmetic operator corresponds to a `Tree` operator.

The `Tree` language has no unary arithmetic operators. Unary negation of integers can be implemented as subtraction from zero; unary complement can be implemented as XOR with all ones.

Unary floating-point negation cannot be implemented as subtraction from zero, because many floating-point representations allow a *negative zero*. The negation of negative zero is positive zero, and vice versa. Thus, the `Tree` language does not support unary negation very well.

Fortunately, the MiniJava language doesn't support floating-point numbers; but in a real compiler, a new operator would have to be added for floating negation.

[1] A different way of checking for `nil` is to unmap page 0 in the virtual-memory page tables, so that attempting to fetch/store fields of a `nil` record results in a page fault.

CONDITIONALS

The result of a comparison operator will be a `Cx` expression: a statement s that will jump to any true-destination and false-destination you specify.

Making "simple" `Cx` expressions from comparison operators is easy with the `CJUMP` operator. However, the whole point of the `Cx` representation is that conditional expressions can be combined easily with the MiniJava operator `&&`. Therefore, an expression such as `x<5` will be translated as `Cx`(s_1), where

$$s_1(t, f) = \text{CJUMP}(\text{LT}, x, \text{CONST}(5), t, f)$$

for any labels t and f.

To do this, we extend the `Cx` class to make a subclass `RelCx` that has private fields to hold the left and right expressions (in this case x and 5) and the comparison operator (in this case `Tree.CJUMP.LT`). Then we override the `unCx` method to generate the `CJUMP` from these data. It is not necessary to make `unEx` and `unNx` methods, since these will be inherited from the parent `Cx` class.

The most straightforward thing to do with an `if`-expression

if e_1 **then** e_2 **else** e_3

is to treat e_1 as a `Cx` expression, and e_2 and e_3 as `Ex` expressions. That is, use the `unCx` method of e_1 and the `unEx` of e_2 and e_3. Make two labels t and f to which the conditional will branch. Allocate a temporary r, and after label t, move e_2 to r; after label f, move e_3 to r. Both branches should finish by jumping to a newly created "join" label.

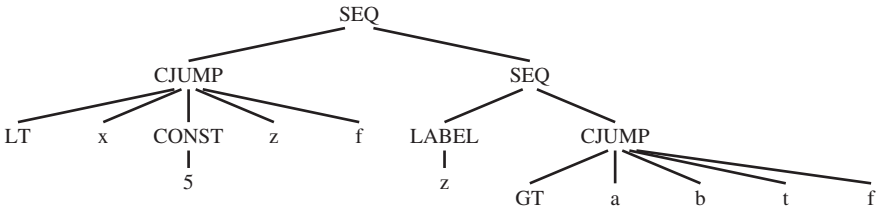
This will produce perfectly correct results. However, the translated code may not be very efficient at all. If e_2 and e_3 are both "statements" (expressions that return no value), then their representation is likely to be `Nx`, not `Ex`. Applying `unEx` to them will work – a coercion will automatically be applied – but it might be better to recognize this case specially.

Even worse, if e_2 or e_3 is a `Cx` expression, then applying the `unEx` coercion to it will yield a horrible tangle of jumps and labels. It is much better to recognize this case specially.

For example, consider

if $x < 5$ **then** $a > b$ **else** 0

As shown above, $x < 5$ translates into `Cx`(s_1); similarly, $a > b$ will be translated as `Cx`(s_2) for some s_2. The whole `if`-statement should come out approximately as



$\text{SEQ}(s_1(z, f),\ \text{SEQ}(\text{LABEL}\ z, s_2(t, f)))$

for some new label z.

Therefore, the translation of an **if** requires a new subclass of `Exp`:

```
class IfThenElseExp extends Exp {
   Exp cond, a, b;
   Label t = new Label();
   Label f = new Label();
   Label join = new Label();
   IfThenElseExp(Exp cc, Exp aa, Exp bb) {
            cond=cc; a=aa; b=bb;
   }
   Tree.Stm unCx(Label tt, Label ff) { ··· }
   Tree.Exp unEx() { ··· }
   Tree.Stm unNx() { ··· }
}
```

The labels `t` and `f` indicate the beginning of the then-clause and else-clause, respectively. The labels `tt` and `ff` are quite different: These are the places to which conditions inside the then-clause (or else-clause) must jump, depending on the truth of those subexpressions.

STRINGS

A string literal is typically implemented as the constant address of a segment of memory initialized to the proper characters. In assembly language, a label is used to refer to this address from the middle of some sequence of instructions. At some other place in the assembly-language program, the *definition* of that label appears, followed by the assembly-language pseudo-instruction to reserve and initialize a block of memory to the appropriate characters.

For each string literal `lit`, a translator must make a new label `lab`, and return the tree `Tree.NAME(lab)`. It should also put the assembly-language fragment `frame.string(lab,lit)` onto a global list of such fragments to be handed to the code emitter. "Fragments" are discussed further on page 157.

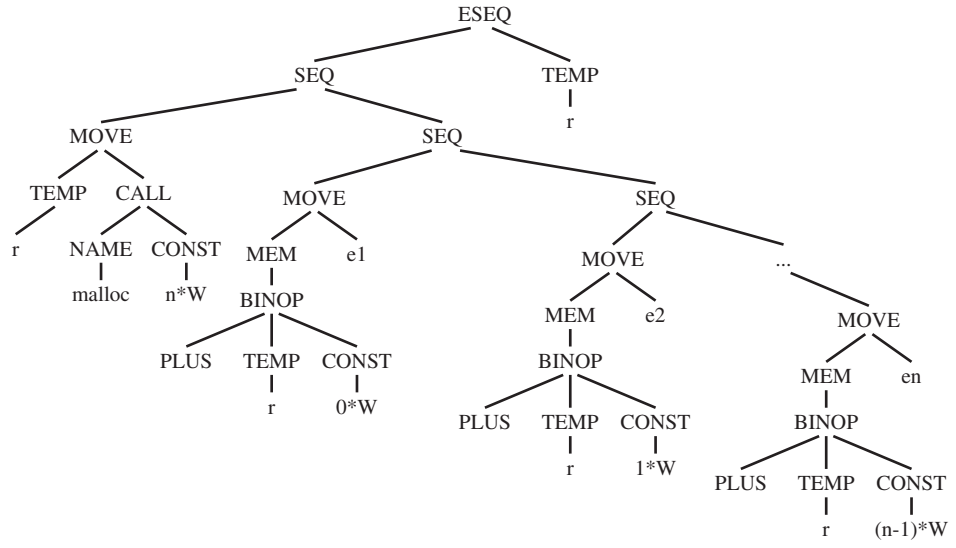


FIGURE 7.4. Object initialization.

All string operations are performed in functions provided by the runtime system; these functions heap-allocate space for their results, and return pointers. Thus, the compiler (almost) doesn't need to know what the representation is, as long as it knows that each string pointer is exactly one word long. We say "almost" because string literals must be represented.

In Pascal, strings are fixed-length arrays of characters; literals are padded with blanks to make them fit. This is not very useful. In C, strings are pointers to variable-length, zero-terminated sequences. This is much more useful, though a string containing a zero byte cannot be represented.

RECORD AND ARRAY CREATION

Imagine a language construct $\{e_1, e_2, ..., e_n\}$ which creates an n-element record initialized to the values of expressions e_i. This is like an object constructor that initializes all the instance variables of the object. Such a record may outlive the procedure activation that creates it, so it cannot be allocated on the stack. Instead, it must be allocated on the *heap*. If there is no provision for freeing records (or strings), industrial-strength systems should have a *garbage collector* to reclaim unreachable records (see Chapter 13).

The simplest way to create a record is to call an external memory-allocation function that returns a pointer to an n-word area into a new temporary r. Then

a series of MOVE trees can initialize offsets $0, 1W, 2W, ..., (n-1)W$ from r with the translations of expressions e_i. Finally, the result of the whole expression is TEMP(r), as shown in Figure 7.4.

In an industrial compiler, calling `malloc` (or its equivalent) on every record creation might be too slow; see Section 13.7.

Array creation is very much like record creation, except that all the fields are initialized to the same value. The external `initArray` function can take the array length and the initializing value as arguments, see later.

In MiniJava we can create an array of integers by the construct

```
new int [exp]
```

where `exp` is an expression that evaluates to an integer. This will create an integer array of a length determined by the value of `exp` and with each value initialized to zero.

To translate array creation, the compiler needs to perform the following steps:

1. Determine how much space is needed for the array. This can be calculated by:

   ((length of the array) + 1) × (size of an integer, e.g., 4).

   The reason we add one to the length of the array is that we want to store the length of the array along with the array. This is needed for bounds checking and to determine the length at run time.
2. Call an external function to get space on the heap. The call will return a pointer to the beginning of the array.
3. Generate code for saving the length of the array at offset 0.
4. Generate code for initializing each of the values in the array to zero starting at offset 4.

Calling runtime-system functions. To call an external function named `initArray` with arguments a, b, simply generate a CALL such as

```
static Label initArray = new Label("initArray");
new CALL(new NAME(initArray),
         new Tree.ExpList(a, new Tree.ExpList(b, null)));
```

This refers to an external function `initArray` which is written in a language such as C or assembly language – it cannot be written in MiniJava because MiniJava has no mechanism for manipulating raw memory.

But on some operating systems, the C compiler puts an underscore at the beginning of each label; and the calling conventions for C functions may differ from those of MiniJava functions; and C functions don't expect to receive a static link, and so on. All these target-machine-specific details should be encapsulated into a function provided by the `Frame` structure:

```
public abstract class Frame {
   ⋮
   abstract public Tree.Exp externalCall(String func,
                                         Tree.ExpList args);
}
```

where `externalCall` takes the name of the external procedure and the arguments to be passed.

The implementation of `externalCall` depends on the relationship between MiniJava's procedure-call convention and that of the external function. The simplest possible implementation looks like

```
Tree.Exp externalCall(String s, Tree.ExpList args) {
     return new Tree.CALL(new Tree.NAME(new Temp.Label(s)),
                          args);
}
```

but may have to be adjusted for static links, or underscores in labels, and so on. Also, calling `new Label(`*s*`)` repeatedly with the same *s* makes several label objects that all mean the same thing; this may confuse other parts of the compiler, so it might be useful to maintain a string-to-label table to avoid duplication.

WHILE LOOPS

The general layout of a **while** loop is

test:
    if not(*condition*) goto *done*
    body
    goto *test*
done:

If a **break** statement occurs within the *body* (and not nested within any interior **while** statements), the translation is simply a JUMP to *done*.

Translation of **break** statements needs to have a new formal parameter `break` that is the *done* label of the nearest enclosing loop. In translating a

while loop, the translator will be called recursively upon *body* with the *done* label passed as the `break` parameter. When the translator is recursively calling itself in nonloop contexts, it can simply pass down the same `break` parameter that was passed to it.

FOR LOOPS

A **for** statement can be expressed using other kinds of statements:

```
for (i=lo; i<=hi; i++;) {
        // body
}
```

```
i=lo;
limit=hi;
while (i<=limit) {
        // body
        i++;
}
```

A straightforward approach to the translation of **for** statements is to rewrite the *abstract syntax* into the abstract syntax of the **while** statement shown, and then translate the result.

This is almost right, but consider the case where *limit*=*maxint*. Then $i + 1$ will overflow; either a hardware exception will be raised, or $i \leq limit$ will always be true! The solution is to put the test at the *bottom* of the loop, where $i < limit$ can be tested *before* the increment. Then an extra test will be needed before entering the loop to check $lo \leq hi$.

FUNCTION CALL

Translating a function call $f(a_1, ...a_n)$ is simple:

$$\text{CALL}(\text{NAME}\ l_f, [e_1, e_2, ..., e_n])$$

where l_f is the label for f. In an object-oriented language, the implicit variable `this` must be made an explicit argument of the call. That is, $p.m(a_1, ...a_n)$ is translated as

$$\text{CALL}(\text{NAME}\ l_{c\$m}, [p, e_1, e_2, ..., e_n])$$

where p belongs to class c, and $c\$m$ is the m method of class c. For a static method, the computation of address $l_{c\$m}$ can be done at compile time – it's a simple label, as it is in MiniJava. For dynamic methods, the computation is more complicated, as explained in Chapter 14.

STATIC LINKS

Some programming languages (such as Pascal, Scheme, and ML) support nesting of functions so that the inner functions can refer to variables declared in the outer functions. When building a compiler for such a language, frame representations and variable access are a bit more complicated.

When a variable x is declared at an outer level of static scope, static links must be used. The general form is

$$\text{MEM}(+(\text{CONST } k_n,\ \text{MEM}(+(\text{CONST } k_{n-1}, \ldots \\ \text{MEM}(+(\text{CONST } k_1,\ \text{TEMP FP}))\ldots))))$$

where the k_1, ..., k_{n-1} are the various static-link offsets in nested functions, and k_n is the offset of x in its own frame.

In creating `TEMP` variables, those temporaries that escape (i.e., are called from within an inner function) must be allocated in the stack frame, not in a register. When accessing such a temporary from either the same function or an inner function, we must pass the appropriate static link. The `exp` method of `Frame.Access` would need to calculate the appropriate chain of dereferences.

Translating a function call $f(a_1, ...a_n)$ using static links requires that the static link must be added as an implicit extra argument:

$$\text{CALL}(\text{NAME } l_f, [sl, e_1, e_2, ..., e_n])$$

Here l_f is the label for f, and sl is the static link, computed as described in Chapter 6. To do this computation, both the `level` of f and the `level` of the function calling f are required. A chain of (zero or more) offsets found in successive `level` descriptors is fetched, starting with the frame pointer TEMP(FP) defined by the `Frame` module.

7.3 DECLARATIONS

For each variable declaration within a function body, additional space will be reserved in the `frame`. Also, for each function declaration, a new "fragment" of `Tree` code will be kept for the function's body.

VARIABLE DEFINITION

The translation of a variable declaration should return an augmented type environment that is used in processing the remainder of the function body.

However, the initialization of a variable translates into a `Tree` expression that must be put just before the body of the function. Therefore, the translator must return a `Translate.Exp` containing assignment expressions that accomplish these initializations.

If the translator is applied to function and type declarations, the result will be a "no-op" expression such as `Ex(CONST(0))`.

FUNCTION DEFINITION

A function is translated into a segment of assembly language with a *prologue*, a *body*, and an *epilogue*. The body of a function is an expression, and the *body* of the translation is simply the translation of that expression.

The *prologue*, which precedes the body in the assembly-language version of the function, contains

1. pseudo-instructions, as needed in the particular assembly language, to announce the beginning of a function;
2. a label definition for the function name;
3. an instruction to adjust the stack pointer (to allocate a new frame);
4. instructions to save "escaping" arguments into the frame, and to move nonescaping arguments into fresh temporary registers;
5. store instructions to save any callee-save registers – including the return address register – used within the function.

Then comes

6. the function *body*.

The *epilogue* comes after the body and contains

7. an instruction to move the return value (result of the function) to the register reserved for that purpose;
8. load instructions to restore the callee-save registers;
9. an instruction to reset the stack pointer (to deallocate the frame);
10. a *return* instruction (JUMP to the return address);
11. pseudo-instructions, as needed, to announce the end of a function.

Some of these items (1, 3, 9, and 11) depend on exact knowledge of the frame size, which will not be known until after the register allocator determines how many local variables need to be kept in the frame because they don't fit in registers. So these instructions should be generated very late, in a `FRAME` function called `procEntryExit3` (see also page 252). Item 2 (and 10), nestled between 1 and 3 (and 9 and 11, respectively) are also handled at that time.

To implement 7, the `Translate` phase should generate a move instruction

```
MOVE(RV, body)
```

that puts the result of evaluating the body in the return value (`RV`) location specified by the machine-specific `frame` structure:

```
package Frame;
public abstract class Frame {
   ⋮
   abstract public Temp RV();
}
```

Item 4 (moving incoming formal parameters), and 5 and 8 (the saving and restoring of callee-save registers), are part of the *view shift* described on page 128. They should be done by a function in the `Frame` module:

```
package Frame;
public abstract class Frame {
   ⋮
   abstract public Tree.Stm procEntryExit1(Tree.Stm body);
}
```

The implementation of this function will be discussed on page 251. `Translate` should apply it to each procedure body (items 5–7) as it is translated.

FRAGMENTS

Given a function definition comprising an already-translated `body` expression, the `Translate` phase should produce a descriptor for the function containing this necessary information:

frame: The frame descriptor containing machine-specific information about local variables and parameters;
body: The result returned from `procEntryExit1`.

Call this pair a *fragment* to be translated to assembly language. It is the second kind of fragment we have seen; the other was the assembly-language pseudo-instruction sequence for a string literal. Thus, it is useful to define (in the `Translate` interface) a `frag` datatype:

```
package Translate;
public abstract class Frag {  public Frag next; }
public ProcFrag(Tree.Stm body, Frame.Frame frame);
public DataFrag(String data);
```

```
class Vehicle {
    int position;
    int gas;
    int move (int x) {
        position = position + x;
        return position;
    }
    int fill (int y) {
        gas = gas + y;
        return gas;
    }
}
```

PROGRAM 7.5. A MiniJava program.

```
public class Translate {
   ⋮
  private Frag frags;    // linked list of accumulated fragments
  public void procEntryExit(Exp body);
  public Frag getResult();
}
```

The semantic analysis phase calls upon `new Translate.Level(···)` in processing a function header. Later it calls other methods of `Translate` to translate the body of the function. Finally the semantic analyzer calls `procEntryExit`, which has the *side effect* of remembering a `ProcFrag`.

All the remembered fragments go into a private fragment list within `Translate`; then `getResult` can be used to extract the fragment list.

CLASSES AND OBJECTS

Figure 7.5 shows a MiniJava class `Vehicle` with two instance variables `position` and `gas`, and two methods `move` and `fill`. We can create multiple `Vehicle` objects. Each `Vehicle` object will have its own `position` and `gas` variables. Two `Vehicle` objects can have different values in their variables, and in MiniJava, only the methods of an object can access the variables of that object. The translation of `new Vehicle()` is much like the translation of record creation and can be done in two steps:

1. Generate code for allocating heap space for all the instance variables; in this case we need to allocate 8 bytes (2 integers, each of size, say, 4).
2. Iterate through the memory locations for those variables and initialize them—in this case, they should both be initialized to 0.

Methods and the *this* pointer. Method calls in MiniJava are similar to function calls; but first, we must determine the class in which the method is declared and look up the method in that class. Second, we need to address the following question. Suppose we have multiple `Vehicle` objects and we want to call a method on one of them; how do we ensure that the implementation knows for which object we are calling the method? The solution is to pass that object as an extra argument to the method; that argument is the *this* pointer. For a method call

```
Vehicle v;
...
v.move();
```

the `Vehicle` object in variable `v` will be the *this* pointer when calling the `move` method.

The translation of method declarations is much like the translation of functions, but we need to avoid name clashes among methods with the same name that are declared in different classes. We can do that by choosing a naming scheme such that the name of the translated method is the concatenation of the class name and the method name. For example, the translation of `move` can be given the name `Vehicle_move`.

Accessing variables. In MiniJava, variables can be accessed from methods in the same class. Variables are accessed via the *this* pointer; thus, the translation of a variable reference is like field selection for records. The position of the variable in the object can be looked up in the symbol table for the class.

PROGRAM TRANSLATION TO TREES

Design a set of visitors which translate a MiniJava program into intermediate representation trees.

Supporting files in `$MINIJAVA/chap7` include:

`Tree/*` Data types for the `Tree` language.
`Tree/Print.java` Functions to display trees for debugging.

A simpler translator. To simplify the implementation of the translator, you may do without the `Ex`, `Nx`, `Cx` constructors. The entire translation can be done with ordinary value expressions. This means that there is only one `Exp` class (without subclasses); this class contains one field of type `Tree.Exp` and only an `unEx()` method. Instead of `Nx`(*s*), use `Ex`(ESEQ(*s*, CONST 0)). For conditionals, instead of a `Cx`, use an expression that just evaluates to 1 or 0.

The intermediate representation trees produced from this kind of naive translation will be bulkier and slower than a "fancy" translation. But they *will* work correctly, and in principle a fancy back-end optimizer might be able to clean up the clumsiness. In any case, a clumsy but correct translator is better than a fancy one that doesn't work.

EXERCISES

7.1 Suppose a certain compiler translates all statements and expressions into Tree.Exp trees, and does not use the `Nx` and `Cx` constructors to represent expressions in different ways. Draw a picture of the IR tree that results from each of the following MiniJava statements and expressions.

a. `a+5`

b. `b[i+1]`

c. `a<b`, which should be implemented by making an `ESEQ` whose left-hand side moves a 1 or 0 into some newly defined temporary, and whose right-hand side is the temporary.

d. `a = x+y;` which should be translated with an `EXP` node at the top.

e. `if (a<b) c=a; else c=b;` translated using the `a<b` tree from part (c) above; the whole statement will therefore be rather clumsy and inefficient.

f. `if (a<b) c=a; else c=b;` translated in a less clumsy way.

7.2 Translate each of these MiniJava statements and expressions into IR trees, but using the `Ex`, `Nx`, and `Cx` constructors as appropriate. In each case, just draw pictures of the trees; an `Ex` tree will be a Tree `Exp`, an `Nx` tree will be a Tree `Stm`, and a `Cx` tree will be a `Stm` with holes labeled *true* and *false* into which labels can later be placed.

a. `a+5;`

b. `b[i+1]=0;`

c. `while (a<0) a=a+1;`

d. `a<b` moves a 1 or 0 into some newly defined temporary, and whose right-hand side is the temporary.

e. `a = x+y;`

f. `if (a<b) c=a; else c=b;`

7.3 Using the C compiler of your choice (or a compiler for another language), translate some functions to assembly language. On Unix this is done with the `-S` option to the C compiler.

Then identify all the components of the calling sequence (items 1–11), and explain what each line of assembly language does (especially the pseudo-instructions that comprise items 1 and 11). Try one small function that returns without much computation (a *leaf* function), and one that calls another function before eventually returning.

7.4 The `Tree` intermediate language has no operators for floating-point variables. Show how the language would look with new binops for floating-point arithmetic, and new relops for floating-point comparisons. You may find it useful to introduce a variant of MEM nodes to describe fetching and storing floating-point values.

***7.5** The `Tree` intermediate language has no provision for data values that are not exactly one word long. The C programming language has signed and unsigned integers of several sizes, with conversion operators among the different sizes. Augment the intermediate language to accommodate several sizes of integers, with conversions among them.

Hint: Do not distinguish signed values from unsigned values in the intermediate trees, but do distinguish between signed operators and unsigned operators. See also Fraser and Hanson [1995], Sections 5.5 and 9.1.