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 I have chosen simple expression trees.

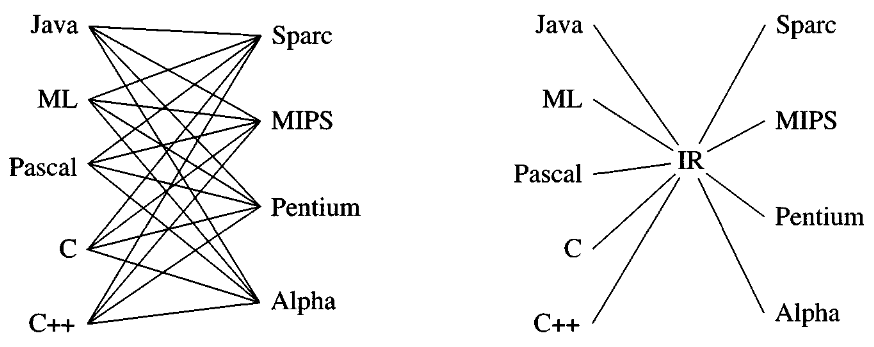


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.

7.1 INTERMEDIATE REPRESENTATION TREES

The intermediate representation tree language is defined by the signature `TREE`, 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.

Here is a description of the meaning of each tree operator. First, the ex-

```
signature TREE =
sig

datatype  exp   = CONST of int
                | NAME of Temp.label
                | TEMP of Temp.temp
                | BINOP of binop * exp * exp
                | MEM of exp
                | CALL of exp * exp list
                | ESEQ of stm * exp

     and  stm   = MOVE of exp * exp
                | EXP of exp
                | JUMP of exp * Temp.label list
                | CJUMP of relop * exp * exp * Temp.label * Temp.label
                | SEQ of stm * stm
                | LABEL of Temp.label

     and binop = PLUS | MINUS | MUL | DIV
                | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR

     and relop = EQ | NE | LT | GT | LE | GE
                | ULT | ULE | UGT | UGE
end
```

FIGURE 7.2. Intermediate representation trees.

pressions (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 Tiger language has no logical operators, 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 Tiger.

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, [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.

KINDS OF EXPRESSIONS

What should the representation of an abstract syntax expression `Absyn.exp` 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 (such as some procedure calls, or **while** expressions in the Tiger language) 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`.

Therefore, we will make a datatype `exp` in the `Translate` module to model these three kinds of expressions:

```
datatype exp = Ex of Tree.exp
             | Nx of Tree.stm
             | Cx of Temp.label * Temp.label -> Tree.stm
```

`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 Tiger expression `a>b|c<d` might translate to the conditional:

```
Cx(fn (t,f) => SEQ(CJUMP(GT,a,b,t,z),
                   SEQ(LABEL z, CJUMP(LT,c,d,t,f))))
```

for some new label z.

Sometimes we will have an expression of one kind and we will need to convert it to an equivalent expression of another kind. For example, the Tiger statement

```
flag := (a>b | c<d)
```

```
structure T = Tree

fun unEx (Ex e) = e
  | unEx (Cx genstm) =
        let val r = Temp.newtemp()
            val t = Temp.newlabel() and f = Temp.newlabel()
         in T.ESEQ(seq[T.MOVE(T.TEMP r, T.CONST 1),
                             genstm(t,f),
                             T.LABEL f,
                             T.MOVE(T.TEMP r, T.CONST 0),
                             T.LABEL t],
                       T.TEMP r)
        end
  | unEx (Nx s) = T.ESEQ(s,T.CONST 0)
```

PROGRAM 7.3. The conversion function unEx.

requires the conversion of a Cx into an Ex so that a 1 (for true) or 0 (for false) can be stored into flag.

It is helpful to have three conversion functions:

```
unEx :  exp → Tree.exp
unNx :  exp → Tree.stm
unCx :  exp → (Temp.label×Temp.label→Tree.stm)
```

Each of these behaves as if it were simply stripping off the corresponding constructor (Ex, Nx, or Cx), but the catch is that each conversion function must work no matter what constructor has been used!

Suppose *e* is the representation of a>b | c<d, so

```
e = Cx(fn(t,f) => ···)
```

Then the assignment statement can be implemented as

MOVE(TEMP$_{flag}$, unEx(e)).

We have "stripped off the Ex constructor" even though Cx was really there instead.

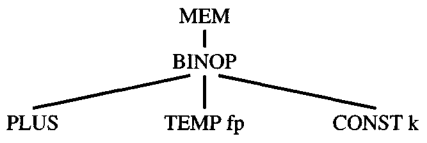
Program 7.3 is the implementation of unEx. To convert a "conditional" into a "value expression," we invent a new temporary r and new labels t and f. Then we make a Tree.stm that moves the value 1 into r, and a conditional jump genstm(t, f) that implements the conditional. If the condition is false, then 0 is moved into r; if 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 functions `unCx` and `unNx` are left as an exercise. It's helpful to have `unCx` treat the cases of CONST 0 and CONST 1 specially, since they have particularly simple and efficient translations. Also, `unCx(Nx _)` need not be translated, as it should never occur in compiling a well typed Tiger program.

SIMPLE VARIABLES

The semantic analysis phase has a function that type-checks a variable in the context of a type environment `tenv` and a value environment `venv`. This function `transVar` returns a record `{exp, ty}` of `Translate.exp` and `Types.ty`. In Chapter 5 the `exp` was merely a place-holder, but now `Semant` must be modified so that each `exp` holds the intermediate-representation translation of each Tiger expression.

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 Tiger compiler we make the simplifying assumption that all variables are the same size – the natural word size of the machine.

Interface between Translate and Semant. The type `Translate.exp` is an abstract data type, whose `Ex` and `Nx` constructors are visible only within `Translate`.

The manipulation of MEM nodes should all be done in the `Translate` module, not in `Semant`. Doing it in `Semant` would clutter up the readability of that module and would make `Semant` dependent on the `Tree` representation.

We add a function

```
val simpleVar : access * level -> exp
```

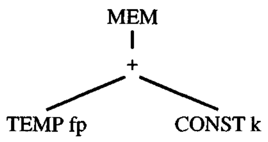
to the `Translate` signature. Now `Semant` can pass the `access` of x (obtained from `Translate.allocLocal`) and the `level` of the function in which x is used and get back a `Translate.exp`.

With this interface, `Semant` never gets its hands dirty with a `Tree.exp` at all. In fact, this is a good rule of thumb in determining the interface between `Semant` and `Translate`: the `Semant` module should not contain any direct reference to the `Tree` or `Frame` module. Any manipulation of IR trees should be done by `Translate`.

The `Frame` structure[1] 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:

```
signature FRAME =
sig
   ⋮
   val FP : Temp.temp
   val wordSize: int
   val exp : access -> Tree.exp -> Tree.exp
end
```

In this and later chapters, I 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)

[1]In a retargetable compiler, one would want the `Translate` module to be completely independent of the choice of target architecture. We do this by putting all machine-dependencies inside `SparcFrame`, or `MipsFrame` (etc.), each of which matches the `FRAME` signature. Then the ML module system allows

```
functor Translate(Frame: FRAME) = struct ... end
structure SparcTranslate = Translate(Sparc)
structure MipsTranslate = Translate(Mips)
```

But for simplicity we could have `Translate` refer directly to a particular frame structure (without using a functor) by including a definition like

```
structure Frame = SparcFrame
```

near the beginning of `Translate`.

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

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

Why bother to pass the tree expression `TEMP(Frame.FP)` as an argument? The answer is that the address of the frame is the same as the current frame pointer *only* when accessing the variable from its own level. When accessing a from an inner-nested function, the frame address must be calculated using static links, and the result of this calculation will be the `Tree.exp` argument to `Frame.exp`.

If a is a register access such as `InReg`(t_{832}), then the frame-address argument to `Frame.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 that can appear only on the right side of an assignment. Fortunately, only MEM and TEMP nodes can appear on the left of a MOVE node.

FOLLOWING STATIC LINKS

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

To construct this expression, we need the `level` l_f of the function f in which x is used, and the `level` l_g of the function g in which x is declared. As we strip levels from l_f, we use the static link offsets $k_1, k_2, \ldots$ from these levels to construct the tree. Eventually we reach l_g, and we can stop.

How can we tell when we have reached l_g? The `level` type described on page 141 cannot easily be tested for equality. So we add a `unit ref` to the `level` data structure to allow testing for identity of levels.

`Translate.simpleVar` must produce a chain of MEM and + nodes to fetch static links for all frames between the level of use (the `level` passed to `simpleVar`) and the level of definition (the `level` within the variable's `access`).

ARRAY VARIABLES

For the rest of this chapter I will not specify all the interface functions of `Translate`, as I 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 Tiger (as in Java and ML), array variables behave like pointers. Tiger has no named array constants as in C, however. Instead, new array values are created (and initialized) by the construct $t_a[n]$ `of` i, where t_a is the name of an array type, n is the number of elements, and i is the initial value of each element. In the program

```
let
 type intArray = array of int
 var a := intArray[12] of 0
 var b := intArray[12] of 7
in a := b
end
```

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

Tiger record values are also pointers. Record assigment, like array assignment, is pointer assigment and does not copy all the fields. This is also true of modern 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 assigment 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 Tiger are scalar. Even a Tiger array or record variable is really a pointer (a kind of scalar); the *Tiger Language Reference Manual* does not say so explicitly, because it is talking about Tiger semantics instead of Tiger 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:

```
signature TREE =
sig
    type size = int
    datatype exp = ···
                 | MEM of exp * size
                    ⋮
end
```

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).

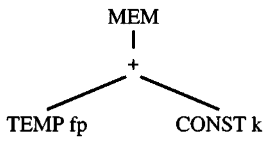
Leaving out the size on MEM nodes makes the Tiger compiler easier to implement, but limits the generality of its intermediate representation.

SUBSCRIPTING AND FIELD SELECTION

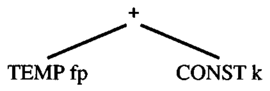
To subscript an array in Pascal or C (to compute $a[i]$), just calculate the address of the ith element of a: $(i-l) \times s + a$, where l is the lower bound of the index range, s is the size (in bytes) of each array element, and a is the base address of the array elements. If a is global, with a compile-time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

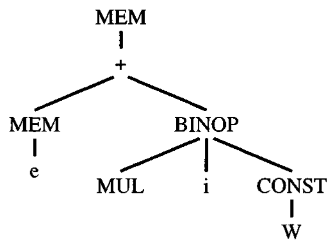


What could happen to this l-value?

- 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 Tiger language, there are no structured, or "large," l-values. This is because all record and array values are really pointers to record 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 $p, p+W, p+2W, \ldots$ (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.[2]

ARITHMETIC

Integer arithmetic is easy to translate: each `Absyn` 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. Some numerical programs rely on identities such as $-0 < 0$. Thus, the `Tree` language does not support unary negation very well.

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

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 `Absyn` 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 Tiger operators `&` and `|`. 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.

The `&` and `|` operators of the Tiger language, which combine conditionals with short-circuit conjunction and disjunction (*and* and *or*) respectively, have

[2]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.

already been translated into `if`-expressions in the abstract syntax.

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, apply `unCx` to e_1 and `unEx` to 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.

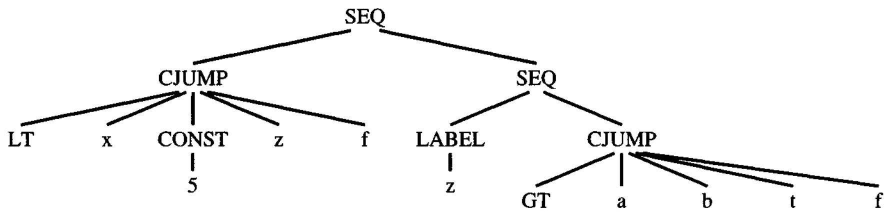
This will produce perfectly correct results. However, the translated code may not be very efficient at all. If e_2 and e_3 are both "statements" (expressions that return no value), then their representation is likely to be `Nx`, not `Ex`. Applying `unEx` to them will work – a coercion will automatically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



SEQ(s_1(z, f), SEQ(LABEL z, s_2(t, f)))

for some new label z.

String comparison. Because the string equality operator is complicated (it must loop through the bytes checking byte-for-byte equality), the compiler should call a runtime-system function `stringEqual` that implements it. This function returns a 0 or 1 value (false or true), so the CALL tree is naturally contained within an `Ex` expression. String not-equals can be implemented by generating `Tree` code that complements the result of the function call.

STRINGS

A string literal in the Tiger (or C) language is 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`, the `Translate` module makes a new label `lab`, and returns the tree `Tree.NAME(lab)`. It also puts 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 169; translation of string fragments to assembly language, on page 262.

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. I say "almost" because string literals must be represented.

But how are strings represented in Tiger? In Pascal, they 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.

Tiger strings should be able to contain arbitrary 8-bit codes (including zero). A simple representation that serves well is to have a string pointer point to a one-word integer containing the length (number of characters), followed immediately by the characters themselves. Then the `string` function in the machine-specific `Frame` module (`MipsFrame`, `SparcFrame`, `Pentium-Frame`, etc.) can make a string with a label definition, an assembly-language pseudo-instruction to make a word containing the integer length, and a pseudo-instruction to emit character data.

RECORD AND ARRAY CREATION

The Tiger language construct $a\{f_1 = e_1, f_2 = e_2, ..., f_n = e_n\}$ creates an n-element record initialized to the values of expressions e_i. Such a record may

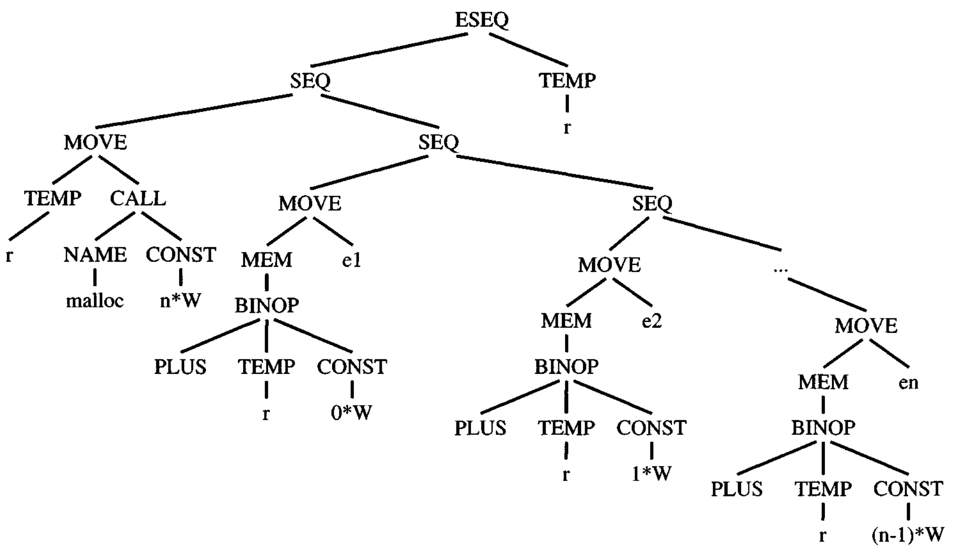


FIGURE 7.4. Record allocation.

outlive the procedure activation that creates it, so it cannot be allocated on the stack. Instead, it must be allocated on the *heap*. There is no provision for freeing records (or strings); industrial-strength Tiger 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.

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

CALL(NAME(`Temp.namedlabel("initArray")`), $[a, b]$)

This refers to an external function `initArray` which is written in a language such as C or assembly language – it cannot be written in Tiger because Tiger 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 Tiger 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:

```
signature FRAME =
sig
        :
    val externalCall: string * Tree.exp list -> Tree.exp
end
```

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

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

```
fun externalCall(s,args) =
     T.CALL(T.NAME(Temp.namedlabel s), args)
```

but may have to be adjusted for static links, or underscores in labels, and so on.

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*.

So that `transExp` can translate **break** statements, it will have a new formal parameter `break` that is the *done* label of the nearest enclosing loop. In translating a **while** loop, `transExp` is called upon *body* with the *done* label passed as the `break` parameter. When `transExp` is recursively calling itself in nonloop contexts, it can simply pass down the same `break` parameter that was passed to it.

The `break` argument must also be added to the `transDec` function.

FOR LOOPS

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

```
for i := lo to hi          let var i := lo
  do body                      var limit := hi
                           in while i <= limit
                                 do (body; i := i+1)
                           end
```

A very straightforward approach to the translation of **for** statements is to rewrite the *abstract syntax* into the abstract syntax of the **let/while** expression shown, and then call `transExp` on 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, except 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

The clause to type-check **let** expressions was shown on page 117. It is not hard to augment this clause to translate into `Tree` expressions. `TransExp` and `transDec` now take more arguments than before (as described elsewhere in this chapter), and `transDec` must now return an extra result – the `Translate.exp` resulting from the evaluation of the declaration (this will be explained below).

The call to `transDec` will not only return a result record (containing a new type environment, value environment, and `Translate.exp`) but also will have side effects: for each variable declaration within the declaration, additional space will be reserved in the current level's `frame`. Also, for each function declaration, a new "fragment" of `Tree` code will be kept for the function's body.

VARIABLE DEFINITION

The `transDec` function, described in Chapter 5, returns an augmented value environment and an augmented type environment that are used in processing the body of a **let** expression.

However, the initialization of a variable translates into a `Tree` expression that must be put just before the body of the **let**. Therefore, `transDec` must also return an `exp list` of assignment expressions that accomplish these initializations.

If `transDec` is applied to function and type declarations, the `exp list` will be empty.

FUNCTION DEFINITION

Each Tiger function is translated into a segment of assembly language with a *prologue*, a *body*, and an *epilogue*. The body of a Tiger 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 – including the static link – 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 261). 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:

```
signature FRAME =
sig
        ⋮
    val RV : Temp.temp      (* as seen by callee *)
end
```

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 page 136. They should be done by a function in the `Frame` module:

```
signature FRAME =
sig
     ⋮
    val procEntryExit1 : frame * Tree.stm -> Tree.stm
end
```

The implementation of this function will be discussed on page 261. `Translate` should apply it to each procedure body (items 5–7) as it is translated.

FRAGMENTS

Given a Tiger function definition comprising a `level` and 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:

```
signature FRAME =
sig
    ⋮
   datatype frag = PROC of  {body: Tree.stm, frame: frame}
                 | STRING of Temp.label * string
end

signature TRANSLATE =
sig
    ⋮
   val procEntryExit : {level: level, body: exp} -> unit

   structure Frame : FRAME
   val getResult : unit -> Frame.frag list
end
```

The semantic analysis phase calls upon `Translate.newLevel` in processing a function header. Later it calls other interface fields of `Translate` to translate the body of the Tiger function; this has the side effect of remembering `STRING` fragments for any string literals encountered (see pages 163 and 262). Finally the semantic analyzer calls `procEntryExit`, which has the *side effect* of remembering a `PROC` fragment.

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

PROGRAM TRANSLATION TO TREES

Design the `TRANSLATE` signature, implement the `Translate` structure, and rewrite the `Semant` structure to call upon `Translate` appropriately. The result of calling `Semant.transProg` should be a `Translate.frag list`.

To keep things simpler (for now), keep all local variables in the frame; do not bother with `FindEscape`, and assume that every variable escapes.

In the `Frame` module, a "dummy" implementation

```
fun procEntryExit1(frame,body) = body
```

is suitable for preliminary testing of `Translate`.

Supporting files in `$TIGER/chap7` include:

`tree.sml`	Data types for the `Tree` language.
`printtree.sml`	Functions to display trees for debugging.

and other files as before.

A simpler Translate. To simplify the implementation of `Translate`, you may do without the `Ex`, `Nx`, `Cx` constructors. The entire `Translate` module can be done with ordinary value-expressions. This makes `Translate.exp` type identical to `Tree.exp`. That is, instead of `Ex`(e), just use e. Instead of `Nx`(s), use the expression `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 `Translate` module is better than a fancy one that doesn't work.

EXERCISES

7.1 Suppose a certain compiler translates all expressions and subexpressions 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 expressions. Assume all variables are nonescaping unless specified otherwise.

a. `a+5`

b. `b[i+1]`

c. `p.z.x`, where `p` is a Tiger variable whose type is

```
type m = {x:int,y:int,z:m}
```

d. `write(" ")`, as it appears on line 13 of Program 6.3.

e. `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.

f. `if a then b else c`, where `a` is an integer variable (true if $\neq 0$); this should also be translated using an ESEQ.

g. `a := x+y`, which should be translated with an EXP node at the top.

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

i. `if a<b then c:=a else c:=b`, translated in a less clumsy way.

7.2 Translate each of these 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. `output := concat(output,s)`, as it appears on line 8 of Program 6.3. The `concat` function is part of the standard library (see page 519), and for purposes of computing its static link, assume it is at the same level of nesting as the `prettyprint` function.

c. `b[i+1]:=0`

d. `(c:=a+1; c*c)`

e. `while a>0 do a := a-1`

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

g. `if a then b else c`, where `a` is an integer variable (true if $\neq 0$).

h. `a := x+y`

i. `if a<b then a else b`

j. `if a<b then 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.