Chapter 6
Intermediate-Code Generation

The art of free society consists first in the maintenance of the symbolic code; and secondly in fearlessness of revision, to secure that the code serves those purposes which satisfy an enlightened reason.

Alfred North Whitehead (1869–1947)

The ultimate goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code that can run on the computer. This does not mean that we need to translate directly from the high-level abstract syntax to machine code. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called *intermediate code*.

Apart from structuring the compiler into smaller jobs, using an intermediate language has other advantages:

- If the compiler needs to generate code for several different machine architectures, only one translation to intermediate code is needed. Only the translation from intermediate code to machine language (i.e., the *back-end*) needs to be written in several versions.
- If several high-level languages need to be compiled, only the translation to intermediate code need to be written for each language. They can all share the back-end, i.e., the translation from intermediate code to machine code.
- Instead of translating the intermediate language to machine code, it can be *interpreted* by a small program written in machine code or in a language for which a compiler or interpreter already exists.

T. Æ. Mogensen, *Introduction to Compiler Design*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-46460-7_6

- A lot of optimisations are more easily done at the intermediate-code level than at the source-code level or the machine-code level. And they can be shared by all compilers that use the intermediate code.

The advantage of using an intermediate language is most obvious if many languages are to be compiled to many machines. If translation is done directly, the number of compilers is equal to the product of the number of languages and the number of machines. If a common intermediate language is used, one front-end (i.e., compiler to intermediate code) is needed for every language and one back-end (interpreter or code generator) is needed for each machine, making the total number of front-ends and back-ends equal to the sum of the number of languages and the number of machines. Additionally, each of these is typically simpler than a compiler that compiles directly to machine code.

If an interpreter for an intermediate language is written in a language for which there already exist implementations for the target machines, the same interpreter can be interpreted or compiled for each machine. This way, there is no need to write a separate back-end for each machine. The advantages of this approach are:

- No actual back-end needs to be written for each new machine, as long as the machine is equipped with an interpreter or compiler for the implementation language of the interpreter for the intermediate language.
- A compiled program can be distributed in a single intermediate form for all machines, as opposed to shipping separate binaries for each machine.
- The intermediate form may be more compact than machine code. This saves space both in distribution and on the machine that executes the programs (though the latter is somewhat offset by requiring the interpreter to be kept in memory during execution).

The disadvantage is speed: Interpreting the intermediate form will in most cases be (sometimes a lot) slower than executing translated code directly. Nevertheless, the approach has seen some success, e.g., in early implementations of Java and in the Moscow ML compiler for Stadard ML.

Some of the speed penalty can be eliminated by translating the intermediate code to machine code immediately before or during execution of the program. This hybrid form is called *just-in-time compilation* and is often used in modern implementations of Java for executing the intermediate code (the Java Virtual Machine).

We will in this book, however, focus mainly on using the intermediate code for traditional compilation, where the intermediate form will be translated to machine code by the back-end of the compiler.

6.1 Designing an Intermediate Language

An intermediate language should, ideally, have the following properties:

- It should be easy to translate high-level languages to the intermediate language. This should be the case for a wide range of different source languages.
- It should be easy to translate the intermediate language to machine code. This should be true for a wide range of different target architectures.
- The intermediate format should be suitable for optimisations.

The first two of these properties can be somewhat hard to reconcile. A language that is intended as target for translation from a high-level language should be fairly high level itself. However, this may be hard to achieve for more than a small number of similar high-level languages. Furthermore, a high-level intermediate language puts more burden on the back-ends. A low-level intermediate language may make it easy to write back-ends, but puts more burden on the front-ends. A low-level intermediate language, also, may not fit all machines equally well, though this is usually less of a problem than the similar problem for front-ends, as machine languages typically are more similar than high-level languages.

A solution that may reduce the translation burden, though it does not address the other problems, is to have two intermediate levels: One, which is fairly high-level, is used for the front-ends and the other, which is fairly low-level, is used for the back-ends. A single shared translator is then used to translate between these two intermediate formats.

When the intermediate format is shared between many compilers, it makes sense to do as many optimisations as possible on the intermediate format. This way, the (often substantial) effort of writing good optimisations is done only once instead of in every compiler.

Another thing to consider when choosing an intermediate language is the "granularity": Should an operation in the intermediate language correspond to a large amount of work or to a small amount of work?

The first of these approaches is often used when the intermediate language is interpreted, as the overhead of decoding instructions is amortised over more actual work, but it can also be used for compiling. In this case, each intermediate-code instruction is typically translated into a sequence of machine-code instructions. When coarse-grained intermediate code is used, there is typically a fairly large number of different intermediate-code instructions.

The opposite approach is to let each intermediate-code instruction be as simple as possible. This means that each intermediate-code instruction is typically translated into a single machine-code instruction or that several intermediate-code instructions can be combined into one machine-code instruction. The latter can, to some degree, be automated as each machine-code instruction can be described as a sequence of intermediate-code instructions. When intermediate-code is translated to machine-code, the code generator can look for sequences that match machine-code instructions. By assigning cost to each machine-code instruction, this can be turned into a combinatorial optimisation problem, where the least-cost solution is found. We will return to this in Chap. 7.

6.2 The Intermediate Language

For this chapter, we have chosen a fairly low-level fine-grained intermediate language, as it is best suited to convey the techniques we want to cover.

We will not treat translation of function calls until Chap. 9, so a "program" in our intermediate language will, for the time being, keep function definitions, calls and returns as primitive constructions in the intermediate language. In Chap. 9, we will see how these can be translated into lower-level code.

The grammar for the intermediate language is shown in Grammar 6.1.

A program is a sequence of function definitions, each of which consists of a header and a body. The header defines the name of the function and its arguments, and the body is a list of instructions. The instructions are:

- A label. This has no effect but serves only to mark the position in the program as a target for jumps.
- An assignment of an atomic expression (constant or variable) to a variable.

Grammar 6.1 The intermediate language

$$\begin{array}{lll}
\mathit{Program} & \rightarrow & \mathit{Functions} \\
\\
\mathit{Functions} & \rightarrow & \mathit{Function}\ \mathit{Functions} \\
\mathit{Functions} & \rightarrow & \mathit{Function} \\
\\
\mathit{Function} & \rightarrow & \mathit{Header}\ \mathit{Body} \\
\\
\mathit{Header} & \rightarrow & \mathbf{functionid}(\mathit{Args}) \\
\\
\mathit{Body} & \rightarrow & [\ \mathit{Instructions}\ ] \\
\\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
\\
\mathit{Instruction} & \rightarrow & \mathtt{LABEL}\ \mathbf{labelid} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{id}\ \mathbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \mathbf{id} \\
\mathit{Instruction} & \rightarrow & \mathtt{GOTO}\ \mathbf{labelid} \\
\mathit{Instruction} & \rightarrow & \mathtt{IF}\ \mathbf{id}\ \mathbf{relop}\ \mathit{Atom}\ \mathtt{THEN}\ \mathbf{labelid}\ \mathtt{ELSE}\ \mathbf{labelid} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathtt{CALL}\ \mathbf{functionid}(\mathit{Args}) \\
\mathit{Instruction} & \rightarrow & \mathtt{RETURN}\ \mathbf{id} \\
\\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
\mathit{Atom} & \rightarrow & \mathbf{num} \\
\\
\mathit{Args} & \rightarrow & \mathbf{id} \\
\mathit{Args} & \rightarrow & \mathbf{id}\ ,\ \mathit{Args}
\end{array}$$

- A unary operator applied to an atomic expression, with the result stored in a variable.
- A binary operator applied to a variable and an atomic expression, with the result stored in a variable.
- A transfer from memory to a variable. The memory location is an atomic expression.
- A transfer from a variable to memory. The memory location is an atomic expression.
- A jump to a label.
- A conditional selection between jumps to two labels. The condition is found by comparing a variable with an atomic expression by using a relational operator ($=$, $\neq$, $<$, $>$, $\leq$ or $\geq$).
- A function call. The arguments to the function call are variables and the result is assigned to a variable. This instruction is used even if there is no actual result (i.e., if a procedure is called instead of a function), in which case the result variable is a dummy variable.
- A return statement. This returns the value of the specified variable as the result of the current function. We require that a function will always exit by executing a `RETURN` instruction, i.e, not by "falling out" of the last instruction in its body. In practice, this means that the last instruction in a function body must be either a `RETURN` instruction, a `GOTO` instruction or an `IF-THEN-ELSE` instruction, the branches of which are also in these three categories.

An atomic expression is either a variable or a constant.

We have not specified the set of unary and binary operations, but we expect these to include normal integer arithmetic and bitwise logical operations.

Variables (including function parameters) are local to the function definition in which they are used, and they do not have to be declared in advance. We assume that all values are integers. Adding floating-point numbers and other primitive types (such as Booleans, characters or pointers) is not difficult, though: Variables and operators should be given types. If this is done, memory accesses should use pointer, and there should be explicit operators for converting between the different types. To keep things simple, we stick to integers, which are assumed to be of a size bounded by one machine word (typically 64 bits).

A label is local to the function in which it occurs. There can not be two `LABEL` instructions with the same label name in the same function definition.

Jumps are local to the function definition in which they occur, so you can not jump into or out of a function definition. If there is a jump to a label l in a function definition, there must also be a `LABEL` l instruction in the same function definition.

6.3 Syntax-Directed Translation

We will generate code using translation functions for each syntactic category, similar to the functions we used for interpretation and type checking. We generate code for a syntactic construct independently of the constructs around it, except that the parameters of a translation function may hold information about the context (such as symbol tables) and the result of a translation function may (in addition to the generated code) hold information about how the generated code interfaces with its context (such as which variables it uses). Since the translation closely follows the syntactic structure of the program, it is called *syntax-directed translation*.

Given that translation of a syntactic construct is mostly independent of the surrounding and enclosed syntactic constructs, we might miss opportunities to exploit synergies between these and, hence, generate less than optimal code. We will try to remedy this in later chapters by using various optimisation techniques.

6.4 Generating Code from Expressions

Grammar 6.2 shows a simple language of expressions, which we will use as our initial example for translation. Again, we have let the set of unary and binary operators be unspecified but assume that the intermediate language includes all those used by the expression language. We assume that there is a function *transop* that translates the name of an operator in the expression language into the name of the corresponding operator in the intermediate language. The tokens **unop** and **binop** have the names of the actual operators as attributes, accessed by the function *getopname*.

When writing a compiler, we must decide what needs to be done at compile-time and what needs to be done at run-time. Ideally, as much as possible should be done at compile-time, but some things need to be postponed until run-time, as they need access to the actual values of variables, etc., which are not known at compile-time. When we, below, explain the workings of the translation functions, we might use phrasing like "the expression is evaluated and the result stored in the variable". This describes actions that are performed at run-time by the code that is generated at compile-time. At times, the textual description may not be 100% clear as to what

Grammar 6.2 A simple expression language

$$
\begin{array}{lcl}
Exp & \rightarrow & \mathbf{num} \\
Exp & \rightarrow & \mathbf{id} \\
Exp & \rightarrow & \mathbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \mathbf{binop}\ Exp \\
Exp & \rightarrow & \mathbf{id}(Exps) \\
 & & \\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

happens at which time, but the notation used in the translation functions makes this clear: Intermediate-language code (or equivalent machine language) is executed at run-time, the rest is done at compile time. Intermediate-language instructions may refer to values (constants and variable names) that are generated at compile time. When instructions have operands that are written in *italics*, these operands are variables in the compiler that contain compile-time values that are inserted into the generated code. For example, if *place* holds the variable name `t14` and *v* holds the value 42, then the code template [*place* := *v*] will generate the code `[t14 := 42]`.

When we want to translate the expression language to the intermediate language, the main complication is that the expression language is tree-structured while the intermediate language is flat, requiring the result of every operation to be stored in a variable and every (non-constant) argument to be fetched from a variable. We use a function *newvar* at compile time to generate new intermediate-language variable names. Whenever *newvar* is called, it returns a previously unused variable name, so it is not a function in the mathematical sense (as a mathematical function would return the same value every time).

We will describe translation of expressions by a translation function using a notation similar to the notation we used for type-checking functions in Chap. 5.

Some attributes for the translation function are obvious: The translation function must return the code as a synthesised attribute. Furthermore, it must translate the names of variables and functions used in the expression language to the names these correspond to in the intermediate language. This can be done by symbol tables *vtable* and *ftable* that bind variable and function names in the expression language into the corresponding names in the intermediate language. The symbol tables are passed as inherited attributes to the translation function. In addition to these attributes, the translation function must use attributes to determine where it should put the values of subexpressions. This can be done in two ways:

(1) The locations (variables) of the values of a subexpression can be passed up as a synthesised attribute to the parent expression, which decides on a location for its own value and returns this as synthesised attribute.
(2) The parent expression can determine in which variables it wants to find the values of its subexpressions, and pass this information down to the subexpressions as inherited attributes.

In both cases, new variables will usually be generated for the intermediate values, though it can be tempting to reuse variables.

When generating code for a variable expression by method 1, we might want to simply pass the (intermediate-code version of) that variable up as the location of the value of the subexpression. This, however, only works under the assumption that the variable is not updated before the value is used by the parent expression. If expressions can have side effects, this is not always the case, as the C expression "`x+(x=3)`" shows. If `x` has the value 5 prior to evaluation, the intended result is 8. But if the addition just uses the location of `x` as both arguments for the addition, it will return 6. We do not have assignments in our expression language, but to prepare for later extensions, it is best to copy the value of a variable into a new variable whenever it is used.

Method 2 will have a similar problem if we add assignments: When generating code for an assignment (using C-like notation) "x = e", where e is an expression, it could be tempting to just pass (the intermediate-code version of) x down as the location where to store the value of e. This will work only under the assumption that the code for e does not store anything in its given location until the end of the evaluation of e. While this may seem like a reasonable assumption, it is better to be safe and generate a new variable and only copy the value of this to x when the assignment is made. Generation of new variables is done at compile time, so it does not cost anything at run time. Copying values from one variable to another may cost something at run time, but as we shall see in Chaps. 7, 8, and 10, most of the copying can be eliminated in later phases.

If new variables are generated for all intermediate values, both methods will give the same code (up to renaming of variables). Notationally, method 2 is slightly less cumbersome, so we will use this for our translation function $Trans_{Exp}$, which is shown in Fig. 6.3. We will, however, use method 1 for the translation function $Trans_{Exps}$, that generates code for a list of expressions, partly to illustrate both styles, but also because it is slightly more convenient to use method 1 for $Trans_{Exps}$.

In $Trans_{Exp}$, the inherited attribute *place* is the intermediate-language variable that the result of the expression must be stored in.

If the expression is just a number, the value of that number is stored in the *place*.

If the expression is a variable, the intermediate-language equivalent of this variable is found in *vtable*, and an assignment copies it into the intended *place*.

A unary operation is translated by first generating a new intermediate-language variable to hold the value of the argument of the operation. Then the argument is translated using the newly generated variable for its *place* attribute. We then use an **unop** operation in the intermediate language to assign the result to the inherited *place*. The operator $++$ concatenates two lists of instructions.

A binary operation is translated in a similar way. Two new intermediate-language variables are generated to hold the values of the arguments, then the arguments are translated, and finally a binary operation in the intermediate language assigns the final result to the inherited *place*.

A function call is translated by first translating the arguments, using the auxiliary function $Trans_{Exps}$. Then a function call is generated using the argument variables returned by $Trans_{Exps}$, with the result assigned to the inherited *place*. The name of the function is looked-up in *ftable* to find the corresponding intermediate-language name.

$Trans_{Exps}$ generates code for each argument expression, storing the results into new variables. These variables are returned along with the code, so they can be put into the argument list of the call instruction.

Fig. 6.3 Translating an expression

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`

num	$v = getvalue(\mathbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\mathbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\mathbf{unop}))$ $code_1 ++ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\mathbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\mathbf{id}))$ $code_1 ++ [place :=$ `CALL` $fname(a_1, \ldots, a_n)]$

$Trans_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`

Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
Exp , $Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 ++ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

6.4.1 Examples of Expression Translation

Translation of expressions is always relative to symbol tables and a place for storing the result. In the examples below, we assume a variable symbol table that binds `x`, `y` and `z` to `v0`, `v1` and `v2`, respectively, and a function table that binds `f` to `_f`. The place for the result is `t0`, and we assume that calls to $newvar()$ return, in sequence, the variable names `t1`, `t2`, `t3`, ...

We start by the simple expression `x-3`. This is a binop-expression, so we first call $newvar()$ twice, giving $place_1$ the value `t1` and $place_2$ the value `t2`. We then call $Trans_{Exp}$ recursively with the expression `x` and $place_1$ (which is equal to `t1`) as the intended location of the result. When translating this, we first look up `x` in the variable symbol table, yielding `v0`, and then return the code [`t1` := `v0`]. Back in the translation of the subtraction expression, we assign this code to $code_1$ and once more call $Trans_{Exp}$ recursively, this time with the expression `3`. This is translated to the code [`t2` := `3`], which we assign to $code_2$. The final result is produced by $code_1$++$code_2$++[`t0 := t1-t2`] which yields [`t1 := v0, t2 := 3, t0 := t1-t2`]. The source-language operator - is by *transop* translated to the intermediate-language operator `-`.

The resulting code looks quite sub-optimal, and could, indeed, be shortened to [`t0 := v0-3`]. When we generate intermediate code, we want, for simplicity, to treat each subexpression independently of its context. This may lead to such superfluous assignments. We will look at ways of getting rid of these when we treat machine code generation, register allocation, and data-flow analysis in Chaps. 7, 8 and 10.

A more complex expression is `3+f(x-y,z)`. Using the same assumptions as above, this yields the code

```
t1 := 3
  t4 := v0
  t5 := v1
 t3 := t4 - t5
 t6 := v2
t2 := CALL _f(t3,t6)
t0 := t1 + t2
```

We have, for readability, laid the code out on separate lines rather than using a comma-separated list. The indentation indicates the depth of calls to $Trans_{Exp}$ that produced the code in each line.

Suggested exercises: 6.1.

6.5 Translating Statements

We now extend the expression language in Grammar 6.2 with statements. The extensions are shown in Grammar 6.4. Note that we use `:=` for assignment to distinguish assignment from equality comparison. The grammar is ambiguous, but we work on abstract syntax, where the ambiguities have been resolved.

When translating statements, we will need the symbol table for variables (for translating assignment), and since statements contain expressions, we also need *ftable* so we can pass both symbol tables on to $Trans_{Exp}$.

Grammar 6.4 Statement language

Stat	→	*Stat* ; *Stat*
Stat	→	**id** := *Exp*
Stat	→	`if` *Cond* `then` *Stat*
Stat	→	`if` *Cond* `then` *Stat* `else` *Stat*
Stat	→	`while` *Cond* `do` *Stat*
Stat	→	`repeat` *Stat* `until` *Cond*
Cond	→	*Exp* **relop** *Exp*

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`

$Stat_1$; $Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 {+}{+} code_2$
id := Exp	$place = newvar()$ $x = lookup(vtable, getname(\mathbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place) {+}{+} [x := place]$
`if` $Cond$ `then` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_1 {+}{+} [\texttt{LABEL}\ label_1] {+}{+} code_2$ $ {+}{+} [\texttt{LABEL}\ label_2]$
`if` $Cond$ `then` $Stat_1$ `else` $Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 {+}{+} [\texttt{LABEL}\ label_1] {+}{+} code_2$ $ {+}{+} [\texttt{GOTO}\ label_3,\ \texttt{LABEL}\ label_2]$  $ {+}{+} code_3 {+}{+} [\texttt{LABEL}\ label_3]$
`while` $Cond$ `do` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] {+}{+} code_1$ $ {+}{+} [\texttt{LABEL}\ label_2] {+}{+} code_2$ $ {+}{+} [\texttt{GOTO}\ label_1,\ \texttt{LABEL}\ label_3]$
`repeat` $Stat_1$ `until` $Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] {+}{+} code_1$ $ {+}{+} code_2 {+}{+} [\texttt{LABEL}\ label_2]$

Fig. 6.5 Translation of statements

Just like we use *newvar* to generate new, not previously used, variables, we use a similar function *newlabel* to generate new labels. The translation function for statements is shown in Fig. 6.5. It uses an auxiliary translation function shown in Fig. 6.6 for translating conditions.

A sequence of two statements is translated by putting the code for these in sequence.

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1$++$code_2$++[`IF` t_1 op t_2 `THEN` $label_t$ `ELSE` $label_f$]

Fig. 6.6 Translation of simple conditions

An assignment is translated by translating the right-hand-side expression to code that places the result in a new variable, and then copying this to the left-hand-side variable.

When translating statements that use conditions, we use an auxiliary function $Trans_{Cond}$. $Trans_{Cond}$ translates the arguments to the condition and generates an `IF-THEN-ELSE` instruction using the same relational operator as the condition (but translated to an intermediate-code operator). The target labels of this instruction are inherited attributes to $Trans_{Cond}$.

An `if-then` statement is translated by first generating two labels: One for the `then`-branch, and one for the code following the `if-then` statement. The condition is translated by $Trans_{Cond}$, which is given the two labels as attributes. When (at run-time) the condition is true, the first of these are selected, and when false, the second is chosen. Hence, when the condition is true, the `then`-branch is executed followed by the code after the `if-then` statement. When the condition is false, we jump directly to the code following the `if-then` statement, hence bypassing the `then`-branch.

An `if-then-else` statement is treated similarly, but now the condition must choose between jumping to the `then`-branch or the `else`-branch. At the end of the `then`-branch, a jump bypasses the code for the `else`-branch by jumping to the label at the end. Hence, there is need for three labels: One for the `then`-branch, one for the `else`-branch and one for the code following the `if-then-else` statement.

If the condition in a `while-do` loop is true, the body must be executed, otherwise the body is by-passed and the code after the loop is executed. Hence, the condition is translated with attributes that provide the label for the start of the body and the label for the code after the loop. When the body of the loop has been executed, the condition must be re-tested for further passes through the loop. Hence, a jump is made to the start of the code for the condition. A total of three labels are thus required: One for the start of the loop, one for the loop body and one for the end of the loop.

A `repeat-until` loop is slightly simpler. The body precedes the condition, so there is always at least one pass through the loop. If the condition is true, the loop is terminated and we continue with the code after the loop. If the condition is false, we jump to the start of the loop. Hence, only two labels are needed: One for the start of the loop and one for the code after the loop.

Fig. 6.7 Example statements

```
f := 1;
n := 7;
repeat
  f := f * n;
  n := n - 1
until n = 0
```

```
 t1 := 1
v0 := t1
 t2 := 7
v1 := t2
LABEL L1
  t3 := v0
  t4 := v1
 v0 := t3 * t4
  t5 := v1
  t6 := 1
 v1 := t5 - t6
  t7 := v1
  t8 := 0
 IF t7 = v1 THEN L2 ELSE L1
LABEL L2
```

Fig. 6.8 Example statement translation

6.5.1 Example of Statement Translation

Using the translation rules in Figs. 6.5 and 6.6, the statements in Fig. 6.7 get translated into the code shown in Fig. 6.8, using a symbol table that binds `f` to `v0` and `n` to `v1`.

Suggested exercises: 6.2.

6.6 Logical Operators

Logical conjunction, disjunction and negation are often available for conditions, so we can write, e.g., ($x = y$ **or** $y = z$), where **or** is a logical disjunction operator. There are typically two ways to treat logical operators in programming languages:

(1) Logical operators are similar to arithmetic operators: The arguments are evaluated, and the operator is applied to find the result.
(2) The second operand of a logical operator is only evaluated if the first operand is insufficient to determine the result. This means that a logical **and** will only evaluate its second operand if its first argument evaluates to **true**, and a logical

or will only evaluate its second operand if its first argument is **false**. This variant is called *sequential logical operators*.

The C language has both variants. The arithmetic logical operators are called `&` and `|`, and the sequential variants are called `&&` and `||`.

The first variant is typically implemented by using bitwise logical operators and uses 0 to represent **false** and some nonzero value (typically 1 or -1) to represent **true**. In C, there is no separate Boolean type, so integers are used even at the source-code level to represent truth values. While any nonzero integer is treated as logical truth by conditional statements, comparison operators return 1, and bitwise logical operators `&` (bitwise **and**) and `|` (bitwise **or**) are used to implement the corresponding logical operations, so 1 is the preferred representation of logical truth. Logical negation is *not* handled by bitwise negation, as the bitwise negation of 1 is not 0. Instead, a special logical negation operator `!` is used that maps any non-zero value to 0, and 0 to 1. We assume an equivalent operator is available in the intermediate language. Some languages use -1 to represent logical truth, as all bits in this value are 1 (assuming two's complement representation is used, which is normally the case). This makes bitwise negation a valid implementation of logical negation.

Adding non-sequential logical operators to our language is not too difficult if we simply assume that the intermediate language includes the required relational operators, bitwise logical operations, and logical negation. We can now simply allow any expression to be used as a condition by adding the production

$$Cond \rightarrow Exp$$

to Grammar 6.4. If there is a separate Boolean type, we assume that a type checker has verified that the expression is of Boolean type. If, as in C, there is no separate Boolean type, the expression must be of integer type.

We then extend the translation function for conditions as follows:

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`

Exp_1 **relop** Exp_2	$t_1 = newvar()$
	$t_2 = newvar()$
	$code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$
	$code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$
	$op = transop(getopname(\textbf{relop}))$
	$code_1$++$code_2$++[`IF` t_1 op t_2 `THEN` $label_t$ `ELSE` $label_f$]
Exp	$t = newvar()$
	$code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$
	$code_1$++[`IF` $t \neq 0$ `THEN` $label_t$ `ELSE` $label_f$]

We need to convert the numerical value returned by $Trans_{Exp}$ into a choice between two labels, so we generate an `IF` instruction that does just that.

The rule for relational operators is now actually superfluous, as the case it handles is covered by the second rule (since relational operators are assumed to be included in the set of binary arithmetic operators in the intermediate language). However, we can consider it an optimisation, as the code it generates is shorter than the equivalent code generated by the second rule. It will also be natural to keep comparison as a special case when we add sequential logical operators.

6.6.1 Sequential Logical Operators

We will use the same names for sequential logical operators as C, i.e., `&&` for logical **and**, `||` for logical **or**, and `!` for logical negation. The extended language is shown in Grammar 6.9. Note that we allow an expression to be a condition as well as a condition to be an expression. This grammar is highly ambiguous (not least because **binop** overlaps **relop**). As before, we assume such ambiguity to be resolved by the parser before code generation. We also assume that the last productions of *Exp* and *Cond* are used when no other productions apply, as this will yield the best code.

The revised translation functions for *Exp* and *Cond* are shown in Fig. 6.10. Only the new cases for *Exp* are shown.

As expressions, `true` and `false` are the numbers 1 and 0.

A condition *Cond* is translated into code that chooses between two labels. When we want to use a condition as an expression, we must convert this choice into a number. We do this by first assuming that the condition is false by assigning 0 to the target location. We then, if the condition is true, jump to code that assigns 1 to the target location. If the condition is false, we jump around this code, so the value remains 0. We could equally well have done things the other way around, i.e., first assign 1 to the target location and modify this to 0 when the condition is false. Note

Grammar 6.9 Example language with logical operators

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
Exp & \rightarrow & \texttt{true} \\
Exp & \rightarrow & \texttt{false} \\
Exp & \rightarrow & Cond \\
\\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps \\
\\
Cond & \rightarrow & Exp\ \textbf{relop}\ Exp \\
Cond & \rightarrow & \texttt{true} \\
Cond & \rightarrow & \texttt{false} \\
Cond & \rightarrow & \texttt{!}\ Cond \\
Cond & \rightarrow & Cond\ \texttt{\&\&}\ Cond \\
Cond & \rightarrow & Cond\ \texttt{||}\ Cond \\
Cond & \rightarrow & Exp
\end{array}
$$

$Trans_{Exp}(Exp, vtable, ftable, place)$ = `case` Exp `of`	
	$\vdots$
`true`	$[place := 1]$
`false`	$[place := 0]$
$Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $[place := 0]$++$code_1$ ++[`LABEL` $label_1$, $place := 1$] ++[`LABEL` $label_2$]

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ = `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1$++$code_2$++[`IF` t_1 op t_2 `THEN` $label_t$ `ELSE` $label_f$]
`true`	[`GOTO` $label_t$]
`false`	[`GOTO` $label_f$]
`!` $Cond_1$	$Trans_{Cond}(Cond_1, label_f, label_t, vtable, ftable)$
$Cond_1$ `&&` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, arg_2, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$++[`LABEL` arg_2]++$code_2$
$Cond_1$ `||` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, arg_2, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$++[`LABEL` arg_2]++$code_2$
Exp	$t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1$++[`IF` $t \neq 0$ `THEN` $label_t$ `ELSE` $label_f$]

Fig. 6.10 Translation of sequential logical operators

that this code assigns to the *place* location before evaluating the condition, so it is important that *place* is not the name of a variable that might be used in the condition.

It gets a bit more interesting in $Trans_{Cond}$, where we translate conditions. We have already seen how comparisons and expressions are translated, so we move directly to the new cases.

The constant `true` condition just generates a jump to the label for true conditions, and, similarly, `false` generates a jump to the label for false conditions.

Logical negation generates no code by itself, it just swaps the attribute-labels for true and false when translating its argument. This effectively negates the argument condition.

Sequential logical **and** (`&&`) is translated as follows: The code for the first operand is translated such that if it is false, the second condition is not tested. This is done by jumping straight to the label for false conditions when the first operand is false. If the first operand is true, a jump to the code for the second operand is made. This is handled by using the appropriate labels as arguments to the call to $Trans_{Cond}$. The call to $Trans_{Cond}$ for the second operand uses the original labels for true and false. Hence, both conditions have to be true for the combined condition to be true.

Sequential **or** (`||`) is similar: If the first operand is true, we jump directly to the label for true conditions without testing the second operand, but if it is false, we jump to the code for the second operand. Again, the second operand uses the original labels for true and false.

Note that the translation functions now work even if **binop** and **unop** do not contain relational operators or logical negation, as we can just choose the last rule for expressions whenever the **binop** rules do not match. However, we can not in the same way omit arithmetic (bitwise) **and** and **or**, as these always evaluate both arguments, which the sequential equivalents do not. Replacing an arithmetic logical operator with a sequential ditto may seem like an optimisation, but there is a visible difference in behaviour: If the second argument has side effects (for example function calls), it is observable whether or not this is evaluated. So the two types of logical operators are not interchangeable.

We have, in the above, used two different nonterminals for conditions and expressions, with some overlap between these and consequently ambiguity in the grammar. It is possible to resolve this ambiguity by rewriting the grammar and get two non-overlapping syntactic categories in the abstract syntax. Another solution is to join the two nonterminals into one, e.g., *Exp* and use two different translation functions for this nonterminal: Whenever an expression is translated, the translation function most appropriate for the context is chosen. For example, `if-then-else` will choose a translation function similar to $Trans_{Cond}$ while assignment will choose one similar to the current $Trans_{Exp}$.

6.6.2 *Example of Translation of Conditions*

Using the rules in Figs. 6.5 and 6.10 and a symbol table that binds `x` to `v0` and `y` to `v1`, the following statement

```
y := x < 0 || !(x < 9)
```

gets translated into the code shown in Fig. 6.11.

Fig. 6.11 Example of translating conditions

```
 t1 := 0
  t2 := v0
  t3 := 0
  IF t2 < t3 THEN L1 ELSE L3
  LABEL L3
  t4 := v0
  t5 := 9
  IF t4 < t5 THEN L2 ELSE L1
 LABEL L1
 t1 := 1
 LABEL L2
v1 := t1
```

Note that the ! operator doesn't generate code of its own, it just swaps the branches of the comparison between `t4` and `t5`.
Suggested exercises: 6.3, 6.4.

6.7 Advanced Control Statements

We have, so far, shown translation of simple conditional statements and loops, but some languages have more advanced control features. We will briefly discuss how such can be implemented, without showing actual translation functions.

Goto and Labels

Source-code labels are stored in a symbol table that binds each source-code label to a corresponding label in the intermediate language. A jump to a label will generate a `GOTO` statement to the corresponding intermediate-language label. Unless labels are declared before use, an extra pass may be needed to build the symbol table before the actual translation. Alternatively, an intermediate-language label can be chosen, and an entry in the symbol table be created at the first occurrence of the label even if it is in a jump rather than a declaration. Subsequent jumps or declarations of that label will use the intermediate-language label that was chosen at the first occurrence. By setting a mark in the symbol-table entry when the label is declared, it can be checked that all labels are declared exactly once. This check ought to have been done during the type-checking phase (see Chap. 5), though.

The scope of labels can be controlled by the symbol table, so labels can be local to a procedure or block.

Break, Exit and Continue

Some languages allow exiting loops from the middle of the loop-body by a `break` or `exit` statement. To handle these, the translation function for statements must have an extra inherited attribute which is the label to which a `break` or `exit` statement must jump. This attribute is changed whenever a new loop is entered. Before the first loop is entered, this attribute is undefined. The translation function should check for this, so it can report an error if a `break` or `exit` occurs outside loops. This should, rightly, be done during type-checking, though.

C's `continue` statement, which jumps to the start of the current loop, can be handled similarly: Using an inherited attribute to set the label to which a `continue` statement should jump.

Case-Statements

A `case`-statement evaluates an expression and selects one of several branches (statements) based on the value of the expression. In most languages, the `case`-statement will be exited at the end of each of these statements. In this case, the case-statement can be translated as an assignment that stores the value of the expression followed by a nested `if-then-else` statement, where each branch of the `case`-statement becomes a `then`-branch of one of the `if-then-else` statements (or, in case of the default branch, the final `else`-branch).

In C, the default is that *all* `case`-branches following the selected branch are executed, unless the `case`-expression (called `switch` in C) is explicitly terminated with a `break` statement (see above) at the end of the branch. In this case, the `case`-statement can still be translated to a nested `if-then-else`, but the branches of these are now `GOTO`'s to the code for each `case`-branch. The code for the branches are placed in sequence after the nested `if-then-else`, with `break` handled by `GOTO`'s as described above. Hence, if no explicit jump is made, one branch will fall through to the next.

We will look at translation of a variant of case statements with pattern matching in Sect. 12.3.

6.8 Translating Structured Data

So far, the only values we have used are integers and booleans. However, most programming languages provide floating-point numbers and structured values like arrays, records (structs), unions, lists or tree-structures. We will now look at how these can be translated. We will first look at floats, then at one-dimensional arrays, multi-dimensional arrays and finally other data structures.

6.8.1 Floating-Point Values

Floating-point values are, in a computer, typically stored in a different set of registers than integers. Apart from this, they are treated the same way we treat integer values: We use temporary variables to store intermediate expression results and assume the intermediate language has arithmetic operators for floating-point numbers. The register allocator will have to make sure that the temporary variables used for floating-point values are mapped to floating-point registers. For this reason, it may be a good idea to let the intermediate code indicate which temporary variables hold floats. This can be done by giving them special names or by using a symbol table to hold type information.

6.8.2 Arrays

We extend our example language with one-dimensional arrays by adding the following productions:

$$\begin{array}{lcl} \mathit{Exp} & \rightarrow & \mathit{Indexed} \\ \mathit{Stat} & \rightarrow & \mathit{Indexed} := \mathit{Exp} \\ \mathit{Indexed} & \rightarrow & \mathbf{id}[\mathit{Exp}] \end{array}$$

Indexed is an array element, which can be used the same way as a variable, either as an expression or as the left part of an assignment statement.

We will initially assume that arrays are zero-based (i.e.. the lowest index is 0).

Arrays can be allocated statically, i.e., at compile-time, or *dynamically*, i.e., at run-time. In the first case, the *base address* of the array (the address at which index 0 is stored) is a compile-time constant. In the latter case, a variable will contain the base address of the array. In either case, we assume that the symbol table for variables binds an array name to the constant or variable that holds its base address.

Most modern computers are byte-addressed, while integers typically are 32 or 64 bits long. This means that the index used to access array elements must be multiplied by the size of the elements (measured in bytes), e.g., 4 or 8, to find the actual offset from the base address. In the translation shown in Fig. 6.12, we use 8 for the size of integers. We show only the new parts of the translation functions for *Exp* and *Stat*.

We use a translation function $\mathit{Trans}_{\mathit{Indexed}}$ for array elements. This returns a pair consisting of the code that evaluates the address of the array element and the variable that holds this address. When an array element is used in an expression, the contents of the address is transferred to the target variable using a memory-load instruction. When an array element is used on the left-hand side of an assignment, the right-hand side is evaluated, and the value of this is stored at the address using a memory-store instruction.

The address of an array element is calculated by multiplying the index by the size of the elements (here, 8) and adding this to the base address of the array. Note that *base* can be either a variable or a constant (depending on how the array is allocated, see below), but since both are allowed as the second operator to a **binop** in the intermediate language, this is no problem.

As an example, the assignment `a[x]` := `a[y]` is, given a symbol table that binds `a`, `x`, and `y` to `v1`, `v2`, and `v1`, respectively, translated into the code

$Trans_{Exp}(Exp, vtable, ftable, place) = \texttt{case}\ Exp\ \texttt{of}$	
$Indexed$	$(code_1, address) = Trans_{Indexed}(Indexed, vtable, ftable)$ $code_1 +\!\!+ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) = \texttt{case}\ Stat\ \texttt{of}$	
$Indexed := Exp$	$(code_1, address) = Trans_{Indexed}(Indexed, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 +\!\!+ code_2 +\!\!+ [M[address] := t]$

$Trans_{Indexed}(Indexed, vtable, ftable) = \texttt{case}\ Indexed\ \texttt{of}$	
$\mathbf{id}[Exp]$	$base = lookup(vtable, getname(\mathbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 +\!\!+ [t := t * 8, t := t + base]$ $(code_2, t)$

Fig. 6.12 Translation for one-dimensional arrays

```
 t2 := v2
 t2 := t2 * 8
 t2 := t2 + v1
 t3 := v3
 t3 := t3 * 8
 t3 := t3 + v1
t1 := M[t3]
M[t2] := t1
```

6.8.2.1 Allocating Arrays

So far, we have only hinted at how arrays are allocated. As mentioned, one possibility is static allocation, where the base-address and the size of the array are known at compile-time. The compiler, typically, has a large address space where it can allocate statically allocated objects. When it does so, the new object is simply allocated after the end of the previously allocated objects.

Dynamic allocation can be done in several ways. One is allocation local to a procedure or function, such that the array is allocated when the function is entered and deallocated when it is exited. This typically means that the array is allocated on a stack and popped from the stack when the procedure is exited. If the sizes of locally allocated arrays are fixed at compile-time, their base addresses are constant offsets from the stack top (or from the *frame pointer*, see Chap. 9) and can be calculated by adding the constant offset to the stack (or frame) pointer at every array-lookup, or once only at the entry of the function and then stored in a local variable. If the sizes

Fig. 6.13 A two-dimensional array

	1st column	2nd column	3rd column	⋯
1st row	`a[0][0]`	`a[0][1]`	`a[0][2]`	⋯
2nd row	`a[1][0]`	`a[1][1]`	`a[1][2]`	⋯
3rd row	`a[2][0]`	`a[2][1]`	`a[2][2]`	⋯
⋮	⋮	⋮	⋮	⋱

of these arrays are given at run-time, the offset from the stack or frame pointer to an array is not constant. So we need to use a variable to hold the base address of each array. The base address is calculated when the array is allocated and then stored in a local variable. This can subsequently be used as described in $Trans_{Indexed}$ above. At compile-time, the array-name will, in the symbol table, be bound to the variable that at runtime will hold the base-address.

Dynamic allocation can also be done globally, so the array will survive until the end of the program or until it is explicitly deallocated. In this case, there must be a global address space available for run-time allocation. Often, this is handled by the operating system which handles memory-allocation requests from all programs that are running at any given time. Such allocation may fail due to lack of memory, in which case the program must terminate with an error or release memory enough elsewhere to make room. The allocation can also be controlled by the program itself, which initially asks the operating system for a large amount of memory and then administrates this itself. This can make allocation of arrays faster than if an operating system call is needed every time an array is allocated. Furthermore, it can allow the program to use *garbage collection* to automatically reclaim the space used for arrays that are no longer accessible. We will return to garbage collection in Sect. 12.2.

6.8.2.2 Multi-dimensional Arrays

Multi-dimensional arrays can be laid out in memory in two ways: *row-major* and *column-major*. The difference is best illustrated by two-dimensional arrays, as shown in Fig. 6.13. A two-dimensional array is addressed by two indices, e.g., (using C-style notation) as `a[i][j]`. The first index, `i`, indicates the *row* of the element and the second index, `j`, indicates the *column*. The first row of the array is, hence, the elements `a[0][0]`, `a[0][1]`, `a[0][2]`, ... and the first column is `a[0][0]`, `a[1][0]`, `a[2][0]`, ...[1]

In row-major form, the array is laid out one row at a time and in column-major form it is laid out one column at a time. In a 3×2 array, the ordering for row-major is

```
a[0][0], a[0][1], a[1][0], a[1][1], a[2][0], a[2][1]
```

[1] Note that the coordinate system is rotated 90° clockwise compared to mathematical tradition.

For column-major the ordering is

```
a[0][0], a[1][0], a[2][0], a[0][1], a[1][1], a[2][1]
```

If the size of an element is *size* and the sizes of the dimensions in an n-dimensional array are $dim_0, dim_1, \ldots, dim_{n-2}, dim_{n-1}$, then in row-major format an element at index $[i_0][i_1]\ldots[i_{n-2}][i_{n-1}]$ has the address

$$base + ((\ldots(i_0 * dim_1 + i_1) * dim_2 \ldots + i_{n-2}) * dim_{n-1} + i_{n-1}) * size$$

In column-major format the address is

$$base + ((\ldots(i_{n-1} * dim_{n-2} + i_{n-2}) * dim_{n-3} \ldots + i_1) * dim_0 + i_0) * size$$

Note that column-major format corresponds to reversing the order of the indices of a row-major array. i.e., replacing i_0 and dim_0 by i_{n-1} and dim_{n-1}, i_1 and dim_1 by i_{n-2} and dim_{n-2}, and so on.

We extend the grammar for array-elements to accommodate multi-dimensional arrays:

$$\begin{aligned} Indexed &\rightarrow \mathbf{id}[Exp] \\ Indexed &\rightarrow Indexed[Exp] \end{aligned}$$

and extend the translation functions as shown in Fig. 6.14. This translation is for row-major arrays. We leave column-major arrays as an exercise.

With these extensions, the symbol table must return both the base-address of the array and a list of the sizes of the dimensions. Like the base-address, the dimension sizes can either be compile-time constants or variables that at run-time will hold the sizes. We use an auxiliary translation function $Calc_{Indexed}$ to calculate the position of an element. In $Trans_{Indexed}$ we multiply this position by the element size and add the base address. As before, we assume the size of elements is 8.

In some cases, the sizes of the dimensions of an array are not stored in separate variables, but in memory next to the space allocated for the elements of the array. This uses fewer variables (which may be an issue when these need to be allocated to registers, see Chap. 8) and makes it easier to return an array as the result of an expression or function, as only the base-address needs to be returned. The size information is normally stored just before the base-address so, for example, the size of the first dimension can be at address $base-8$, the size of the second dimension at $base-16$ and so on. Hence, the base-address will always point to the first element of the array no matter how many dimensions the array has. If this strategy is used, the necessary dimension sizes must be loaded into variables when an index is calculated. Since this adds several extra (somewhat costly) loads, optimising compilers often try to re-use the values of previous loads, e.g., by doing the loading once outside a loop and referring to variables holding the values inside the loop.

$$\begin{array}{l|l}
\multicolumn{2}{l}{Trans_{Exp}(Exp, vtable, ftable, place) = \texttt{case}\ Exp\ \texttt{of}} \\
\hline
Indexed & (code_1, address) = Trans_{Indexed}(Indexed, vtable, ftable) \\
 & code_1 \mathbin{++} [place := M[address]]
\end{array}$$

$$\begin{array}{l|l}
\multicolumn{2}{l}{Trans_{Stat}(Stat, vtable, ftable) = \texttt{case}\ Stat\ \texttt{of}} \\
\hline
Indexed := Exp & (code_1, address) = Trans_{Indexed}(Indexed, vtable, ftable) \\
 & t = newvar() \\
 & code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t) \\
 & code_1 \mathbin{++} code_2 \mathbin{++} [M[address] := t]
\end{array}$$

$$\begin{array}{l}
Trans_{Indexed}(Indexed, vtable, ftable) = \\
\hline
\quad (code_1, t, base, []) = Calc_{Indexed}(Indexed, vtable, ftable) \\
\quad code_2 = code_1 \mathbin{++} [t := t * 8, t := t + base] \\
\quad (code_2, t)
\end{array}$$

$$\begin{array}{l|l}
\multicolumn{2}{l}{Calc_{Indexed}(Indexed, vtable, ftable) = \texttt{case}\ Indexed\ \texttt{of}} \\
\hline
\mathbf{id}[Exp] & (base, dims) = lookup(vtable, getname(\mathbf{id})) \\
 & t = newvar() \\
 & code = Trans_{Exp}(Exp, vtable, ftable, t) \\
 & (code, t, base, tail(dims)) \\
\hline
Indexed[Exp] & (code_1, t_1, base, dims) = Calc_{Indexed}(Indexed, vtable, ftable) \\
 & dim_1 = head(dims) \\
 & t_2 = newvar() \\
 & code_2 = Trans_{Exp}(Exp, vtable, ftable, t_2) \\
 & code_3 = code_1 \mathbin{++} code_2 \mathbin{++} [t_1 := t_1 * dim_1, t_1 := t_1 + t_2] \\
 & (code_3, t_1, base, tail(dims))
\end{array}$$

Fig. 6.14 Translation of multi-dimensional arrays

6.8.2.3 Index Checks

The translations shown so far do not test if an index is within the bounds of the array. Index checks are fairly easy to generate: Each index must be compared to the size of (the dimension of) the array and if the index is too big, a jump to some error-producing code is made. If the comparison is made on unsigned numbers, a negative index will look like a very large index. Hence, a single conditional jump using unsigned comparison is inserted at every index calculation.

This is still fairly expensive, but various methods can be used to eliminate some of these tests. For example, if the array-lookup occurs within a `for`-loop, the bounds of the loop-counter may guarantee that array accesses using this variable will be within bounds. More generally, it is possible to make an analysis that finds cases where the index-check condition is subsumed by previous tests, such as the exit test for a loop, the test in an `if-then-else` statement or previous index checks. We will return to this in Chap. 10.

6.8.2.4 Non-zero-based Arrays

We have assumed our arrays to be zero-based, i.e., that the indices start from 0. Some languages (such as Pascal) allow indices to be arbitrary intervals, e.g., −10 to 10 or 10 to 20. If such are used, the starting-index must be subtracted from each index when the address is calculated. In a one-dimensional array with known size and base-address, the starting-index can be subtracted (at compile-time) from the base-address instead. In a multi-dimensional array with known dimensions, the starting-indices can be multiplied by the sizes of the dimensions and added together to form a single constant that is subtracted from the base-address, instead of subtracting each starting-index from each index. Even if the bounds are not known at compile time, a single offset can be calculated when the array is allocated.

6.8.3 Strings

Strings are often implemented in a fashion similar to one-dimensional arrays. In some languages (e.g. C or pre-ISO-standard Pascal), strings *are* just arrays of characters. This assumes a character is of fixed size, typically one byte. In some languages, such as Haskell, a string is a list of characters.

However, strings often differ from arrays in various ways:

- Different characters may have different size. For example, in UTF-8 encoding, characters in the ASCII subset are represented as one byte, and other characters as up to four bytes.
- Operations such as concatenating strings or extracting substrings are more common than similar operations on arrays.
- Many languages support lexicographic (alphabetic) comparison of strings, but not of arrays. Lexicographic ordering may even need to cater for local rules, e.g., where non-ASCII letters are placed in the alphabet and how certain ligatures (such as œ) are alphabetized.

So strings are often represented unlike arrays, both to cater for non-constant element size and to optimise operations such as concatenation and substring extraction. The representation can be a binary tree where leaf nodes contain single characters or short strings, and inner nodes store pointers to two subtrees as well as the size of the string stored in the left subtree. This allows relatively fast indexing and very fast concatenation.

Regardless of representation, operations on strings, such concatenation, comparison, and substring extraction, are typically implemented by library functions.

6.8.4 Records/Structs and Unions

Records (structs) have many properties in common with arrays. They are typically allocated in a similar way (with a similar choice of possible allocation strategies), and the fields of a record are typically accessed by adding an offset to the base-address of the record. The differences are:

- The types (and hence sizes) of the fields may be different.
- The field-selector is known at compile-time, so the offset from the base address can be calculated at this time.

The offset for a field is simply the sum of the sizes of all fields that occur before it. For a record-variable, the symbol table for variables must hold both the type and the base-address of the record. The symbol table for types must for a record type hold the types and offsets for each field in the record type. When generating code for a record field access, the compiler uses the symbol table for variables to find the base address and the type, which is used with the symbol table for types to find the field offset. Alternatively, the symbol table for variables can hold all this information.

In a union (sum) type, the fields are not consecutive, but are stored at the same address, i.e., the base-address of the union. The size of an union is the maximum of the sizes of its fields.

In some languages, union types include a *tag*, which identifies which variant of the union is stored in the variable. This tag is stored as a separate field before the union-fields. Some languages (e.g., Standard ML) enforce that the tag is tested when the union is accessed, others (e.g., Pascal) leave this as an option to the programmer.

Suggested exercises: 6.8.

6.9 Translation of Declarations

In the translation functions used in this chapter, we have several times required that "The symbol table must contain …". It is the job of the compiler to ensure that the symbol tables contain the information necessary for translation. When a name (variable, label, type, etc.) is declared, the compiler must, in the symbol-table entry for that name, keep the information necessary for compiling any use of that name. For scalar variables (e.g., integers), the required information is the intermediate-language variable that holds the value of the variable. For array variables, the information includes the base-address and dimensions of the array. For records, it is the offsets for each field and the total size. If a type is given a name, the symbol table must for that name provide a description of the type, such that variables that are declared to be that type can be given the information they need for their own symbol-table entries.

The exact nature of the information that is put into the symbol tables, and how this information is split among the symbol tables for types and the symbol table for variables or functions, will depend on the translation functions that use these tables, so it is usually a good idea to write first the translation functions for *uses* of names and then translation functions for their declarations.

6.9.1 Simple Local Declarations

We extend the statement language by the following productions:

$$\begin{aligned} Stat &\rightarrow Decl \; ; \; Stat \\ Decl &\rightarrow \texttt{int}\ \mathbf{id} \\ Decl &\rightarrow \texttt{int}\ \mathbf{id}[\mathbf{num}] \end{aligned}$$

We can, hence, declare integer variables and one-dimensional integer arrays for use in the following statement. An integer variable should be bound to a location in the symbol table, so this declaration should add such a binding to *vtable*. An array should be bound to a variable containing its base address. Furthermore, code must be generated for allocating space for the array. We assume arrays are heap allocated and that the intermediate-code variable *HP* points to the first free element of the (upwards growing) heap. Figure 6.15 shows the translation of these declarations using the simplifying assumption that there is enough space in the heap. A real compiler would need to insert code to check this, and take appropriate action if there is not enough space. See Sect. 12.2 shows more detail about heap allocation.

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`

$Decl$; $Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1$++$code_2$

$Trans_{Decl}(Decl, vtable) =$ `case` $Decl$ `of`

`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([], vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([t_1 := HP, HP := HP + (8 * getvalue(\mathbf{num}))], vtable_1)$

Fig. 6.15 Translation of simple declarations

6.9.2 Translation of Function Declarations

Given that the intermediate language includes function declarations, translating simple function definitions is quite easy: We translate a function declaration just by mapping the function and argument names to intermediate-language names in *vtable* and *ftable*, make a function header using the new names and then translating the body statement or expression using *vtable* and *ftable* as symbol tables. If the body is a statement, we just extend $Trans_{Stat}$ to translate a `return` statement into a `RETURN` instruction. If the body is an expression, we translate this and add a `RETURN` instruction to return its value. Local variable declarations are translated like the local declarations above.

As an example, the function definition

```
fac(n)
  f := 1;
  repeat
    f := f * n;
    n := n − 1
  until n = 0
  return f
```

is translated into the code shown in Fig. 6.16, using a *vtable* that binds `f` to `v0` and `n` to `v1`, and an *ftable* that binds `fac` to `_fac`. Note that the body of the function is very similar to the statement in Sect. 6.5.

Fig. 6.16 Example function definition translation

```
_fac(v1)
    t1 := 1
  v0 := t1
  LABEL L1
      t3 := v0
      t4 := v1
    v0 := t3 * t4
      t5 := v1
      t6 := 1
    v1 := t5 − t6
      t7 := v1
      t8 := 0
    IF t7 = t8 THEN L2 ELSE L1
  LABEL L2
RETURN v0
```

If functions can call functions that are declared later, we use two passes: One to build *ftable* and another to translate the function definitions using this *ftable*. This is similar to how we in Chap. 5 type-checked mutually recursive function definitions.

At some later point, we will need to expand the intermediate-level function definitions, calls and returns into lower-level code. We will return to this in Chap. 9.

Suggested exercises: 6.13.

6.10 Further Reading

A comprehensive discussion about intermediate languages can be found in [6].

Functional and logic languages often use high-level intermediate languages, which are in many cases translated to lower-level intermediate code before emitting actual machine code. Examples of such intermediate languages can be found in [2, 3] and [1].

A well-known high-level intermediate language is the Java Virtual Machine [5], abbreviated JVM. This language evaluates expressions using a stack instead of temporary variables, and has single instructions for such complex things as virtual method calls and creating new objects. The high-level nature of JVM was chosen for several reasons:

- By letting common complex operations be done by single instructions, the code is smaller, which reduces transmission time when sending the code over the Internet.
- JVM was originally intended for interpretation, and doing more work in a single instruction also helps reduce the overhead of interpretation.
- A program in JVM is *validated* (essentially type-checked) before interpretation or further translation. This is easier when the code is high-level.

The Java Virtual Machine has been criticised for making too many assumptions about the source language, which makes it difficult to use for languages that are dissimilar to Java. Since JVM was designed specifically for Java, this is not surprising. A less language-specific intermediate language is The Low-Level Virtual Machine [4], abbreviated LLVM. Where JVM uses a stack for temporary values, LLVM (like the intermediate language used in this chapter) uses temporary variables, and it uses *static single assignment form* (SSA), which we will look at in Sect. 10.11.

6.11 Exercises

Exercise 6.1 Use the translation functions in Fig. 6.3 to generate code for the expression `2+g(x+y,x*y)`. Use a *vtable* that binds `x` to `v0` and `y` to `v1` and an *ftable* that binds `g` to `_g`. The result of the expression should be put in the intermediate-code variable `r` (so the *place* attribute in the initial call to $Trans_{Exp}$ is `r`).

Exercise 6.2 Use the translation functions in Figs. 6.5 and 6.6 to generate code for the statement

```
x:=2+y;
if x<y then x:=x+y;
repeat
  y:=y*2;
  while x>10 do x:=x/2
until x<y
```

use the same *vtable* as in Exercise 6.1.

Exercise 6.3 Use the translation functions in Figs. 6.5 and 6.10 to translate the following statement

```
if x<=y && !(x=y || x=1)
then x:=3
else x:=5
```

use the same *vtable* as in Exercise 6.1.

Exercise 6.4 De Morgan's law tells us that `!`$(p$ `||` $q)$ is equivalent to `(!`p`)&&(!`q`)`. Show that the two conditions generate identical code when compiled with $Trans_{Cond}$ from Fig. 6.10.

Exercise 6.5 Show that, in any code generated by the functions in Figs. 6.5 and 6.10, every `IF-THEN-ELSE` instruction will be followed by one of the target labels.

Exercise 6.6 Extend Fig. 6.5 to include a `break`-statement for exiting loops, as described in Sect. 6.7, i.e., extend the statement syntax by

$$Stat \rightarrow \texttt{break}$$

and add a rule for this to $Trans_{Stat}$. Add whatever extra attributes you may need to do this.

Exercise 6.7 We extend the statement language with the following statements:

$$Stat \rightarrow \textbf{labelid}\ :$$
$$Stat \rightarrow \texttt{goto}\ \textbf{labelid}$$

for defining and jumping to labels.

Extend Fig. 6.5 to handle these as described in Sect. 6.7. Labels have scope over the entire program (statement) and need not be defined before use. You can assume that there is exactly one definition for each used label.

Exercise 6.8 Show translation functions for multi-dimensional arrays in column-major format. **Hint:** Starting from Fig. 6.14, it may be a good idea to rewrite the productions for *Index* so they are right-recursive instead of left-recursive, as the address formula for column-major arrays groups to the right. Similarly, it is a good idea to reverse the list of dimension sizes, so the size of the rightmost dimension comes first in the list.

Exercise 6.9 When statements are translated using the functions in Fig. 6.5, it will often be the case that the statement immediately following a label is a `GOTO` statement, i.e., we have the following situation:

$$\begin{array}{ll}\texttt{LABEL} & \mathit{label}_1\\ \texttt{GOTO} & \mathit{label}_2\end{array}$$

It is clear that any jump to $label_1$ can be replaced by a jump to $label_2$, and that this will result in faster code. Hence, it is desirable to do so. This is called jump-to-jump optimisation, and can be done after code-generation by a post-process that looks for these situations. However, it is also possible to avoid most of these situations by modifying the translation function.

This can be done by adding an extra inherited attribute *endlabel*, which holds the name of a label that can be used as the target of a jump to the end of the code that is being translated. If the code is immediately followed by a `GOTO` statement, *endlabel* will hold the target of this `GOTO` rather than a label immediately preceding this.

(a) Add the *endlabel* attribute to $Trans_{Stat}$ from Fig. 6.5 and modify the rules so *endlabel* is exploited for jump-to-jump optimisation. Remember to set *endlabel* correctly in recursive calls to $Trans_{Stat}$.
(b) Use the modified $Trans_{Stat}$ to translate the following statement:

```
while x>0 do
  x := x-1;
  if x>10 then x := x/2
```

The extent of the while loop is indicated by indentation.
Use the same *vtable* as Exercise 6.1 and use `endlab` as the *endlabel* for the whole statement.

Exercise 6.10 In Fig. 6.5, `while` statements are translated in such a way that every iteration of the loop executes an unconditional jump (`GOTO`) in addition to the conditional jumps in the loop condition.

Modify the translation so each iteration only executes the conditional jumps in the loop condition, i.e., so an unconditional jump is saved in every iteration. You may have to add an unconditional jump outside the loop.

Exercise 6.11 In mathematics, logical conjunction is associative:
$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$

Show that this also applies to the sequential conjunction operator `&&` when translated as in Fig. 6.10, i.e., that p `&&` $(q$ `&&` $r)$ generates the same code (up to renaming of labels) as $(p$ `&&` $q)$ `&&` r.

Exercise 6.12 Figure 6.14 shows translation of multi-dimensional arrays in row-major layout, where the address of each element is found through multiplication and addition. On machines with fast memory access but slow multiplication, an alternative implementation of multi-dimensional arrays is sometimes used: An array with dimensions $dim_0, dim_1, \ldots, dim_n$ is implemented as a one-dimensional array of size dim_0 with pointers to dim_0 different arrays each of dimension $dim_1, \ldots, dim_n$, which again are implemented in the same way (until the last dimension, which is implemented as a normal one-dimensional array of values). This takes up more room, as the pointer arrays need to be stored as well as the elements. But array-lookup can be done using only addition and memory accesses.

(a) Assuming pointers and array elements need eight bytes each, what is the total number of bytes required to store an array of dimensions $dim_0, dim_1, \ldots, dim_n$?
(b) Write translation functions for array-access in the style of Fig. 6.14 using this representation of arrays. Use addition to multiply numbers by 8 for scaling indices by the size of pointers and array elements.

Exercise 6.13 We add function declarations and function return to the example language by adding the productions

$$
\begin{array}{lcl}
FunDec & \rightarrow & \textbf{id}\ (\ Params\ )\ Stat \\
Params & \rightarrow & \textbf{id} \\
Params & \rightarrow & \textbf{id}\ ,\ Params \\
Stat & \rightarrow & \texttt{return}\ Exp
\end{array}
$$

Using the informal explanation in Sect. 6.9.2, extend $Trans_{Stat}$ and write translation functions $Trans_{FunDec}$ and $Trans_{Params}$ to implement these extensions. You can assume that you already have a *ftable* that maps source-level function names to intermediate-code function names, so this can be used as inherited attribute by $Trans_{FunDec}$. You can also assume that there are no repeated parameter names, as this would have been detected by a type checker.

References

1. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. MIT Press (1991)
2. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
3. Jones, S.L.P., Lester, D.: Implementing Functional Languages – A Tutorial. Prentice Hall (1992)
4. Lattner, C.: LLVM language reference manual (2011). http://llvm.org/docs/LangRef.html
5. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading, Massachusetts (1999)
6. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)