Course Script

INF 5110: Compiler construction

INF5110, spring 2024

Martin Steffen

Contents

Chapter 9
Intermediate code generation

What is it about?

Learning Targets of this Chapter

1. intermediate code
2. three-address code and P-code
3. translation to those forms
4. translation between those forms

Contents

9.1 Intro

At the current stage in the lecture (and the current "stage" in a compiler) we have to process as input an abstract syntax tree which has been type-checked and which thus is equipped with relevant type information. As discussed, key type information is often not stored *inside* the AST, but associated with it via a symbol table. More precisely, the symbol table mostly stores type information for variables, identifiers (the "symbols"), etc., not for all nodes of the AST, since that it typically sufficient. As far as code generation is concerned, we have at least gotten a feeling for certain aspects of code generation, without details, namely in connection with implementing high-level abstractions in connection with *data*. The layout of how certain types can be implemented and how scoping, memory management etc. is arranged. As far as the *control-part* of a program is concerned (not the data part), we also know that the run-time environment maintains a stack of *return adresses* to take care of the call-return behavior of the procedure abstraction. We have also seens, though not in very much detai, the so-called *calling conventions* and *calling sequences*, low-level instructions that take care of "data-aspects" of maintaining the procedure abstraction (taking care of parameter passing, etc.). All that was done, as said, not with concrete (machine) code, but explaining what needs to be achieved and how those aspects (memory management, stack-arrangement etc.) are designed.

The task of code generation is to generate instructions which are put into *code segment* which is a part of the *static* part of the memory. That concept was discussed in the introductory part of the chapter about run-time environments. (Intermediate) code generation is basically the task to translate procedure *bodies* into sequences of instructions.

Ultimately, the generated instructions are binaries, resp. in machine code, which is *platform dependent*. Generating platform dependent code is thus part of the back-end. The task of generating code, however, is usually split into generating first **platform-independent intermediate code** and afterwards, real code. This chapter here is about this intermediate code generation. Intermediate code as another *intermediate representation* internal to the compiler is commonplace. It may take different forms, however, and we will encounter two flavors.

Why does one want another intermediate representation as opposed to go all the way to machine code in one step? There are a couple of reasons for that. Code generation may not be altogether trivial. Especially, at the lower ends of the compiler, one may throw many different and complex optimizations at the task. So, **modularizing** the task into smaller subphases is good design. Related to that: doing it stepwise helps in **portability.** The intermediate code still is kind of machine independent. It may resemble the instruction set of typical hardware, or more likely resembling a subset of such an instruction set leaving out esotheric, specialized commands some hardwares may offer. Additionally, intermediate code will still rely on some **abstractions** no longer available on any hardware binaries. One is that the IC typically still works with *variables* and so-called temporaries, where ultimately the real code operates on addresses and registers.

If one has some machine-code resembling the chosen intermediate representation, the task of porting a compiler to the platform is easier. Furthermore, one can start doing certain code analyses and optimization already on the IC, thereby making optimizations available for all platform-dependent backends, without reimplementing the wheel multiple times. Of course, analyses and optimizations could and should *also* be done on the platform-dependent phase. For instance, crucially important for the ultimate perfomance of the code is the good use of **registers**. That, however, is platform dependent: different chips offer different register sets and support different ways of using them, reserving some registers for special use. Also in this lecture, the intermedatiate code generation postpones register allocation for the subsequent phase and chapter.

We said, that IR is platform independent. That does not mean, that it may not be "influenced" by targeted platforms. The are different flavors of instruction sets (RISC vs. CISC, three-address code, two-address code etc.), and the intermediate code has to make a choice what flavor of instructions it plans resemble most. We will deal with two prominent ways. One is a three-address code, the other one is P-code (which could be also called 1-address code). The latter one does not resembles typical instruction sets, but is a known IC format nonetheless. It resembles (conceptually) byte-code.

Code comes in different forms, there is relocatable vs. "absolute" code, relocatable code from libraries, assembler code, etc. Depending on the operating system, files containing code often carry specific file extensions. In Unix/Linux uses extensions as `*.s`, `*.o`, and `*.a` for assembler code, resp. relocatable code resp. relocatable code from a library.

Absolute code is stored in files without file extension (but set as "executable" as file permission). Windows use `.exe` as file extension.[1]

One big distriction is between code "natively" executable, i.e., on a particular (HW) platform on the one hand, and "byte code" or related concepts on the other. The latter is a terminology mostly known in connection with Java, while the underlying concept is not. E.g., in .NET/C♯, there is *CIL* (common intermediate language). It's actually sometimes called p-code (representing *portable code* or *interpreter code*). It's not natively executed but run on an *interpreter* or *virtual machine* (for Java byte code, that's of course the JVM). The terminology "byte code" refers to the fact that the op-codes, i.e., instructions of the byte code language, are intended to be represented by one byte. That piece of information alone, that opcodes fit into one byte, does not give much insight, though, and there may be many different "byte code representations". They are often intendend to be executed on a virtual machine, but of course they can also be used as another intermediate representation (in the sense of the topic of this chapter). A virtual machine is a "machine" simulated in software, and the architecture can resemble the execution mechanism of HW, or can follow principles typically not found in HW. For example, one typical architecture is a *stack machine*. One find also virtual machines that resemble *register machines*.

We will look into two formats, one called p-code, one called three-address intermediate code (3AIC). As indicated, the terminology is a bit fuzzy: P-code normally and originally stands for portable code, but 3AIC is also portable. P-code here resembles (at least conceptually) Java byte code, but also the op-code of 3AIC would fit into one byte.

As further remark concerning interpretation and "virtual machines" and virtualization in general. The distinction between compilation and interpretation is not a matter of black and white. Already in the introductory chapter, full interpretation was mentioned, where the execution is done directly on the user syntax is rather seldom. "Directly on the syntax" can mean on some abstract syntax, which is seen as basically as the programming language syntax, just stripped from the particularities of concrete syntax. But doing rewriting directly on that level, in particular on concrete syntax and on character string level, is an unpractical execution mechanism. Interpreting a language on a virtual machine is already quite closer to machine execution, the virtual machine works like a software-simulated machine model, and that may be more or less low-level. On the very lowest end, there is complete virtualization, where a whole operating system is simulated (maybe running multiple instances of operating system "on the cloud"). In that case, one can generate *native* code.

As mentioned, we will discuss 3AIC and p-code. P-code may be called one-address-code. A good criterion for different ICs is the *format* of the instructions, a better criterion at any rate than the size of the op-code (one byte) or the fact that it's portable (p-code). By format one mainly refers to how many arguments (many of) the instructions take. One, two, three, there is even zero-address code. So, that is one dimension for classification of intermediate code. Another dimension is what kind of *addressing modes* are supported. That has to do (often) with the use of registers. Not all intermediate codes work with the concept of registers, for instance, in this lecture, the two formats are *independent* from

[1] `.exe`-files include more, and "assembly" in .NET even more

registers, and we also don't go into details here of indirect addressing and similar, which are often used in connection with registers, but can also be understood independently.

As far as the different formats go: formats like 3AC and 2AC are common for current HW. That means, that 3AIC is a viable format (resembling current HW). 1-address code and 0-address code is not really found as HW design, but still a viable format for intermediate code. Especially for intermediate code intended to run on a virtual machine. One example is JVM and Java byte code. However, historically, there are machine designs based on such idea. One very early was the British KDF9 computer, which used a zero-address format and, more widely known, some designs from the Burroughs company (like the very unique B5000). A programming language, which gives a feeling of stack-machine programming is *Forth* (there is a linux/gnu version of it (`gforth`)). Forth, in a way, continues to live on in the form of the well-known Postscript language (run on printers), Postscript[2] is said to be inspired by Forth.

9.2 Intermediate code

This section gives a short preview of the two forms of intermediate code we will cover in the lecture. Three-address intermediate code is covered in Section 9.3 and p-code in Section 9.4.

Three-address code (3AC, 3AIC) is a generic, platform-independent, abstract machine code. It uses new variables for intermediate results. Those variables are called temporary variables or **temporaries.** They can be seen as unbounded pool of maschine registers. **P-code** is a different form, originally proposed for interpretation, but p-code is now often translated before execution (cf. also JIT-compilation). The word stands for portable code. It's often also called "Pascal-code", as such a representation was prominently used in Pascal, but also Java byte code resembles p-code. While 3AIC (and 2AIC) uses temporaries, p-code organizes the computation storing intermediate results in a **stack**, use with postfix operations.

There are *many* variations and elaborations for both kinds. One difference could be whether addresses are represented *symbolically* or as *numbers* (or both). Also the level of supported abstraction and the granularity and the instruction set: the high-level operations available e.g., for array-access or is already a translation in more elementary operations needed. Are operands (still) typed or not ...

There will various **translations** involving the the intermediate code languages in this chapter (see Figure 9.1). Abstract syntax trees here are assumed tree structures *after* semantic analysis, for instance with type information available. Let's call it AST^+ or just simply AST. The chapter touches upon general problems and techniques in translating to intermediate code, as well as translating from one form to the other. One (important) aspect ignored for now is *register allocation*, as we are dealing with *intermediate* code.

[2] Postscript is an interesting language in our context discussing high-level, low-level, and intermediate level languages. It's not only a Turing-complete (domain-specific) programming language, but it is also intended to directly run on hardware/firmware, on printers. PDF, which came later and unlike Postscript, is not a general-purpose language.

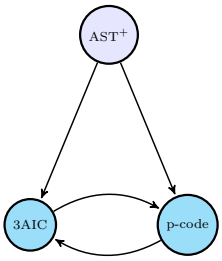


Figure 9.1: Different translations involving intermediate code

As mentioned earlier, the translation from typed ASTs to p-code corresponds to the task of the second **oblig,** where the target code will be some stack-oriented byte-code format. The corresponding interpreter or virtual machine as execution mechanism is provided in the form of a Java library.

9.3 Three-address (intermediate) code

Three-address code is a common format, not just for intermediate code, but also for machine code. The name comes from that fact that some instructions make use of three "addresses". Not all operations use three, some use less, but the most general ones make use of 2 **source** variables or addresses for the arguments, and one **target** variable or address for the result. In particular, binary operations that do calculations use 3, like addition or bitwise and. See equation (9.1).

We mentioned that our intermediate code does not make use of addresses and registers (which is common for intermediate code). That means, the instructions don't literally work with 3 addresses, but rather they involve 3 variables or constants. Beside "ordinary" variables (like the ones that originate from the source code), the intermediate code generation introduces **temporary variables** or **temporaries** for short to store intermediate results. At this phase there is no attempt to economize on the amount of temporaries. An unbounded supply of those temporaries is assumed, and each time some intermediate result needs to be remembered, a fresh temporary is used for that.

Of course, ultimately, that's a wasteful use of memory. In particular, ultimately the temporaries should be preferably be stored in registers, and there will be a limited amout of them; temporaries are typically short-lived, so often after having served their purpose storing an intermediate result, the space, like a register, can be reused to hold the next intermediate result. Of course not just temporaries are better be kept in registers, if possible. Also ordinary variables compete for the scarce register resource, passing parameters via registers may be a good idea, etc.

All that is a complex optimization task, and since our intermediate code is platform independent, it's not clear at that point, how many register there will be. Thus, there is not too much motivation to economize on temporaries already now, which greatly simplifies the task of intermediate code generation.

The 3AIC is also a **linear** form of intermediate code. That means, a piece of intermediate code is an instruction *list* (not a (syntax) tree or a graph, or some other more structured representation). That also means, for non-linear control flow, there are op-codes for **jumps** and **conditional jumps**; as opposed to more structured syntax, like conditionals or loops. Those would correspond to a tree-structured, not linear code format. A linear instruction list, perhaps stored in an array, resembles the arrangement of actual machine code, with the position of the instruction inside the list or array being an abstract form of its address.

Jump intructions transfer the control to a specified address, the control "jumps to" the instruction at that target address. To jump to one instruction, one could use its position in the list to specify that. That's ultimately also what will later happen in real machine code (jumping to addresses). One can do that more elegantly, however, specifying jump targets **symbolically**. The "symbols" to represent jump targets (or lines of code, or abstract addresses) are called **labels**. So the intermediate code allow to label instructions, giving them unique labels. Concretely, the 3AIC here does not directly label instructions (which would be an alternative way of doing it), it's rather that there is an extract *label instruction* which is part of the instruction set. Of course, it's equivalent. Adding a label instruction like `label L` means that one can use for instance `jmp L` to jump effectivly the instruction *following* the line `label L`. Jumping to a position in a program will be translated to a real machine code instruction. Being jumped-to for a labelled place will, of course, not be reflected by some instruction in the machine code. Therefore, instructions like `label L` are also called **pseudo instructions**.

Jumping (and labelling) take care of the control flow. Jump statements obviously don't use 3 addresses, as in equation (9.1). And indeed, jumping and labelling is independent of the general instruction format, and that means that also the one-address code or p-code from Section 9.4 will use the same principles (and the same can be done for 2-address code).

There is some parallel between labels and temporaries. Both are symbolic representations of addresses. Temporaries (like variables) correspond to addresses containing *data*, labels represent addresses to jump to, and that point in the control flow graph. Besides that, in both cases, the code generator assumes an unlimited reservoir of those and labels and both are never reused. Each time the code generator encounters the need to store an intermedate result or need to specify another jump target, it generates a fresh temporary resp. a fresh jump label. Of course, the p-code later will not make use of temporaries. Instead it will employ an (unbounded) stack to store intermediate results, so there will be no need to create fresh temporaries.

Three-address-code is named after the form of its assignment, which in the most general form looks as follows

$$x = y \ \mathbf{op} \ z \tag{9.1}$$

where x, y, and z represent names, constants, temporaries ..., and not all instructions in three address code have three arguments, some operations need fewer arguments.

It's a typical example of a **linear IR**. Linear means that the code consists of a sequences of extructions, there are no constructs for *structured* programming like loops or conditionals.

Control flow is represented by sequences of three-address instructions and jumps resp. conditional jumps. There are other linear intermediate representations or instruction sets (on a similar level of abstraction), like 1-address (or even 0) code, which represent stack-machine code, and 2 address code. Three-address code is is well-suited for optimizations and modern architectures often have 3-address code like instruction sets, especially on RISC-architectures.

Example 9.3.1 (3AIC example (expression)). Figure 9.2 shows an abstract syntax tree for the expression $2 * a + (b - 3)$. Listing 9.1 and 9.2 show two possible (and quite similar)

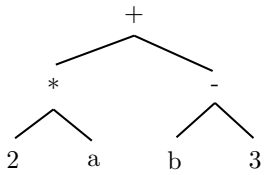


Figure 9.2: AST

translations into three-address intermediate code.

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

Listing 9.1: 3AIC

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

Listing 9.2: Alternative sequence

Both use 3 temporary variables, which correspond to the 3 non-leaf nodes in the abstract syntax tree. The fact that both do the same captures the fact that the order of evaluation does not matter. □

We have encountered the notion of **temporaries** already in connection with the activation records. Activation records needs space for various things, like parameters, local variables, return addresses etc., but also for *intermediate results*. That's the temporary variables of the intermediate code or temporaries for short, which we talk about here.

In our code examples, though, the convention is: different variable names mean different memory locations, so by writing `a` and `b`, there is no aliasing. Of course, if the 3AIC uses *references* (resp. indirect addressing), then different variable names don't guarantee absence of aliasing. A related remark concerns the temporaries. The example uses three different ones t_1, t_2, and t_3. Using different names for the temporary indicate that they are all different. However, that may look like a waste of memory: One could have "optimized" it by perhaps avoiding t_3 and reuse t_2 or t_3. One could indeed, but we discussed hat earlier: code generation at the current stage does not try to cut down on the use of temporaries. For each intermediate result, it uses just a new, fresh temporary. It will be the task of later stages, to do something about it, like minimizing the number of temporaries (and put as many of them into registers). However, the amount of registers is typically only known at the platform-dependent stage. Most intermediate code formats (like ours) are unaware of registers or, in other words, assume a (abstract) machine model without registers.

Using a fresh temporary each time we need one means, each temporary is assigned-to only *once* (at least if we ignore loops). That restriction is sometimes called *static single*

assignment. Static means, there is only one line in the code ("statically") where a variable is assigned to. That does not guarantee "dynamic" or absolute single assignemnt: because of loops or subroutines, a variable that is statically only assigned one, may be assigned to more than once. Note that that SSA restriction applies to temporaries only, user-level variables may be assigned to multiple times.

There is also the possibility, to make also the standard variables to follow the SSA regime. This actually is a quite popular format for intermediate code, and has advantages concerning subsequent semantic analyses and optimization. In its generality, SSA a bit more complex than just using new variables all the time. Therefore we won't go into that.

The 3AIC **instruction set** supports $x = y \,\mathbf{op}\, z$ (as shown in equation (9.1), but also $x = \mathbf{op}\, z$, and $x = y$ can be used. The instruction set supports a number of operators, like +,-,*,/, <, >, `and`, `or`, operations for I/O (`read` x, `write` x), a labelling instruction `label` L (sometimes called a pseudo-instruction), jumps and conditional jumps. `if_false` x `goto` L. Besides variables, as said, 3AIC makes use of temporaries t_1, t_2, t_3 .... (or `t1, t2, t3` .... It's assume that there is an *unbounded* reservoir of those.

The terminology of **pseudo instruction** comes from the fact that there is no real instruction connected to it. It's just a way to refer to the corresponding line number a bit more abstractly. So, in a similar way that temporaries are an abstraction of memory locations (ultimately addresses in main memory, if registers cannot be used), labels are likewise a representation of addresses, ultimately translated to relocatable addresses and ultimately to addresses in the code segment.

Example 9.3.2 (Translation to 3AIC). Let's illustrate the translation two three-address intermediate code on a small example. See the source code from Listing 9.3.

```
read x;      // input an integer
if  0<x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact // output: factorial of x
end
```

Listing 9.3: Factorial (source)

The translated code is shown in Listing 9.4. The translation is pretty straightforward. It makes use of a number of temporaries. For instance, the format for conditional jumps insists that the condition argument is stored in a variable or temporary.[3]

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
```

[3] Also a constant value would be allowed, in which case it is more an unconditional jump ...

```
if_false t4 goto L2
write fact
label L1
halt
```

Listing 9.4: Factorial: 3AIC

□

There can be **variations** in the design of 3AI-code. There can be arithmetic operators for `int`, `long`, `float` .... There are different ways how to represent program *variables* (by names resp. symbols, or by pointing to the declaration in the symbol table, or already (abstract) machine addresses?

There are also different ways to store and represent 3A *instructions* as data structure inside the compiler. One obvious way is by a **quadruples** (in the most general case). There are three "addresses" plus the operator. Alternative, a *triple* is possible, if it's used in an intermediate code generator that uniformely assumes that the target-address, the left-hand side, is always a *new temporary*.

```
typedef enum {rd,gr,if_f,asn,lab,mul,
              sub, eq, wri, halt, ... } OpKind;
typedef enum {Empty, IntConst, String } AddrKind;

typedef struct  {
  AddrKind kind;
  union {
    int val;
    char * name;
  } contents;
}  Address;

typedef struct {
  OpKind op;
  Address addr1, addr2, addr3;
} Quad
```

Listing 9.5: Possible quadruple representation in C

A 3A(I)C has three addresses and one piece of information to specify the instruction itself. That makes 4 pieces of information, a quadruple. The code illustrates how one could represent it in C. It would look analogous to some extent in other languages. As a reminder of the typing section: we see how the representation uses the (not-so-type-safe) union type of C, to squeeze a few bits. We also see the use of so-called `enum` type for finite enumerations. The code is meant as illustration of how it can be done in C, but it depends obviously on details of the specification of the intermediate code and the supported types (here called kinds in the code).

9.4 P-code

As mentioned, one of the two formats covered in the chapter is sometimes called **p-code**. We also said that the terminolgy is not so informative. Perhaps a better name would be one-address code. There is even zero-address code (which works similarly), but we don't

cover it. Both one-address code and zero-address code have in common that they rely on **stack**-manipulations. Very roughly, where 3AIC uses temporaries to store intermediate results, p-code stores those on the stack. Its handling of expressions is done in a **post-fix** manner. We will see details for both later, when we look how to compile to either intermediate code format.

While we cover 3AIC and "1AIC" (p-code), there is also 2AC / 2AIC, which we will not cover, at least not in this chapter.[4] For the real code generation, we may have a look at the problem: how to generate 2AC from 3AIC, in particular how to deal with registers (assuming a 2AC hardware platform).

P-code is an abbreviation for *portable code.* Some also connect it to Pascal (like "p" stands for Pascal). Many Pascal compilers were based on p-code for reasons of portability. Pascal was influential some time ago, especially for computer science curricula. The so-called *p-code* machine was not invented for Pascal or by the Pascal-people, but perhaps Pascal was the most prominent language run on a p-code architecture. So, in a way, p-code was some LLVM or JVM of the 70ies and 80ies...

Example 9.4.1 (P-code example $(2*a+(b-3))$). Let's see the p-code for a small, compound expression, again $2*a+(b-3)$ from Example 9.3.1 and abstract syntax tree from Figure 9.2.

```
ldc 2   ; load constant 2
lod a   ; load value of variable a
mpi     ; integer multiplication
lod b   ; load value of variable b
ldc 3   ; load constant 3
sbi     ; integer substraction
adi     ; integer addition
```

Listing 9.6: p-code

□

The code should be clear enough (with the help of the comments on the right-hand column). This first example is concerned with *expression evaluation*, in particular expressions without side effects. Expressions are dealt with in **post-fix** manner. The expression is built-up from *binary* operators. Those work in a stack-like virtual machine as follows: both arguments have to be on top of the stack, then executing the opcode corresponding to the binary operators takes those top to elements and removes them them from the stack ("pop"), connects them as argments of the operation, and the result is the the new top of the stack ("push").

That pattern can be seen clearly in the code 3 times (there are three operators to be translated, addition, multiplication, and substraction). Constants and variables are pushed onto the stack by corresponding load-commands (`ldo` and `ldc`).

Loading the content of a variable with `ldo`, as shown in this example, is only one way to to "load a variable", namely loading its **content**. There is a second way, namely loading the **address** of a variable. That is not needed for evaluating expressions, and therefore

[4] Two-address codes have fallen more or less in disuse for *intermediate* code.

not part of this example. Next we show the translation of **assignment** to p-code. For that one needs both versions of the load-command.

Example 9.4.2 (P-code for assignments). Let's look at how to translate the following assignment:

$$\texttt{x} := \texttt{y} + 1$$

The generated p-code is shown in Listing 9.7.

```
lda x      ; load address of x
lod y      ; load value of y
ldc 1      ; load constant 1
adi        ; add
sto        ; store top to address
           ; below top & pop both
```

Listing 9.7: p-code (assignment)

The message of this example concerns the treatment of variables, in particular the fact that variables on the left-hand side of an assignment are treated differently from those on the right-hand side. For the programmer, the distinction may not always be too visible. Of course, one is aware that in an assignment, like the one shown in the code, the variable on the left hand side is assigned to, the variable on the right-hand side is read from. Everyone knows that. We write `:=` for assignments, to make the distinction more visible. In languages like C and Java, that is not visible, one writes `=` for assignment, but it's not equality: it's not symmetric in that `a=b` is not the same `b=a`, when `=` is meant as assignment. Of course, everyone knows that too.

In the generated code, we see a (related) difference, which may be less obvious. For `x`, the **address** is loaded as part of a step, for `y` it's the value or **content.** We need the address of `x` to store back the result at the end of the generated code. One also speaks of the **L-value** and the **R-value** of a variable.

We mentioned that the stack-machine architecture leads to a post-fix treatment of evaluation. That is true as long as one interprets "evaluation" as determining, in a side-effect free manner the *value* of expression (like in the previous example). Now, in this example, there are side-effects and the strict post-fix schema no longer works: the *first* thing to do is load the address of `x` with `lda`, i.e., that's not "post-fix", that is "pre-fix" treatment.

Finally a comment to the last opcode `sto`: it takes arguments (on the stack), and stores, in the example, the result of the computation to the given address (which here is the address of `x`). Additionally, *both* top elements are **popped off** the stack. Consequently, the value as the result of the commputation on the right-hand side is *no longer available*. So, this translation does *not* correspond to the semantics of assignments in languages like C and Java. There, things like `(x := y +1) + 5` are allowed, but for a compilation of a languages with this kind of semantics, the `sto` command, popping off both elements, is not how it's done. We see below an alternative operation, `stn`, which abbreviates *store non-destructively*, which would be adequate if one had a semantics as in Java or C. □

Example 9.4.3 (Factorial function). Let's translate also the factorial function into p-code. The source code was shown earlier in Listing 9.3, as part of the translation to 3AIC in Example 9.3.2.

```
lda x      ; load address of x
rdi        ; read an integer, store to
           ; address on top of stack (& pop it)
lod x      ; load the value of x
ldc 0      ; load constant 0
grt        ; pop and compare top two values
           ; push Boolean result
fjp L1     ; pop Boolean value, jmp to L1 if false
lda fact   ; load address of fact
ldc 1      ; load constant 1
sto        ; pop two values, store first
           ; to address represented by the second
lab L2     ; pseudo instruction L2
lda fact   ; load address of fact
lod fact   ; load value of fact
lod x      ; load value of x
mpi        ; multiply
sto        ; store top to address of second, and pop
lda x      ; load address of x
lod x      ; load value of x
ldc 1      ; load constant 1
sbi        ; substract
sto        ; store (as before)
lod x      ; load value of x
ldc 0      ; load constant 0
equ        ; test for equality
fjp L2     ; jump to L2 if false
lod fact   ; load value of fact
wri        ; write top of stack & pop
lab L1     ; pseudo instruction
stp        ; stop
```

Figure 9.3: Factorial (p-code)

□

9.5 Generating P-code

After having introduced the concept of p-code in Section 9.4, including (relevant parts of) the instruction set, let's have a look at code **generation**; we will do the same for 3AIC in Section 9.6. Actually, it's not very hard. We have a look at that problem from different angles: we make use of attribute grammars, look at some C-code implementation, and sketch also some code in a functional language. All three angles are basically equivalent. The focus here is on **straight-line code**. In other words, control-flow constructs, like conditionals and loops, are not covered right now. Those are translated making use of (conditional) jumps and labels. We will deal with those aspects later.

One way to describe the code generation is with an *attribute grammar*. So let's therefore fix a **context-free grammar first**, fixing the syntax, for which we later show appropriate semantic rules in the attribute grammar formalism.

As said, we focus first on straight-line code, there will be no control-flow constructs such as conditionals and the like. The atomic building blocks of straight-line code are assignments; the syntax we will use formalizes not (just) assignments of the form $x := e$ where e is a

side-effect free expression. The expressions of the grammar below allow assignments inside expressions, to make it more flexible and the code generation slightly more interesting. So the syntax allows expressions like `(x:=x+3)+4`. However, we need to be careful when allowing assignments inside expressions. We touched upon an issue in that context before before, in Section 9.4, when we gave an example of how p-code for an expression could look like (see Example 9.4.2). In that example, the expression was side-effect free, but for the current example, that's not the case. That expressions like `(x:=x+3)+4` make sense at all, the semantics of an assignment $x := e$ must be such that it results in a *value* and not in "nothing". In the corresponding type system, the type of the assignment $x := e$ is the same as the type of e (and not `void`). In the previous Section 9.4, we assumed the semantics of assignments to not give back a value (i.e., to be of type `void`), but here we want to do it otherwise. Consequently, the p-code in the example from the older section is *not* what would be generated here.

$$\begin{array}{rcl} exp_1 & \rightarrow & \textbf{id} := exp_2 \\ exp & \rightarrow & aexp \\ aexp & \rightarrow & aexp_2 + factor \\ aexp & \rightarrow & factor \\ factor & \rightarrow & (\ exp\ ) \\ factor & \rightarrow & \textbf{num} \\ factor & \rightarrow & \textbf{id} \end{array} \tag{9.2}$$

Example 9.5.1. Let's look at a small example:

$$(\texttt{x} := \texttt{x} + \texttt{3}) + \texttt{4} \tag{9.3}$$

The corresponding abstract syntax tree is shown in Figure 9.4. We will use it for illustration of code generation later.

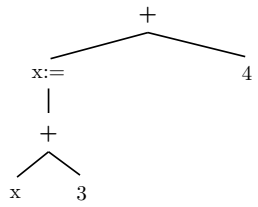


Figure 9.4: Abstract syntax tree for `(x:=x+3)+4`

□

9.5.1 Describing p-code generation with attribute grammars

In this section we treat **p-code as attribute** of the grammar symbols and nodes of the syntax trees.

The use of attribute grammars is perhaps more a conceptual picture. In practice, one may not formally or explicitly use a-grammars and corresponding tools in the *implementation* (though there exists tools for working with a-grammars). Remember that in many situations, the AST in a compiler is a "just" a data structure programmed inside the chosen

meta-language. For instance, in the compila language, most will have chosen a Java implementation making use of different abstract and concrete classes, perhaps making a visitor pattern and what not. Anyway, it's not in a format directly represented to be handled by an attribute-grammar tool (though also that is possible). Anyway, realizing the semantic rules we show in a-grammar format in a programming language format, operating on the AST tree data structure is not complex. In particular, since the attribute grammar is of a particularly simple format: it uses a **synthesized** attribute only (which is the simplest format). It works bottom-up or in a divide-and-conquer or compositional manner: the code of a compound statement consist of compiling the substatements and connecting the resulting translated code, with some additional commands. For expressions, the additional instructions are done at the end ("post-fix"), in more general situations, one encounters also pre-fix code (and sometimes even infix).

That captures the core of the compilation, it better be compositional: to compile a large program means, to break it down into pieces, compile smaller pieces and the put the compiled pieces together for the overall result.

The principle of **compositionality** or divide-and-conquer is perhaps so typical or natural for compilation in general, to appear as not even worth mentioning. That maybe so, but the principle applies only when ignoring *optimization*. Optimization breaks with the principle of compositionality, mostly. Taking two "optimized" pieces of generated code together in a divide-and-conquer manner will typically not result in an optimized overall piece of code. Optimization is done more globally, not compositionally wrt. the syntax structure of the program. The improvement may refer to the execution time or memory consumption (or even on the size of the code itself, which itself is not a semantic criterion, but the optimization must preserve the semantics, of course). The remarks here about compositionality of code generation and the non-compositionality of analysis and optimization is not particular for p-code generation. The same applies to 3AIC generation and actually to compilation in general. The compilation part is typically compositional and therefore efficient. Analysis and optimization(s) are done afterwards and depending on how much one invests afterwards in analysing the result and how aggressive the optimizations are, that part may no longer be efficient. By efficient I basically mean: linear (or at least polynomial) in the size of the input program (on average and on ordinary programs; if there are artificially and tailor-made examples where some worst-case complexity kicks in, one can live with that).

When saying, analysis and optimization is not compositional (unlike code generation), that probably should be understood as a qualified, not absolute statement. It's mostly not possible to invest in an absolutely global analysis, it would be too costly. It may be "compositional" in respecting the user-level syntax in that it does analyses each procedure individually, but tries not to make a global optimization across procedure body boundaries. Or even simpler, the optimization focuses on stretches of *straight-line code*. For instance, if one translates a conditional, there will be in the translation some jumps and labels, but those mark the boundaries of the optimization. In a way, the two branches of a conditional are optimized independently, in that sense the optimization is composition as far as the user-level syntax is concerned, and one does not attempt to see if *additional* gains could be achieve to analyze both branches "globally". These issues —analysis, optimization, and various levels of "globality" for that— will be relevant in the next chapter, where we

discuss the ultimate code generation, not intermediate code generation. Of course, a real compiler may use differerent optimizations in various phases of its compilation process.

The attribute grammar for intermediate code generation is actually rather simple and straightforward, and uses **one synthesized attribute** `pcode`. But as mentioned, the code generated here is for straight-line code only. See Table 9.1.

productions/grammar rules			semantic rules
exp_1	$\rightarrow$	**id** := exp_2	exp_1 .pcode = "lda"^**id**.strval ++ exp_2 .pcode ++ "stn"
exp	$\rightarrow$	$aexp$	exp .pcode = $aexp$ .pcode
$aexp_1$	$\rightarrow$	$aexp_2$ + $factor$	$aexp_1$ .pcode = $aexp_2$ .pcode ++ $factor$ .pcode ++ "adi"
$aexp$	$\rightarrow$	$factor$	$aexp$ .pcode = $factor$ .pcode
$factor$	$\rightarrow$	(exp)	$factor$ .pcode = exp .pcode
$factor$	$\rightarrow$	**num**	$factor$ .pcode = "ldc"^**num**.strval
$factor$	$\rightarrow$	**id**	$factor$ .pcode = "lod"^**num**.strval

Table 9.1: Attribute grammar

The op-codes are marked in red. We use ++ for "string" concatenation, joining "lists" of instructions and ^ to concat one single instruction. The generation is rather simple: the only attribute, containing the generated code, is purely synthesized (which is arguably the simplest form of AGs). It works purely bottom-up, divide and conquer. When are dealing with expressions only, the code generation works similarly as the *evaluation* of side-effect free expressions (which also works bottom-up). However, code generation works also when dealing with assignments (something that we did not do earlier in the atrribute grammar chapter, when doing expression evaluation). As discussed in the previous subsection, we see also the difference between l-values and r-values (`lda` and `lod`).

Remark 9.5.2 (Linearization)**.** Let's address another small point here. As mentioned, we are dealing with a *linear* IR: like 3AIC and other formats, p-code is a linear IR. It is a language consisting of a linear sequence of simple commands (and uses jumps and labels for control, even though those parts are currently not in the focus). The task of code generation (if one assume that one deals with control-structures as well) it to translate the non-linear tree structure into a linear one (using jumps and labels). So, that may be called "linearization". Since currently we don't focus on the control-structures, the task is to translate an already linear language ("straight-line code") to another linear arrangement, the linear p-code. We do so in the AG, assuming operations like ^ and $+\!\!+$. The respesent appending an element to a list resp. concatenating two lists. However, strictly speaking $+\!\!+$ is a binary operation. We wrote in the semantic rules of the AG things like $l_1 +\!\!+ l_2 +\!\!+ l_3$. We did not say how to "think" of that (like to parse it mentally). Is that left or right associative? Or do we mean that the reader understands that it does not really matter, as list concatenation is associative and we mean the resulting overall list, obviously. Sure, it should be clear. Note also, that $+\!\!+$ is understood as separating

two pieces of code from each other (one can think "newline" in code examples). Later, we show an implementation in a functional language, we use the constructor `Seq` for that (for sequential composition). However, we don't implement that as concatenation of lists but as a simple constructor. Consequently, the result of that translation (which corrresponds to the AG here) is *not* technically linear, it's still a tree (even one of quite simple structure). Therefore, in a last step, one needs to flatten out the tree to a ultimate linear list. See Listings 9.11 and 9.12.

Why does one do it two stages, not one? Well, it may be more efficient that way: concatenating lists "on the fly" in functional languages is typically not a tail-recursive procedure and thus not altogether cheap. So one may be better off by first doing another tree-like structure, to be flattened out afterward. It's actually a common technique. And furthermore, if we would right now *also* consider conditionals and loops, etc. it's harder to find the ultimate linear sequence of commands while processing then abstract syntax. Also for that reason, one might be better off to first generate pieces of the code that are afterwards glued together in a linear arrangement. Linearization of a similar form is done for instance in the compiler described in [1] as part of the so-called **canonization** phase, massaging the intermediate code (there some 3AIC) to get get ready for the last phase of generating platform dependent machine code.

But apart from those fine points, the implementation from Listing 9.5.1 later reflects truthfully the AG here. □

Example 9.5.3. Let's revisit the code snippet from Example 9.5.1, resp. equation (9.3). Figure 9.5 shows the attributed tree. The boxed contain the attribute values, i.e., the generated code. The resulting code in the root node is shown in Listing 9.8.

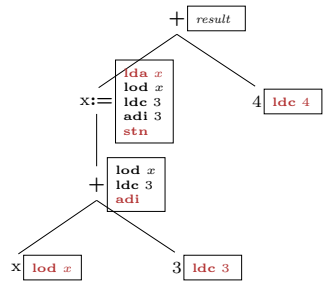


Figure 9.5: Attributed tree

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi     ;  +
```

Listing 9.8: P-code for `(x := x + 3) + 4`

Note: *here* `x:=x+3` has a side-effect *and a return value*, as in C and other languages. Therefore, the store command used to write back the value is **stn**, "store non-destructively". This command takes the top element, stores it at address represented by 2nd slot on top,

discard the address, but not the top-value. Thats's different from **sto** used earlier, which stores analogously and forget the result, i.e., pop also the result value off the stack.

The issue of the semantics of an assignment has been mentioned earlier: does it give back a result or not. Before, the code shown in an example was correct under the assumption no value is "returned". Here, we interpret it different, in accordance with languages like C or Java. Thus, we have to use the command `stn` instead of `sto` from before. □

9.5.2 Implementation in a functional language

In the following we show how the intermediate code generation resp. the AG can be implemented straightforwardly in a functional language. Later, we will see also how the code looks in C, which is also straightforward (though I believe the functional code is more concise).

We start defining the two syntaxes of the two languages, the source code and the target code. There are more or less one-to-one transcripts of the grammars we have seen.

```
type symbol = string

type expr =
  | Var of symbol
  | Num  of int
  | Plus of expr * expr
  | Assign of symbol * expr
```

Listing 9.9: Data-structure for source code expression ASTs

```
  | LOD of symbol
  | LDA of symbol
  | ADI
  | STN
  | STO

type tree = Oneline of instr
  | Seq of tree * tree

type program = instr list
```

Listing 9.10: Data-structures for target intermendiate code (p-code)

Side remark 9.5.4 (Symbols)**.** For simplicity, the code in Listing 9.9 and 9.10 uses strings to refer to variables and temporaries. More realistically, a real compiler would avoid to work with strings for very long, as this is not very efficient (like requiring string comparisons). Instead, strings would be represented differently, and perhaps the symbol table would be involved. But here, for illustrating intermediate code generation, we don't do those smarter ways of handling symbols.

In the target syntax, there are two "stages": a program is a linear list of instructions, but there is also the notion of "tree": the leaves of the trees are "one-line" instructions and trees can be combined using sequential composition. Consequently, the translation (on the next slide) will also have **2 stages:** the first one (which is the interesting one) generates a tree, and the second one flattens out the tree or "combs it" into a list. See also the types of the involved functions in Listing 9.11.

```
val to_tree: Astexprassign.expr -> Pcode.tree

val linearize: Pcode.tree ->  Pcode.program

val to_program: Astexprassign.expr -> Pcode.program
```

Listing 9.11: Types of the translation functions

The translation itself is shown in Listing 9.12. In particular, the function `to_tree` is a rather faithfull representation of the attribute grammar we have seen earlier.

```
open Astexprassign;;
open Pcode;;

let rec to_tree (e: expr) =
  match e with
  | Var s -> (Oneline (LOD s))
  | Num n  -> (Oneline (LDC n))
  | Plus (e1,e2) ->
      Seq (to_tree e1 ,
           Seq(to_tree e2, Oneline ADI))
  | Assign (x, e) ->
      Seq (Oneline (LDA x),
           Seq( to_tree e, Oneline STN))

let rec linearize (t: tree) : program =
  match t with
    Oneline i -> [i]
  | Seq (t1, t2) -> (linearize t1) @ (linearize t2);; (* list concat *)

let to_program e = linearize (to_tree e);;
```

Listing 9.12: Data-structures for target intermendiate code (p-code)

The code makes more visible, that operations like ++ used in the AG are binary, the AG generates a tree rather than a sequence. Nonetheless, flattening out the tree in a second step (`linearize`) is child's play. As mentioned earlier, in connection with that AG: it would be straightforward not to have these 2 stages: instead of using `Seq` for doing the trees first, one could use directly list-append.

9.5.3 Source language AST data in C

Next we do the "same" implementation in C. We start by showing a possible way to represent ASTs. We have seens similar representations in earlier chapters. We have also seen ways to represent such trees in Java where we operated with concrete classes as beeing subclasses of abstract classes. Here, the data structure uses enumeration types and structs (see Listing 9.13). It corresponds to the implementation using inductive data types (in ocaml) from Listing 9.9.

```
typedef enum {Plus,Assign} Optype;
typedef enum {OpKind,ConstKind,IdKind} NodeKind;
typedef struct streenode {
  NodeKind kind;
  Optype op;        /* used with OpKind */
  struct streenode *lchild, *rchild;
  int val           /* used with ConstKind */
  char * strval     /* used for identifiers and numbers */
} STreenode;
typedef STreenode *SyntaxTree;
```

Listing 9.13: AST in C (for expressions with assignments)

Figure 9.6 shows schematically a small sample AST. The table summarizes the "attributes" per node.

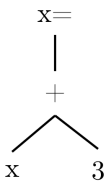


Figure 9.6: Sample AST

node	kind	op	val	strval
x:=	OpKind	assign		
+	OpKind	Plus		
x	IdKind			"x"
3	ConstKind		3	

Table 9.2: "Attributes"

This sketch of a code skeleton from Listing 9.14 basically says: the code generation is a recursive procedure, traversing a given abstract syntax tree.

```
procedure genCode(T: treenode)
begin
 if   T ≠ nil
 then
  ``generate code to prepare for code for  left child''  // prefix
  genCode (left child of T);  // prefix ops
  ``generate code to prepare for code for right child''  //infix
   genCode (right child of T); // infix ops
  ``generate code to implement action(s) for T''  //postfix
end;
```

Listing 9.14: Code-generation via tree traversal (schematic)

During traversal, it involves prefix-actions, post-fix actions and maybe even infix-actions (see Figure 9.7). By actions I mean generating or *emitting* p-code commands. Looking at the functional code we can see that there was no code generated in infix-position, so we can expect to see no such thing in the C-code as well. The sketched skeleton just shows the general shape, there may be other situations more complex that the ASTs covered here that would call for infix code. We, at least don't make use of it here. See Listing 9.15 later for more complete code for `codeGen` for expressions.

The code generation works in principle the same as in the functional implementation (and the AG), of course. In the functional implementation from before from Listing 9.12, we have choosen *not* to emit **strings** already. Instead we have chosen to construct an element of a data structure representing the instructions of the p-code (we called the type `instr`).

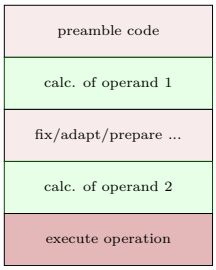


Figure 9.7: Schematic shape of the recursive code generation

Given the fact that we are not yet at the "real" code level, but at an intermediate stage, generating a data structure is more realistic and better than generating a string. A string would have to be parsed again etc., and operating on strings is always more error prone (typos) than operating on constructors of a data structure.

Not that reparsing strings would be hard. Also for debugging reasons a compiler could have the option to emit a "pretty-printed" version of the intermediate code (or some other external exchange format), but a well-designed *internal representation* is a more dignified and realistic way of handing things over to the next stage.

In the functional implementation, we turned the abstract syntax tree into a linear structure (a list) in a two-stage process (cf. also the interface from Listing 9.11). Working with a (functional) list data structure as target, doing it like that is more efficient; functional list concatenation, which would be used in a one-stage approach, is not very efficient.

```
void genCode (SyntaxTree t) {
  char codestr[CODESIZE];
  /* CODESIZE = max length of one line of p-code */
  if (t!=NULL) {
    switch (t->kind {
        case OpKind:
          switch (t->op) {
          case Plus:
            genCode(t->lchild);
            genCode(t->rchild);
            emitCode("adi");
            break;
          case Assign:
            sprintf(codestr,"%s %s", "lda",t->strval);
            emit(codestring);
            getCode(t->lchild);
            emitCode("stn");
            break;
          default:
            emitCode("Error");
            break;
          };
          break;
          case ConstKind:
            sprintf(codestr, "%s %s", "ldc", t->strval);
            emitCode(codestr);
            break;
          case IdKind:
            sprintf(codestr, "%s %s", "lod",t->strval);
            emitCode(codestr);
            break;
          default:
            emitCode("Error");
            break;
      };
  };
}
```

Listing 9.15: `GenCode` for expressions / assignments in C

9.6 Generation of three-address intermediate code

This section does the analogous thing we have done for p-code (one-address code) in Section 9.5. We start by showing how resulting intermediate could look like, using the same factorial example from before. When covering p-code, we did not talk about control-flow constructs. We do the same here, focusing on straight-line code again. Treatment of control-flow will be done in Secion 9.9: Indeed, there is not much difference between 3AIC and p-code as far as the control-flow is concerned: both formats have to use conditional jumps to translate conditionals, loops, and the like. Section .

We have seen examples of translations from source code to three address intermediate code. See for instance Example 9.3.2, in particular the source code and 3AIC target code from Listing 9.3 and 9.4. In this section, as we did for the p-code, we focus on straight-line code, though the example shows also how conditionals and loops are treated (which we cover later). As far as the treatment for the latter constructs is concerned, the p-code generation and the 3AIC code generation works analogously anyway. In the translated target code for the factorial, we see also here labelling commands (pseudo-instructions) and (conditional) jumps, as in the target code when translated to p-code.

9.6.1 Implementation in a functional language

We do the same as for the p-code and show how to realize the code generation in some functional language (ocaml). The source language, expressions in the abstract syntax tree and assignments, are unchanged (the **grammar** was showno in equation (9.2). Remember also the data structures for source code expressions from Listing 9.9.

```
type mem =
    Var of symbol
  | Temp of symbol
  | Addr of symbol  (* &x *)
type operand = Const of int
  | Mem  of mem
type cond = Bool of operand
  | Not of operand
  | Eq of operand * operand
  | Leq of operand * operand
  | Le of operand * operand

type rhs = Plus of operand * operand
  | Times of operand * operand
  | Id of operand

type instr =
    Read of symbol
  | Write of symbol
  | Lab of symbol      (* pseudo instruction *)
  | Assign of symbol * rhs
  | AssignRI of operand * operand * operand (* a    := b[i] *)
  | AssignLI of operand * operand * operand (* a[i] := b *)
  | BranchComp  of cond * label
  | Halt
  | Nop

type tree = Oneline of instr
  | Seq of tree * tree
type program = instr list


(* Branches are not so clear. I take inspiration first from ASU. It seems
that Louden has the TAC if_false t goto L. The Dragonbook allows actually
more complex structure, namely comparisons. However, two-armed branches are
```

Listing 9.16: Data-structures for target intermediate code (3AIC)

The target language is represented by another data structure, shown in Listing 9.16.

That can be compared to the data structures from Listing 9.10 for p-code, from earlier. One can also see: the 3AIC data structure covers more than we (currently) actually need. There is branching and labels. There is also something that deals with using arrays in assignment. More complex data structures like array accesses and indexed access will be coverered later as well, but not right now.

The data structure for the target language does the same two layers we used for the p-code. One "tree" representation that connects single-line instructions using `Seq`, and a linear list of instructions as the final representation.

```
let rec to_tree (e: expr) : tree * temp =
  match e with
    Var s ->  (Oneline Nop, s)
  | Num i ->  (Oneline Nop, string_of_int i)
  | Ast.Plus (e1,e2) ->
      (match (to_tree e1, to_tree e2) with
        ((c1,t1), (c2,t2)) ->
          let t = newtemp() in
          (Seq(Seq(c1,c2),
               Oneline (
               Assign (t,
                       Plus(Mem(Temp(t1)),Mem(Temp(t2))))))),
           t))
  | Ast.Assign (s',e') ->
      let (c,t2) = to_tree(e')
      in  (Seq(c,
               Oneline (Assign(s',
                               Id(Mem(Temp(t2)))))),
           t2)
```

Listing 9.17: Translation to three-address code

For the code generation, we focus on the translation of the part we are currently interested in, assignments and expressions, leaving out the other complications. We see the generation of new temporaries using a function `newtemp`. The implementation of that is not shown, but is easy enough (simply using a counter that generates a new number at each invocation and returning a corresponding temporary). Strictly speaking, such a counter is not purely functional. That's not a problem, most functional languages are not purely declarative, and one can implement such a generating function and other imperative things. Later, we look at a corresponding AG. Normally, an attribute grammar (as a theoretical construct) is purely declarative or functional, which means without side-effects. Still, we will allow ourselves in the AG a function like `newtemp` for convenience.

In principle, one could do a fully functional representation (here in the code as well as in the AG later), simply adding an additional argument, for instance a integer counter that is appropriately handed over. That does not add to the clarity to the code, so a generator like `newtemp` is more concise, it would seem.

An interesting aspect of the code generator is its type, resp. its **return type.** It returns, obviously, 3AIC, more precisely a "tree" of 3AIC instructions. However, it *also* returns an element of type `temp`. This is needed, because in order to generate code for compound statements, **one needs to know where to find the results of the translation** of the sub-expressions. That can be seen, for instance, in the case for addition.

The two recursive calls on the subexpressions of the addition give back a tuple each, i.e., one has two pairs of information; see the correponding match-expression in the code. The resulting code is constructed as trees, and the result is given back in temporaries t_1 and t_2 (or `t1` and `t2` in the code). Then the last 3AIC line generated in the addition-case

is $t := t_1 + t_2$, where t is a new temporary, and the function return the pair of the code together with this freshly generated t.

9.6.2 Describing 3AIC generation using attribute grammars

Like the one for generating p-code, the attribute grammar relies solely on *synthesized* attributes. For the semantics of executing expressions and assignment, we assume that, besides the side effect, also a value is returned. As a consequence, the attribute uses **two** attributes (for the p-code, there had been only one). The functional implementation, which is a rather faithful realization of the attribute grammar, likewise give back a pair of results. The two attributes use are `tacode`, which contains the instructions. As before, we see that as string (potentially empty). The attribute `name` represents the name of the temporary varaible, where result resides.[5] We assume, as before, a *left-to-right* evaluation of expressions. The resulting attribute grammar is shown in Table 9.3.

productions/grammar rules			semantic rules		
exp_1	$\rightarrow$	**id** := exp_2	exp_1 .name	=	exp_2 .name
			exp_1 .tacode	=	exp_2 .tacode ++ **id**.strval^"="^ exp_2 .name
exp	$\rightarrow$	$aexp$	exp .name	=	$aexp$ .name
			exp .tacode	=	$aexp$ .tacode
$aexp_1$	$\rightarrow$	$aexp_2$ + $factor$	$aexp_1$ .name	=	$newtemp()$
			$aexp_1$ .tacode	=	$aexp_2$ .tacode ++ $factor$ .tacode ++ $aexp_1$ .name^"="^ $aexp_2$ .name^ "+"^ $factor$ .name
$aexp$	$\rightarrow$	$factor$	$aexp$ .name	=	$factor$ .name
			$aexp$ .tacode	=	$factor$ .tacode
$factor$	$\rightarrow$	(exp)	$factor$ .name	=	exp .name
			$factor$ .tacode	=	exp .tacode
$factor$	$\rightarrow$	**num**	$factor$ .name	=	**num**.strval
			$factor$ .tacode	=	""
$factor$	$\rightarrow$	**id**	$factor$ .name	=	**num**.strval
			$factor$ .tacode	=	""

Table 9.3: Attribute grammar for 3AI code generation

As mentioned, we allow ourselves here a function *newtemp*() to generate a new temporary in the case of addition, even if, super-strictly speaking, that's not covered by AGs which are introduced as declarative, side-effect free formalism. But doing it purely functional (which is possible) would not add to understanding how 3AIC is generated.

9.6.3 3AIC generation in a C-like or Java-like language

Let's look also at code generation in a C-like notation. The functionality `genCodeTA` from Listing 9.18 is assumed added as **method** to the nodes of that abstract syntax tree.

[5] In the p-code, the result of evaluating expression (also assignments) ends up in the stack (at the top). Thus, one does not need to capture it in an attribute.

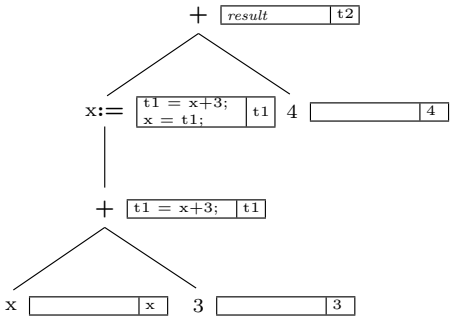


Figure 9.8: Attributed tree (x:=x+3) + 4

```
switch kind {
  case OpKind:
    switch op {
      case Plus: {
        tempname = new temorary name;
        varname_1 = recursive call on left subtree;
        varname_2 = recursive call on right subtree;
        emit ("tempname = varname_1 + varname_2");
        return (tempname);}
      case Assign: {
        varname = id. for variable on lhs (in the node);
        varname 1 = recursive call in left subtree;
        emit ("varname = opname");
        return (varname);}
      }
  case ConstKind; { return (constant-string);} // emit nothing
  case IdKind:  { return (identifier);}        // emit nothing
}
```

Listing 9.18: Translation to three-address code

The code "returns" in a way the two attributes mentioned earlier. The **name** of the variable, a *temporary*, is officially returned. The code is produced by via the `emit`-function. Note, there is *postfix* emission only (in the shown cases).

In an object-oriented language like Java or C++, it's also possible to add `genCode` as *method* to the nodes of the AST. For example, define an abstract method `String genCodeTA()` in the `Exp`-class (or-`Node`, resp. in all AST nodes where it's needed.

```
String genCodeTA() { String s1,s2; String t = NewTemp();
  s1 = left.GenCodeTA();
  s2 = right.GenCodeTA();
  emit (t + "="  + s1 + op + s2);
  return t
}
```

Listing 9.19: Code generation with AST methods

Example 9.6.1. Let's apply the code generation to the expression

$$(\mathtt{x} := \mathtt{x} + 3) + 4\ .$$

We used the same expression before in Example 9.5.3 for the corresponding code generation in p-code. For 3AIC, the attributed tree is shown in Figure 9.8.

The generated code is shown in Listing 9.20, where the result of the calcuation can be found in the temporary `t2`.

```
t1 = x + 3
x  = 1
t2 = t1 + 4
```

Listing 9.20: 3AIC code for `(x:=x+3)+4`

□

9.7 From P-code to 3A-code and back

In this intermezzo we shortly have a look on how to translate back and forth between the two different intermediate code formats, 1-address-code and 3AIC. We do that mainly to touch upon two concepts, **macro-expansion** and **static simulation**. The first is one rather straightforward, the static simulation is a more complex topic.

Apart from the fact that those mentioned concepts are interesting also in contexts different from the one where they are discussing here, one may still ask: why would one want to translate 1AIC to 3AIC and back (beyond using the translations as illustrating some concepts)? Well, notions of 1AC and 3AC exist also independent from their use as *intermediate* code. In particular, hardware may offer an instruction set in 3A-format, or at least partly in 3A-format (or 2A-format). 1A-hardware, though, is non-existant (there had been attempts for that in the past). So, if one has an intermediate representation like the p-code or 1AIC as presented here, then generating code for a 3AC hardware faces problems like those discussed here. Final code generation faces *additional* problems like platform-dependent optimization, and register allocation, which will not enter the picture in this section. For the ultimate code generation, we will probably translate from 3AIC to 2AC machine code, which is not directly covered in this section here, but anyway, our focus later will be on register allocation.

9.7.1 P-code to 3AIC: static simulation

In this section we discuss something called *static simulation*. It's not discussed in much depth, more or less just *illustrated* by transforming p-code to 3AIC. Focusing on basically expressions resp. straight-line code, that's a pretty straightforward task anyway.

Cf. also the concept of *basic blocks* (or elementary blocks), which are blocks of code or intermediate code without branching or other control-flow complications like jumps, conditional jumps etc. The are considered as basic building block for static/semantic analyses.[6] Basic blocks, stretches of straight-line code are the nodes in *control-flow graphs*.

In the section we show two directions of intermediate code translations: from p-code two 3AIC here, and vice versa in Section 9.7.2. The reverse direction later, from 3AIC to p-code can be done quite trivially, by *macro expansion*. That's a technique not based on

[6] We will encounter control-flow graphs and elementary block in the chapter for code generation. But the concept can (and is) equally applied to intermediate codes like 3AIC or p-code.

static simulation or similar approaches, it's simply replaces syntactically each line of, here, 3AIC, by (typically more than) one line of p-code, preserving the behavior.

We won't show how to translate all of the p-language or all of the 3AIC language, we focus on straight-line code; conceptually sequences of assignments. Other parts, like jumps and labels don't need much translation, since they are analogous in both languages (though the commands are called differently).

So, how to do a translation of p-code to 3AIC? The difference between the two formats is that p-code operates on the **stack** which leaves the needed temporary memory implicit, in that the different stack contents have no explicit names. But that's what needed for 3AIC, the "names" being the temporaries.

Now, given an straight-line p-code sequence, the translation traverses the code, i.e, the list of instructions from beginning to end. That can be seen as "simulation", conceptually at least. Indeed, the implementation makes use *use* of an actual stack, when stepping through the p-code.

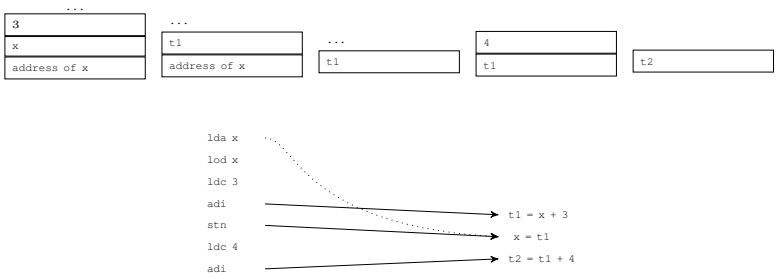


Figure 9.9: `x:=(x+3)+4` (from P-code ⇒ 3AIC)

Figure 9.9 illustrates it on a simple example `x := (x+3) + 4` (which we have seen before). The code on the top of the left-hand side is the "source" code, the p-code instructions. The right-hand side shows the evolution of the abstract p-code machine, when executing the p-code on the left. In particular, the stack as the crucial part is shown in its evolution, not after every single line having been executed, but at crucial intermediate stages. One such stage is after having done `adi`, for instance the first such instance. As discussed, the stack machine uses the stack for intermediate results, that's exactly what happens when executing `adi` (or similar operations): the operands are popped off the stack, and the intermediate result is stored on the stack ("push"). Without stack, the 3AIC needs to store that intermediate result somewhere else, and that's of course a (new) temporary. Note also: the semantics of the abstract syntax is assumed to be that an assignment (like `x := x+3` in the example) gives back a value, like on C or Java. That is reflected in the p-code by using `stn`, the *non-destructive* storing, as discussed earlier. In the translation to 3AIC, the right-hand side is stored in `t1`, and that is used in the last line `t2 := t1 + 3`.

9.7.2 P-code ⇐ 3AIC: macro expansion

For the reverse direction, we also focus on illustrating the general technique, restricting ourselves to straight-line code again. The direction from 3AIC to p-code is simpler, at least when doing it in a simplistic way. It does not need any static simulation of the architecture, i.e., it can work simply on the **syntactic** structure of the input program. It simply expands each line by a corresponding sequence of p-code instructions.

Listing 9.21 shows how the expansion from one line of 3AIC assignments involving adding up the source arguments.

```
lda a
lod b;    // or ``ldc b'' if b is a const
lod c:    // or ``ldc c'' if c is a const
adi
sto
```

Listing 9.21: Macro for general 3AIC instruction: `a := b + c`

Let's illustrate it on a small example, actually one we have seen before.

Example 9.7.1 (`(x:=x+3)+4`). The user level code in this example is

$$(\mathtt{x} := \mathtt{x} + 3) + 4$$

from equation (9.3) again. We had earlier translated the line into p-code as well as into 3AIC (see Listing 9.8 and Listing 9.20). The two resulting intermediate code programs are repeated here in Listings 9.22 and 9.24.

```
t1 = x + 3
x  = t1
t2 = t1 + 4
```

Listing 9.22: Source 3AIC (seen before)

```
;--- t1 = x + 3
lda t1
lod x
ldc 3
adi
sto
;--- x = t1
lda x
lod t1
sto
;--- t2 = t1 + 4
lda t2
lod t1
ldc 4
adi
sto
```

Listing 9.23: P-code via 3AIC by macro expansion

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi     ;  +
```

Listing 9.24: Direct p-code (seen before)

□

The p-code from Listing 9.23 is generated by line-by-line **macro-expansion** from the 3AIC of Listing 9.22. Clearly, the directly translated code is quite shorter (and more efficient), it's 7 instructions vs. 13 instructions. One important factor in that "loss" in the indirect translation is that the macro-expansion is "brainless". That's makes the expansion simple and efficient, but at the price is that the resulting code is not efficient when being executed. We will, in the following at least hint how to do it better. In general, however, generating efficiently non-efficient (but correct) code that is afterwards optimized is not

per se a bad idea. That is commonplace in many compilers (even if compilers might not compiler back-and-forth 1AIC and 3AIC). Anyway, the "better" translation we will look at improves on one piece of inefficiency (in the example). The 3AIC contains a line `x = t1`. After that `x` and `t1` contain obviously the same value. The macro expansion "mindlessly" expands this line, even though one does not need to have two copies of the value around. More generally, the translation does not keep track of which values are stored where, it works purely line-by-line and syntactically. That can be improved, in "static-simulation" style.

It can be done better (static simulation)

As we have seen, the macro expansion leads to inefficient code, for once it is quite longer. A more deeper and more serious (but related) deficiency is, that it uses **too many temporaries.** That's could lead to a higher memory usage, but also, and that is probably worse, too much *memory traffic*, potentially needlessly copying values to different places involving memory. If that would be the result of the translation, that would be seriously unwelcome. Of course, it's perhaps not too common to design the intermediate code generator to first produce p-code and afterwards 3AIC as another intermediate code, paying a hefty efficiency fee for this detour. But then again, intermediate codes like Java byte code resemble p-code. If one produces executable code (for instance also JIT), the compiler may take this detour.

Besides that, intermediate code is not the end of story. Code can and should be **optimized**, the intermediate code or the ultimately generated code (or both). So, indeed, if one had the more compact 3AIC directly, not via macro-expansion, also that should be optimized. The less efficient, indirect one just has more potential for optimization. So with good optimization in place, it may not even make much difference which version of the 3AIC one has at some point. In particular concerning the generous use of temporaries in the macro-expanded code: one problem we have not looked into yet is **register allocation.** We will do that to some extent for *code* generation later (not *intermediate* code generation). That's a task crucial for efficiency, and the temporaries of the intermediate code are generate freely anyway, without yet loosing sleep over the fact that too many temporaries might come at a cost. Later phases, including register allocation, will deal with with the problem.

What we do here is something else: we sketch how to do the translation more cleverly than a plain macro expansion, again in a form of **static simulation**. Instead of "brainlessly" translating the given linear structure, the 3AIC, into another *linear* structure (p-code), we static simulation steps through the code and transforms it into a *fancier* structure, a *tree*. In effect, in the example from before, it *reconstructs* the abstract syntax tree that resulted in the p-code. In more general settings, it's unrealistic to obtain the original syntax tree (or one possible original syntax tree) from a linear intermediate code. So the message is not that using static simulation here allows do recompile intermediate code to source code. But in the small example here it effectively does. The message is more that static simulation use some data structures, collecting information while stepping through the code.

The information or data structure here is a **tree** labeled or attributed with two pieces of information the **operator** to produce the corresponding value and the name of the **variables/temporaries** containing the result. That effectively, in the example, is nothing else than the abstract syntax tree (resp. the **attributed** syntax tree).

Example 9.7.2. Using the (only roughly sketched) idea on the code from Listing 9.22, we obtain the tree representation from Figure 9.10. Note, the instruction `x = t1` from 3AIC

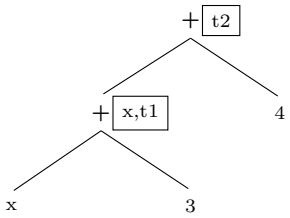


Figure 9.10: Tree

does *not* lead to more nodes in the tree, it recycles the node it has already introduced.

With this tree (which corresponds to the AST), the code generator can work analogously to the one for the *direct* translation from source code to p-code. Consequently, the result for the indirect translation is *identical* to the directly generated p-code, at least in the example (see Listing 9.24). □

9.8 More complex data types

Next we drop one of the simplifications done so far, concerning the involved *data*. We have a look at how to lift the other simplification, lack of control-flow commands in Section 9.9 later. As far as the data is concerned, we have treated only variables (and temporaries) for *simple* data types, but not compound ones like arrays, records, etc. For those some additional *address calculations* needed, and that may (or not) be supported by the intermediate code. Also, we have not looked at **reference** data (pointers). To deal with that adequately and efficiently, intermediate languages support additional ways to access data, i.e., additional **addressing modes**. A taste of that we have seen in the p-code: a variable can be loaded in two different ways, depending on whether the variable is used as l-value or r-value. The two commands are `lod` and `lda`, load the variable's value or load the variable's address.

Table 9.4 lists new addressing modes both for 3AIC and p-code. Note that for p-code, we have already introduced `lda x`, obtaining the address of its argument (which corresponds to `&x`).

The concepts underlying the commands here are typically also supported by standard hardware. There may be special registers for *indexed* access, to make that form of access fast. Indexed access (here in p-code) is an access which has *three* arguments: the address of some place (in memory), and an off-set, given by 2 numbers, say n and s. The offset

3AIC		p-code	
`&x`	*address* of `x` (not for temporaries)	`ind i`	*indirect load*
`*t`	*indirectly* via `t`	`ixa a`	*indexed address*

Table 9.4: Basic new addressing modes

refers to the n^{th} slot away from the base address, and s is a scale factor, specifying the size of each slot. That should remind us to the way that arrays are laid out in memory; we had discussed that earlier. Indeed, HW-supported indexed access is one important reason, why arrays are a very efficient data structure. We will illustrate the new constructions on arrays (but also records) in the following.

In 3AIC, we don't have (currently) indexed addressing, we have a C-like situation, with access to the addresses of variables. The `&x` operation corresponds to the `lda` instruction in p-code.

Loading **indirectly** (in 3AIC and p-code) means: do not load the content of the variable (nor load its address): load the content of the variable (or here the temporary), interpret the loaded value as address, and *then*, load from there. Similarly for storing, when using `*t` on the left-hand side of a 3AIC assignments.

9.8.1 Array access

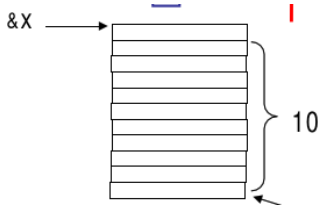
Let's use the new instructions in 3AIC (and afterwards in p-code) to access **array**-data, which requires some **address calculations**. Let's use concretely the following assignment (in C-notation):

$$\texttt{x}[10] = 2 \tag{9.4}$$

The update of the slot with number 10 in the array `a` can be done by simple address calculations ("+10") in combination with the new addressing modes.

```
t1  = &x + 10
*t1 = 2
```

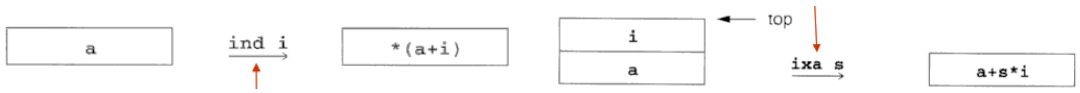
Listing 9.25: Address calculations for `a[10]=2` (3AIC)



One thing one should not forget. Earlier, we sketched how 3-address code can be represented inside a compiler. Now, with alternative **address modes,** the 3-address code **data structure** (earlier represented for example as quadruple), needs to be extended: information concerning the address mode needs to be incorporated.

The compilation is straightforward. The code also shows, that (at least in our 3AIC) there is no *indexed access*. The off-set, in the example 10 is calculated by 3AIC instructions in a form of "**pointer arithmetic**".

Let's redo the example in **p-code**, as well. There, the translation will make use of an indexed access command `ixa`, which is a tailor-made command for the required kind of address calculation.



(a) `ind` (b) `ixa`

Figure 9.11: Addressing modes in p-code

The effect of the two introduced commands `ixa` and `ind` is shown in the transitions Figure 9.11, stepping from the stack content on the left-hand side to the stack on the right-hand side. The two commands correspond to a situation, where a array expression is written-to (`ind`) resp. read-from (`ixa`).

- `ixa i`: integer *scale* factor (here factor 1)

```
lda x
ldc 10
ixa 1     // factor 1
ldc 2
sto
```

Listing 9.26: Address calculation for `a[10]=2` (p-code)

In the two pictures, the `a` is mnemonic for a value representing an address. In the code example: The `ixa` command expects two argument on the stack (and has as third argument the scale factor as part of the command. To make use of the command, we first load the *address* of `x` loaded and afterwards constant `10`. Executing then the `ixa 1` command yields does the calculation in the box, which is intended as address calculation. So the result of that calculation is (intended as) an address again. To that address, the constant `2` is stored (and the values discarded from the stack: `sto` is the "destructive" write).

Let's look a bit more into array accesses and corresponding address calculations. In particular, remember that arrays are typically treated as reference data, I.e., at the source-code level, an array-typed variable does not contain the content of the array itself, but a reference to the array.

```
int a[SIZE]; int i,j;
a[i+1] = a[j*2] + 3;
```

Listing 9.27: More complex array access in C

Listing 9.27 is not just more complex than the previous array example in that the slots in the array require a computation. It also uses a read as well as a write access to an array. The offset to an entry in the array in general needs to be calculated with a **scale factor**, which depends on the size resp. the type of the array elements. For instance, `a[i+1]` (with C-style array implementation)[7], the calculation is done by

```
a + (i+1) * sizeof(int)
```

[7] In C, arrays start at a 0-offset as the first array index is 0. Details may differ in other languages.

where a here is meant to stand *directly* for the base address.

One possible was is to assume 2 additional 3AIC instructions, resp. to design the intermediate code in such a way that it support such instruction. After all *intermediate code* is just a step towards machine instructions, not programs use instruction set of a particular HW. And perhaps the ultimate hardware supports some complex indexing modes.

As **2 new instructions**[8] we assume the following

```
t2 = a[t1] ; fetch value of array element

a[t2] = t1 ; assign to the address of an array element
```

Listing 9.28: Instructions for array accesses in 3AI code

With those, we can translated the array access as follows:

```
a[i+1] = a[j*2] + 3;
```

Listing 9.29: Source code (repeated)

```
t1    = j * 2
t2    = a[t1]
t3    = t2 + 3
t4    = i + 1
a[t4] = t3
```

Listing 9.30: 3ACI

We have mentioned that IC is an intermediate representation that may be more or less close to actual machine code. It's a design decision, and there are trade-offs either way. Like in this case: obviously it's (slightly) easier to translate array accesses to a 3AIC which offers such array accesses itself. It's, however, not too big a step to do the translation without this extra luxury. In the following we see how to do exactly that, without those array-accesses at the IC level (both for 3AIC as well as for P-code). That's done by **macro expansion,** something that we touched upon earlier. The fact that one can "expand away" the extra commands shows there are no real complications either way (with or without that extra expressiveness).

One interesting aspect, though, is the use of the helper-function `elem_size`. Note that this depends on the **type** of the data structure (the elements of the array). It may also depend on the platform, which means, the function `elem_size` is (at the point of intermediate code generation) conceptually not yet available, but must provided and used when generating platform-dependent code. As similar "trick" we will see soon when compiling record-accesses (in the form of a function `field_offset`).

As a side remark: syntactic constructs that can be expressed in that easy way, by forms of macro-expansion, are sometimes also called *syntactic sugar*.

```
t3 = t1 * elem_size(a)
t4 = &a + t3
t2 = *t4
```

Listing 9.31: Expanding `t2=a(t1)`

```
t3  = t2 * elem_size(a)
t4  = &a + t3
*t4 = t1
```

Listing 9.32: Expanding `a(t2)=t1`

The macro-expanded result for `a[i+1] = a[j*2] + 3` then looks as follows

[8] Thuse are still in 3AIC format. Apart from the "readable" notation, it's just two op-codes, say `=[]` and `[]=`.

```
t1   = j * 2
t2   = t1 * elem_size(a)
t3   = &a + t2
t4   = *t3
t5   = t4 +3
t6   = i + 1
t7   = t6 * elem_size(a)
t8   = &a + t7
*t8 = t5
```

Listing 9.33: Macro-expanded version in 3AIC

Let's also show how to expand the two array access versions to p-code.

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

Listing 9.34: Expanding `t2=a(t1)` in p-code

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

Listing 9.35: Expanding `a(t2)=t1` in p-code

That results in the following translation.

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

Listing 9.36: Macro-expanded version of `a[i+1] = a[j*2] + 3` in p-code

Extending grammar & data structures

Let's extend the earlier grammar from equation (9.2) to cover also array accesses.

$$\begin{array}{rcl} exp & \rightarrow & subs := exp_2 \mid aexp \\ aexp & \rightarrow & aexp + factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \mathbf{num} \mid subs \\ subs & \rightarrow & \mathbf{id} \mid \mathbf{id}\,[\, exp\,] \end{array} \tag{9.5}$$

Extending the language (here with arrays) means extending the AST. That means we have to extend the tree definition from Listing 9.13. Actually, the extension is quite small: Compared to the the tree structure from Listing 9.13, the only addition is a new "code" `Sub` in the enumeration `Optype`.

```
typedef enum {Plus, Assign, Sub} Optype;   /* Sub is new */
/* other declaration as before */
```

Listing 9.37: AST in C: additional `OpType`

Figure 9.12 shows example AST in pictorial form.

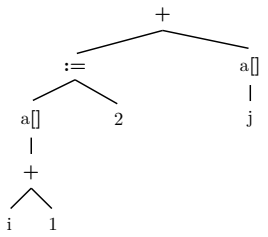


Figure 9.12: Syntax tree for `(a[i+1]:=2)+a[j]`

Code generation for p-code

Listing 9.38 shows how one can generate code for the "array access" grammar from before (in C). Compared to the earlier procedure for code generation from Listing 9.15, the procedure `genCode` has **one additional argument**, a boolean flag. That has to do with the distinction we want to make (here) whether the argument is to be interpreted as address or not. And that in turn is related to the distinction between so called **L-values** and **R-values** and the fact that the grammar allows "assignments" (written `x := exp2`) to be expressions themselves. In the code generation, that is reflected also by the fact we use `stn` (non-destructive writing). Of course, already without arrays, there had been the distinction between L-values and R-values. Nonetheless, the code generation from Listing 9.38 could be achieved without the extra argument `isAddr`.

```
void genCode (SyntaxTree t, int isAddr)  {
  char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL) {
    switch (t->kind) {
    case OpKind:
      { switch (t->op) {
        case Plus:
          if (isAddress) emitCode("Error");  // new check
          else {                             // unchanged
            genCode(t->lchild,FALSE);
            genCode(t->rchild,FALSE);
            emitCode("adi");                 // addition
          }
          break;
        case Assign:
          genCode(t->lchild,TRUE);          //``l-value''
          genCode(t->rchild,FALSE);         //``r-value''
          emitCode("stn");
          break

        case Subs:
          sprintf(codestring,"%s %s", "lda",t->strval);
          emitCode(codestring);
          genCode(t->lchild. FALSE);
          sprintf(codestring,"%s %s %s",
                  "ixa elem_size(", t->strval,")");
          emitCode(codestring);
          if (!isAddr) emitCode("ind 0");  // indirect load
          break;
        default:
          emitCode("Error");
          break;
        }
        break;
      case ConstKind:
        if (isAddr) emitCode("Error");
        else {
          sprintf(codestr,"%s %s","lds",t->strval);
          emitCode(codestr);
        }
        break;
      case IdKind:
        if (isAddr)
          sprintf(codestr,"%s %s", "lda",t->strval);
        else
          sprintf(codestr,"%s %s", "lod",t->strval);
        emitCode(codestr);
        break;
```

```
      default:
        emitCode("Error");
        break;
      }
    }
  }
```

Listing 9.38: `GenCode`, additional argument `isAddr`

9.8.2 Access to records

Let's have also a short look to records. This time we don't show how to extend the abstract syntax further or how to extend the `genCode` implementation in detail. For dealing with records, one may consult also the remarks when discussing record types resp. the memory layout for different data types. Records are not much more complex than arrays, but slots are **not uniformly** sized. Thus one cannot simply access "slot number 10" (using indexed access or pointer arithmetic). Luckily, however, the offsets are all statically known by the compiler, and with that, one can access the corresponding slot.

One complication is: the offset may be statically known, before running the program, but actually **not yet right now**, in the intermediate code phase. It typically may be known only when having decided for the platform. That's still at compile-time, but lies "in the future" in the phased design of the compiler. But it's not hard to solve that. Instead of generating a concrete offset right now, one injects some "function" (say `field_offset`) whose implementation (resp. expansion) will be done later, as part of fixing platform-dependent details. It's similar what we used already in the context of the array-accesses, which made use of a function `elem_size`.

```
typedef struct Rec {
  int i;
  char c;
  int j;
} Rec;
...

Rec x;
```

Listing 9.39: Sample struct type declaration

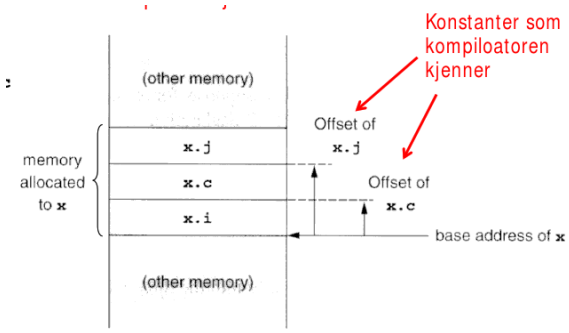


Figure 9.13: Layout for structs or records

Records/structs in 3AIC

As arrays (and objects etc), records are typically are implicitly references. As for arrays, we can just use `&x` and `*x` to do the proper access. Assume a simple read access to `x.j` where `x` is the record. Actually, should better say, `x` is a variable that contains as value a reference to the the array ....

```
t1 = &x + field_offset(x,j)
```

Listing 9.40: simple record access `x.j`

The second example shows read and write record access, i.e., treating a struct as l-value and as r-value, using the address modes of three-address intermediate code.

```
t1  = &x + field_offset(x,j)
t2  = &x + field_offset(x,i)
*t1 = *t2
```

Listing 9.41: Access to structs left and right: `x.j := x.i`

Field selection and pointer indirection in 3AIC Next we cover pointer indirection, actually in connection with records (which is a very common situation). In C-like languages, that's the way one can implement recursive data structure (which makes it an important programming pattern). Of course, in languages without pointers, which may support inductive data types for instance, those structures need to be translated similarly. The C-code in Listing 9.42 shows a typical example, a tree-like data structure.

Listing 9.43 shows how to access those trees, one on the left-hand side, one on the right-hand side of an assignment. The notation -> is C-specific. It's used for the very common situation to access fields of a struct (or union) which is referenced by a pointer as in Listing 9.43. Here the notation is used to "move" up or down the tree.

```
typedef struct treeNode {
  int val;
  struct treeNode * lchild,
                  * rchild;
} treeNode
...

Treenode *p;
```

Listing 9.42: Some tree data structure (using structs)

```
p -> lchild = p;
p           = p->rchild;
```

Listing 9.43: Assignments involving fields

Listing 9.44 and 9.45 show how the two lines in C can be translated to three-address intermediate code, resp. to p-code. Both make use of `field_offset(x,j)` to calculate the off-set. In Listing 9.44, there is no need for the address operator `&` (unlike in earlier examples), since `p` contains is already the address.

```
t1  = p + field_offset(*p,lchild)
*t1 = p
t2  = p + field_offset(*p,rchild)
p   = *t2
```

Listing 9.44: 3AIC

```
lod p
ldc field_offset(*p, lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p, rchild)
sto
```

Listing 9.45: p-code

9.9 Control statements and logical expressions

So far, we have dealt with straight-line code only. In general (intra-procedural) "control" is more complex than straight-line code thanks to **control-statements**. Those include conditionals, switch or case constructs, different forms of loops (while, repeat, for ...), and also breaks, gotos, exceptions ....

The core addition to deal with *control statements* here is the use of **labels**.

> Labels can be seen as "symbolic" representation of "programming lines" or "control points".

Ultimately, in the final binary, the platform will support jumps and conditional jumps which will transfer control (= program pointer) from one address to another, "jumping to an address". Since we are still at an intermediate code level, we do jumps not to real addresses but to labels (referring to the starting point of sequences of intermediate code). As a side remark: also assembly language editors will in general support *labels* to make the program at least a bit more human-readable (and relocatable) for an assembly programmer. Labels and *goto* statements are also known in (not-so-)high-level languages such as classic Basic (and even Java has `goto` as reserved word, even if it makes no use of it).

Besides the treatment of control constructs, we discuss a related issue namely a particular use of boolean expressions. It's discussed here as well, as (in some languages) boolean expressions can behave as control-constructs, as well. Consequently, the translation of that form of booleans, require similar mechanisms (labels) as the translation of standard-control statements. In C-like languages, including Java, that's know as **short-circuiting**.

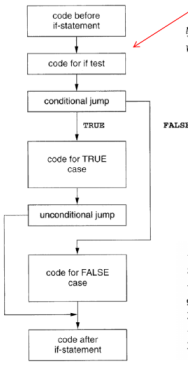
Let's first fix some **abstract syntax,** extending the previous version. The additions are not very fancy, just some syntax for conditionals and for loops. Abstract syntax is in **tree**-form, and the task will be to turn it to a **linear** representation, since we are working with linear intermediate code formats. In principle, the task should be clear, working heavily with conditional jumps to represent conditionals and loops in the abstract syntax; see later Figures 9.14a and 9.14b

$$\begin{aligned} \textit{if-stmt} &\rightarrow \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{while-stmt} &\rightarrow \textbf{while}\ (\ \textit{exp}\ )\ \textit{stmt} \end{aligned} \tag{9.6}$$

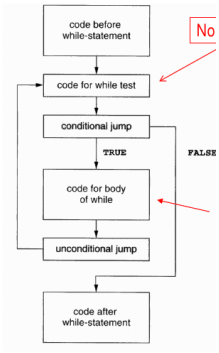
Figure 9.14 shows the structure of the **control-flow graph** of the two structured commands. They should be clear enough.

However, and more importantly for the current discussion: the pictures can also be read as containing more information than the CFG: The graphical arrangement hints at the fact that ultimate, the code is **linear**. Crucial here are *conditional jumps*, but those are **one-armed** commands. That means, one jumps on some condition. But if the condition is not met, one does *not* jump. That is called **fall-through**. In the picture, it's hinted at insofar that the boxes are aligned strictly from top to bottom, it's a linear arrangement of stretches of straight-line code (so called **basic block**) and the arrows are just illustrating the jumps resp. fall-throughs. A graphical illustration of a (control-flow) graph structure would not need to do that, a graph consists of nodes and edges, no matter how one arrange them for illustrative purposes.

Here the two graphs use always the true-case as fall-through. Of course, the underlying intermediate code can support different forms of conditional jumps (like jump-on-zero and jump-on-non-zero) which may swap the situation. Our code will work with **jump-on-false** which explains the true-as-fall-through depiction.



(a) Conditional



(b) While

Figure 9.14: Tracing conditionals and while-loops

Anyway, the pictures are intended to remind us that we are generating code for *linear* intermediate code languages, and in particular, the graph should not be interpreted (with its true and false edge) should not be misunderstood to think we *still* have two-armed jumps.

As said, the "graphical" representation can also be understood as *control flow graph*. The nodes contain sequences of "basic statements" of the form we covered before (like one-line 3AIC assignments) but not conditionals and similar and no procedure calls (we don't cover them in the chapter anyhow). So the nodes (also known as *basic blocks*) contain straight-line code.

In the following we show how to translate conditionals and while statements into intermediate code, both for 3AIC and p-code. The translation is rather straightforward and actually practically identically for both cases, as both forms making use of labels and conditional jumps.

To do the translation, we need to enhance the set of available "op-codes" (= available commands). We need a mechanism for *labeling* and a mechanism for *conditional jumps*. Both kinds of statements need to be added to 3AIC and p-code, and in both variants, they basically work the same, except that the actual syntax of the commands is different. But that's details.

For conditionals **if** (E) **then** S_1 **else** S_2 and while loops **while** (E) S, the 3AIC is given in Listing 9.46, resp. in Listing 9.47.

```
<code to eval E to t1>
if_false t1 goto L1
// goto false branch
<code for S1>
// fall through to true branch
goto L2
// hop over false branch
label L1
<code for S2>
label L2
```

Listing 9.46: Conditional (3AIC)

```
label L1
// label the loop header
<code to evaluate E to t1>
if_false t1 goto L2
// jump to after the loop
<code for S>
goto L1                    // jump back
label L2
// label the loop exit
```

Listing 9.47: While (3AIC)

For comparison, we show also the corresponding p-code in Listing 9.48, resp. in Listing 9.49. We see that both translations work basically the same, which is not surprising, as both linear intermediate code forms have equivalent commands for handling the control flow, namely labeling and jumps to labels, in particular conditional jumps. For p-code, the needed codes are listed in Table 9.5.

```
<code to evaluate E>
fjp L1                  // got false branch
<code for S1>
// fall through to true branch
ujp L2
// hop over false branch
lab L1
<code for S2>
lab L2
```

Listing 9.48: Conditional (p-code)

```
lab L1
// label the loop header
<code to evaluate E>
fjp L2
// jump to after the loop
<code for S>
ujp L1                     // jump back
lab L2
// label the loop exit
```

Listing 9.49: While (p-code)

ujp	unconditional jump ("goto")
fjp	jump on false
lab	label (for pseudo instructions)

Table 9.5: 3 new op-codes for p-code

9.9.1 Boolean expressions

Next we discuss boolean expressions. One may ask, why this is covered in the context of control statements, after all, we have covered how compound expressions are translated, for instance using temporaries in three-address intermediate code.

Booleans are special, as of course they are connected to control-flow. For conditional and loops, the control flow, i.e., where the program execution continues **depends** on the value a the boolean condition. That gives booleans a special place.[9]

Besides that (but connected to that), boolean expression can also evaluated in a way that does not correspond to standard expression evaluating. There are actually two ways how to treat boolean expressions, i.e., two ways how to generate code for boolean expressions. One is to generate code analogous to the principles we covered earlier. The alternative discussed here is known as **short-circuiting**.

Let's look at C, a language which supports short-circuiting evaluation (Java as well).[10] The notation is C-specific, and a popular idiom for nifty C-hackers. For non-C users it may look a bit cryptic.

```
if ((p!=NULL) && p -> val==0)) ...
```

Listing 9.50: Typical idiom in C, relying on short circuiting

A "popular" error in C-like languages are nil-pointer exceptions, and programmers a well-advised to check pointer accesses whether the pointer is nil or not. In the example, the access `p -> val` would derail the program if `p` were nil. However, the "conjunction" checks for nil-ness, and the nifty programmer knows that the first part is checked first. And not only that, if it evaluates to false (or `0` in C), the second conjunct is **not** executed (to find out if it's true or false), it's jumped over. That's known as **circuit evaluation**.

That such circuiting is possible (at source-code level), the semantics must *fix* evaluation order, typically from left to right. Treating boolean expressions like that also make then no longer behave as the usual *logical* constructs. For instance, for logical conjunctions, one would expect $a \wedge b = b \wedge a$. That's not longer the case. But actually it's not so much the fault of short-circuiting or not only, it's in combination with the fact that boolean **expressions** can have **side-effects.** The pointer idiom from Listing 9.50 has no side-effect of the form $x{+}{+}$ or similar. However, the right-hand conjunction in the pointer dereferencing pattern can have a side effect in the form of a null-pointer **exception.** That's precisely why the conjunct on the left-hand side of the conditions checks whether that would happen, and if so, the short-circuiting evaluation jumps over it, avoiding the crash.

Talking about "logic", the short-circuited boolean operators can be "explained" as follows:

$$\begin{aligned} a \textbf{ and } b &\triangleq \textbf{if } a \textbf{ then } b \textbf{ else false} \\ a \textbf{ or } b &\triangleq \textbf{if } a \textbf{ then true else } b \end{aligned} \tag{9.7}$$

That's of course not C (or similar languages), as conditionals are *statements* and cannot be used as expressions. But the pattern captures the idea of short circuiting and shows the

[9] In languages like C, one can also use integers and other non-boolean data in conditionals, so it's not only data officially being declared as boolean that can influence the control-flow.

[10] In Scheme/Lisp, `and` and `or` (among other constructs) are called *special forms* as they behave different from ordinary expressions. It's a different name of the same phenomenon.

mentioned dependency of control-flow on booleans, exploiting that dependence for capture short-circuiting.

To produce intermediate code for short-circuited boolean expression is not very hard. We have discussed how conditionals can be translated using labels and conditional jumps, so the conditional patterns from (9.7) gives a clear way to do it.

We will show it or at least illustrated it on a small example, and we do that for p-code. For that we make use of

2 new op-codes: **equ**, and **neq**

```
lod x
ldc 0
neq     // x!=0 ?
fjp L1  // jump, if x=0
lod y
lod x
equ     //  x =? y
ujp L2  //  hop over
lab L1
ldc FALSE
lab L2
```

Listing 9.51: P-code for `(x!=0) && (x==y)`

The p-code might not be the very best representation, for instance, one may come up with a different solution that does *not* load x two times. A side remark: we are still at intermediate code. Optimizations and the use of registers have not yet entered the picture. That is to say, that the above remark that x is loaded two times might be of not so much concern ultimately, as an optimizer and register allocator should be able to do something about it. On the other hand: why generate inefficient code in the hope the optimizer will clean it up.

Let's also look at how to translated short-circuiting into 3AIC. Conceptually, it's not different from the treatment in p-code.

Example 9.9.1 (Short circuiting in 3AIC). The source code is a compound boolean expression involving "and" and "or".

```
if a < b ||
   (c > d && e >= f)
then
  x = 8
else
  y = 5
endif
```

Listing 9.52: Source code

```
t1 = a < b
if_true t1 goto 1 // short circuit
t2 = c > d
if_false goto 2   // short circuit
t3 = e >= f
if_false t3 goto 2
label 1
x = 8
goto 3
label 2
y = 5
label 3
```

Listing 9.53: 3AIC

□

Let's slightly modify the statement grammar from earlier from equation (9.6), basically adding a break statement (see equation 9.8)). Expressions are dealt with in a rather simplistic manner, supporting only *true* and *false*,

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{if-stmt} \mid \textit{while-stmt} \mid \textbf{break} \mid \textbf{other} \\
\textit{if-stmt} & \rightarrow & \textbf{if (} \textit{exp} \textbf{)} \textit{ stmt } \textbf{else } \textit{stmt} \\
\textit{while-stmt} & \rightarrow & \textbf{while (} \textit{exp} \textbf{)} \textit{ stmt} \\
\textit{exp} & \rightarrow & \textbf{true} \mid \textbf{false}
\end{array} \tag{9.8}
$$

A possible data structure in C for abstract syntax trees for such statements is shown in Listing 9.54.[11]

```
typedef enum {ExpKind, Ifkind, Whilekind,
              BreakKind, OtherKind} NodeKind;

typedef struct streenode {
  NodeKind kind;
  struct streenode * child[3];
  int val; /* used with ExpKind */
           /* used for true vs. false */
} STreeNode;

type STreeNode * SyntaxTree;
```

Listing 9.54: AST in C for statements

Example 9.9.2 (Translating conditionals). Let's use the following conditional statement to illustrate the code generation for such constructs.

```
if (true) while (true) if (false) break else other
```

Listing 9.55: Nested control structures

Note, the code may look ambiguous (remember the dangling-else problem). But that's a parsing issue, and that's behind us. What is meant is shown in the abstract syntax tree in Figure 9.9.2.

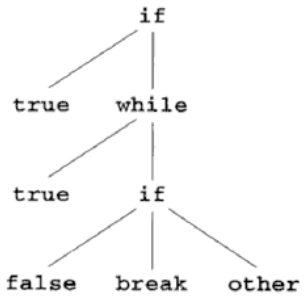


Figure 9.15: Syntax tree

A corresponding possible p-code is shown in Listing 9.56.

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
```

[11] In style it resembles the C data structures we have seen earlier in Listing 9.13 and its extension from Listing 9.37, but those were for expressions. The expressions included side effects, but there were no control-flow complications, including no short-circuiting. Consequently, those AST were translated to straight-line code.

```
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

Listing 9.56: P-code

□

9.9.2 Code generation

Let's look at how to generate intermediate code like the one from Example 9.9.2. We have seen versions of the `genCode`-function already. There had been a "plain" version for expressions in Listing 9.15 and the one from 9.38 (with an additional argument `isAddr`). Now we adapt `genCode` to deal with statements and the control structures, including the `break` statement.

Now we face a similar problem than lead to the addition of additional argument in Listing 9.38. To translate the control structures, the while and the if, involve conditional jumps. To translate a branch of conditional and loops involves generating code containing (conditional) jump statements to a label. Also a break statement occurring in a loop bode or a branch of a condition will be translated to a jump to some label.

The problem is, the jump target is **outside** the branch or body being translated. In the syntax tree, it refers to some code higher up in the tree. One can see it as information not from the statement being translated but from its **context**.[12] As similar phenomenon was in the `genCode` procedure from Listing 9.38: the question whether a variable was interpreted as value or if its *address* was mean is information belonging to the *context* of the variable.

That led back then to give that information as additional argument to the procedure. In the case of Listing 9.38, it base a simple boolean value, stating whether the case for variable should treat a variable as address or not. Here we do the analogous thing.

> We add as **additional argument** to `genCode` the **label** where to jump to, for instance in case of a `break` statement. A variation of the same idea is to hand over two labels, one that represents the "true" and one the "false" case. That may be conceptually clearer, though the generated code will work with one-armed jumps and fall-throughs only, because that's what's supported by the intermediate code.

To do so, we assume that we have a **label generator** (say `genLabel()`. Like for temporaries, it can be used to generate fresh labels when needed. And when is it needed? Of course when generating code for conditionals and while looks, and the code generated for

[12] Context in the same manner that well-typedness of a of an expression containing variable is something that is declared elsewhere, not as part of the expression, i.e., is declared in the expression's context, often before. That's a basic reason why non altogether trivial type checking cannot be context-free.

the while-body resp. the arms of the conditionals are handed over the appropriate jump target(s).

Side remark 9.9.3 (Labels and jumps and control-flow graphs)**.** The label can (also) be seen as designating **nodes** in the **control-flow graph**. I.e., when `genCode` generates labels while traversing the AST, it implicitly generates a CFG.

The labels and the nodes of the control-flow graph may not be 100% in a one-to-one relationship. For instance, looking at the sample (p-)code from Listing 9.56, we see a situation where there are two subsequent lines marked `lab 3; lab 1`. That's not harmful, but uses **two** labels for the same control-flow point or same address. So it can be that the code generator happens to generate two different labels as jump-target to the same node in a control-flow graph. But just because the node carries two different labels does not mean that there are two different nodes, of course. In that sense, nodes and labels generated by procedures like `genCode` are not absolutely alone, but this is just a very minor thing. It's still conceptually true: nodes in the control flow graph correspond to the labels (fine-print applies).

While talking about code generation: it's also possible not to have the code generator implicitly generate some control flow graph in that one can think of the labels as the nodes more or less. Instead a compiler could introduce control-flow graphs **explicitly as another form of intermediate representation**, perhaps a graph-representation of intermediate code, which would afterwards be linearized. The code-generation here, in a way, does the control-flow graph on-the-fly.

As a final remark: we *will* actually cover control-flow graphs later. They will be used in the context for data-flow analysis (which is a very common application for control-flow graphs). However, the way we proceed is: we assume a linear code (with labels and jumps) and before doing the analysis, we first extract the CFG from the linear code. As said, other compilers might officially have a CFG data structure *before* going to a linear code arrangement, so no extraction of the graph from the code would be needed. □

Now to the implementation of the code generation, see Listing 9.57 Listing. The type declaration for the abstract syntax trees was shown earlier in Listing 9.54

```
void genCode(SyntaxTree t, char* label) {
  char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind) {
    case ExpKind:
      if (t->val==0)
           emitCode("ldc false");
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);
      lab1 = genLabel();
      sprintf(codestr,"%s %s", "fjp", lab1);
      emitCode(codestr);
      genCode(t->child[1],label);
      if (t->child[2]!=NULL) {
        lab2 = genLabel();
        sprintf(codestr,"%s %s", "ujp", lab2);
        emitCode(codestr);
      }
      sprintf(codestr,"%s %s", "lab", lab1);
      emitCode(codestr);
      if (t->child[2]!=NULL) {
        genCode(t->child[2],label);
        sprintf(codestr,"%s %s", "lab", lab2);
        emitCode(codestr);
      }
      break;
```

```
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s", "lab", lab1);
      emitCode(codestr);
      genCode(t->child[0],label);
      lab2 = genLabel();
      sprintf(codestr,"%s %s", "fjp", lab2);
      emitCode(codestr);
      genCode(t->child[1],label);
      sprintf(codestr,"%s %s", "ujp", lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s", "lab", lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s", "ujp", label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
    }
}
```

Listing 9.57: Code generation for statements (p-code)

The code being generated is p-code, though actually the important message of that procedure is not that; we know that the treatment of labels and jumps is done analogously for 3AIC. The code also resembles earlier C-code implementation of p-code generation, basically a recursive procedure with a post-fix generation of code for expression evaluation. We have seen that before.

Of course, now we have to make jumps and use labels. As mentioned the (small) challenge we have is: sometimes one has to inject a jump-command to some label which, at that point in the traversal. This is needed (for instance) when doing a break-statement in a loop. The natural way to deal with it is that the recursive procedure takes a label as *additional argument*, that is used to jump-to when processing a break. This argument is handed down the recursive calls. There are alternative ways to deal with this (mini-)challenge. Later we also have a look at an alternative ways, making use of two labels as argument.

Conditionals The conditional has to deal with 3 elements: the boolean expression (which can be a statement and even contain breaks) and the two branches. For the conditional: if a break occurs, we need to jump to the argument label, that's what the label is introduced for, actually. For the two branches (and the condition): All three take the original argument `label` as break-target. We have, however, to take care of the linear arrangement. The control flow graph looks simple enough for a conditional like this, but the code is arranged linearly, and we don't have two-armed conditionals in the target code (only fall-throughs in the else case).

`lab1` is the label for the else-branch. It's the conditional jump target after evaluating the boolean condition and reaching the end without a break. Therefore, it's a branch-on-false. We don't know the target yet, therefore we generate a new label hand it over as argument. The jump is "forward" in a way, which means, we generate the jump code `fjp <lab1>` before we generate the code where we jump to. That's why we need the label as additional argument.

`lab2` is a label needed to hop over the else branch. There's a distinction whether there is an else branch or not (that's done via a check for null-pointer (note that the check is done two times). The label `lab2` is needed only in case there is actually an else case.

It might actually be the case that we don't need `lab2` and use `label` instead.

While The while works similar. But because we have potentially to jump back (by an unconditional jump at the end of the loop body), we need a label for that and that's `lab1`. `lab2` is the break label for the body also generated. It somehow is the "same" point as `label`.

Alternative code generation for boolean expressions and short-circuiting So far, we have sketched code generation in connection with short-circuiting boolean expressions by some examples. In the following we show, also slightly sketchy, how the short-circuiting can be integrated into the `genCode` procedures which we have looked at repeatedly. We do so only for the p-code, but it can be done analogously for the 3AIC. We look at short-circuiting boolean expressions when they are uses in control-flow constructions, i.e., as the boolean condition for conditionals or loop. For that we focus on conditionals, only, i.e., we revisit the the `IfKind` case in the code from Listing 9.57. In that older version, there was *no* short-circuiting. Now, in Listing 9.58, we want to include short-circuiting, and the part is handled by a separate sub-procedure `genBoolCode`; see Listing 9.59

Note that `genBoolCode` takes **two labels** are arguments, one for the true-case one for the false-case. Note also, that there is no general *break* label as third argument. We had introduced that in Listing 9.57 as jump-target "after" the surrounding code in case a break is executed. Basically, we assume that there are no breaks allowed inside boolean expressions. It would be easy to add that to Listing 9.59. as well to treat a possible `break`-case, but the code is a sketch anyway and not all switch-cases are shown. In case the `genBoolCode` does not have a break-label as third argument (as in the shown code), of course, it's not good enough to *assume* that the programmer is not so stupid to use breaks in boolean conditions. If that's forbidden, it should be checked by the semantic analysis phase and if that is violated, an error message should be generated. That's better than letting the compiler stumble upon it during the intermediate code generation phase (for instance not having a `break`-case in `genBoolCode`). If the `genBoolCode` is programmed in C in a similar style as `genCode` from Listing 9.57, there might be a `default` case at the end of the case switch, which at least generates some "error". But, as said, it's better handled in the semantic analysis phase. But having the code generator generating an "error code" now is still better than a situation where the intermediate code generator generates executable code (resp. proper intermediate code that will result afterwards in executable code), where the behavior is unclear (perhaps the code crashes, or does something unexpected, or generates code as if there is no break). And the excuse "the user should not do that and the code generator *assumes* that no one does such a thing" is actually no excuse at all...

On the other hand, there seems indeed no legitimate reason why someone would wish to execute an explicit `break` in an boolean condition. Some would even say, don't use side effects in the boolean condition of a conditional or a loop, though it's quite common practice in C-like languages (also in connection with the short-circuit semantics and the

fact that assignments give back values). Indeed, the short-circuiting treatment of booleans is similar to a break. If, for instance, in an “or” boolean expression, the left sub-expression gives a true, then there is no need to evaluate the right sub-expression, this the execution hops over the corresponding code: it’s like executing a “break” to jump after the rest of the expression and continue there.

```
    case IfKind:
      lab_t = genLabel();
      lab_f = genLabel();
      genBoolCode(t->child[0],lab_t,lab_f);    // boolean condition
      sprintf(codestr,"%s %s", "lab", lab_t); // if-branch
      emitCode(codestr);
      genCode(t->child[1],label);
      lab_x = genLabel();
      if (t->child[2]!=NULL) {                 // does there exists an else branch?
        sprintf(codestr,"%s %s", "ujp", lab_x);
        emitCode(codestr);
      }
      sprintf(codestr,"%s %s", "lab", lab_f); // else-branch
      emitCode(codestr);
      if (t->child[2]!=NULL) {                 // does there exists an else branch?
        genCode(t->child[2],label);
        sprintf(codestr,"%s %s", "lab", lab_x);// post-statement label (if 2 arms)
        emitCode(codestr);
      }
      break;
    case WhileKind:
```

Listing 9.58: Alternative code generation (conditionals)

Anyway, Listing 9.59 shows a few cases, the one for “and” and “or”, and also one comparison operator. The situation for "or" is also shown in Figure 9.16 (where l_t stands for `lab_t` in the code etc.).

```
void genBoolCode (string lab_t, lab_f) =
  ...
  switch ... {
    case "||" : {
      String lab_x = genLabel();
      left.genBoolCode(lab_t, lab_x);
      sprintf(codestr,"%s %s", "lab", lab_x);
      emitCode(codestr);
      right.genBoolCode(lab_t, lab_f);
    }

    case "&&" : {
      String lab_x = genLabel();
      left.genBoolCode(lab_x, lab_f);
      sprintf(codestr,"%s %s", "lab", lab_x);
      emitCode(codestr);
      right.genBoolCode(lab_t, lab_f);
    }

    case "not" : {  // here just a left tree
      left.genBoolCode(lab_f, lab_t);
    }

    case "<" : {             // example for a binary relation
      String t_1, t_2, t_3; //
      t_1 = left.genIntCode();
      t_2 = right.genIntCode();
      t_3 = genLabel();
      emit4(t_3,t_1, "lt", t_2);
      emit3("fjp", t_3, lab_f);
      emit2("ujp", lab_t);
    }
```

Listing 9.59: Alternative `genBoolCode`, short circuiting

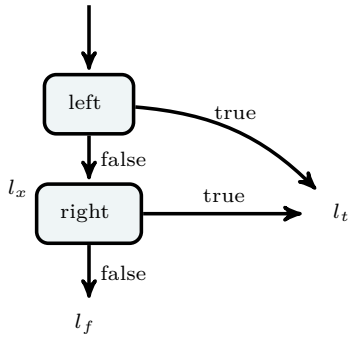


Figure 9.16: Short circuiting booleans, case “or”

Bibliography

[1] Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

[2] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index