Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter
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

The chapter is called *intermediate code generation*. At the current stage in the lecture (and the current "stage" in a compiler) we have to process as input a 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, 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 as discussed in the introductory part of the chapter covering run-time environments. Basically, to translate procedure *bodies* into sequences of instructions.

Ultimately, the generated instructions are binaries, resp. in machine code, which is *platform depedent*. Generating platform dependent code is this part of the back-end. However, the task of generating code is usually split into generating first *intermediate code* and afterwards, "real code". This chapter here is about this *intermediate code* generation.

Making use of intermediate code not just done in this lecture. Using intermediate code as another *intermediate representation* internal to the compiler is commonplace. The intermediate code 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. But it's not the exact instruction set also in that the IR will still rely on some abstractions which are not 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.

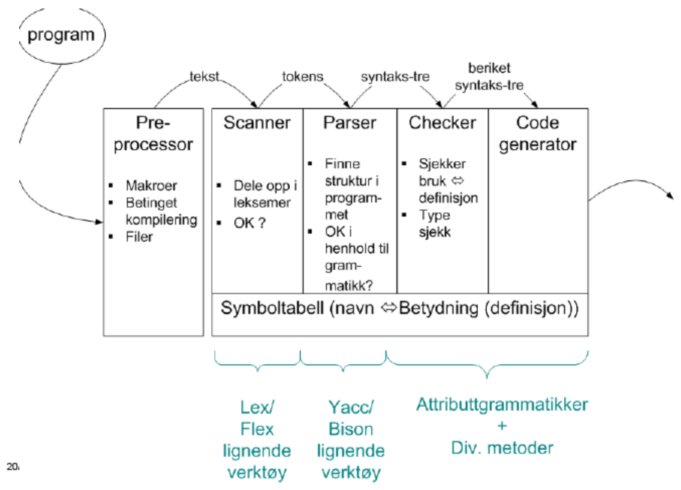
If one has some "machine-code" resembling intermediate representation, the task of porting a compiler to a new platform is easier. Furthermore, one can start doing certain code analyses and optimization already on the IC, thereby making optimizations available for all platform-dependent backends, without 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 subsequente 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.

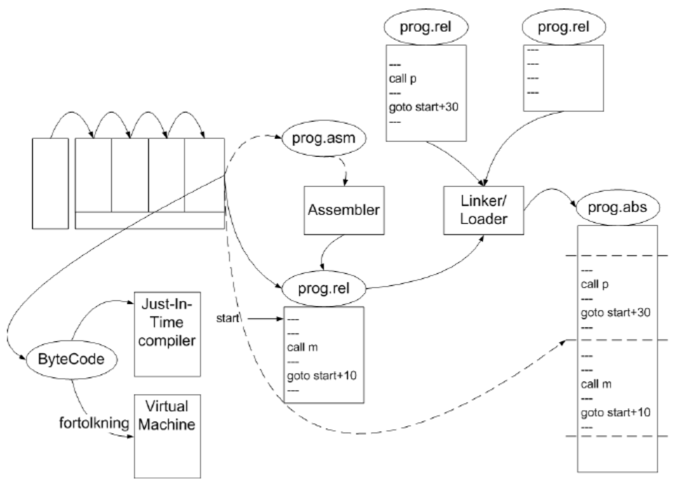
Schematic anatomy of a compiler


program
tekst
tokens
syntaks-tre
beriket syntaks-tre
Pre-processor
Makroer
Betinget kompilering
Filer
Scanner
Dele opp i leksemer
OK ?
Parser
Finne struktur i program-met
OK i henhold til gram-matikk?
Checker
Sjekker bruk ⇔ definisjon
Type sjekk
Code generator
Symboltabell (navn ⇔Betydning (definisjon))
Lex/ Flex lignende verktøy
Yacc/ Bison lignende verktøy
Attributtgrammatikker + Div. metoder
20

- code generator:
  - may in itself be "phased"
  - using additional intermediate representation(s) (IR) and *intermediate code*

We have seen the figure and similars in the introductory chapter, same with the following. It's a reminder that a compiler may involve different forms of machine code. Also interpreted byte code may play a role.

A closer look



Various forms of "executable" code

- different forms of code: relocatable vs. "absolute" code, relocatable code from libraries, assembler, etc.
- often: specific file extensions
  - Unix/Linux etc.
    - asm: `*.s`
    - rel: `*.o`
    - rel. from library: `*.a`
    - abs: files without file extension (but set as executable)
  - Windows:
    - abs: `*.exe`[1]
- *byte code* (specifically in Java)
  - a form of intermediate code, as well
  - executable on the JVM
  - in .NET/C$\sharp$: *CIL*
    - also called byte-code, but compiled further

There are many different forms of code. 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 Java-centric terminology, while the underlying concept is not. 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 representation". 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 can be seen from the above remarks, the terminology is a bit unclear. P-code normally 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

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

level is an unpractical execution mechanism, mostly. Interpreting a language on a virtual machine is already quite closer to machine exectition, 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 are complete virtualization, where a whole operating system is simulated (often 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 a better criterion than the "size" of the op-code ("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 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 nowaday's 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), at least postscript is said to be inspired by Forth.

Generating code: compilation to machine code

- 3 main forms or variations:
  1. machine code in textual **assembly format** (assembler can "compile" it to 2. and 3.)
  2. **relocatable** format (further processed by *loader*)
  3. **binary** machine code (directly executable)
- seen as different representations, but otherwise equivalent
- in practice: for *portability*
  - as another intermediate code: "platform independent" *abstract machine code* possible.
  - capture features shared roughly by many platforms
    * e.g. there are *stack frames*, static links, and push and pop, but *exact* layout of the frames is platform dependent
  - platform dependent details:
    * platform dependent code
    * filling in call-sequence / linking conventions

done in a last step

Byte code generation

- semi-compiled well-defined format
- platform-independent
- further away from any HW, quite more high-level
- for example: Java byte code (or CIL for .NET and C$^\sharp$)
  - can be interpreted, but often compiled further to machine code ("just-in-time compiler" JIT)
- executed (interpreted) on a "virtual machine" (like JVM)
- often: *stack-oriented* execution code (in post-fix format)
- also *internal* intermediate code (in compiled languages) may have stack-oriented format ("P-code")

CIL stands for *common intermediate language* (earlier known as Microsoft Intermediate Language, MSIL). There is actually also another intermediate language called CIL, that's the C intermediate language. Microsoft's intermediate language is certainly more widely used. The C intermediate language, developed at UC Berkely, is used for instance for developing verifying C compilers in Coq, which is an ambitious project (see CompCert).

9.2 Intermediate code

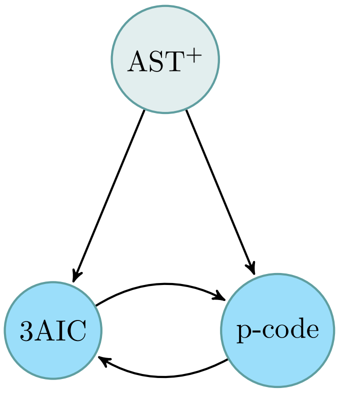
This short section basically gives a short preview of the two forms of intermediate code we will cover in the lecture. Three-address intermediate code is covered in Section 9.3 and p-code in Section 9.4.

Use of intermediate code

- two kinds of IC covered
  1. **three-address code** (3AC, 3AIC)
     - generic (platform-independent) abstract machine code
     - new names for all intermediate results
     - can be seen as unbounded pool of maschine registers
     - advantages (portability, optimization ...)
  2. **P-code** ("Pascal-code", cf. Java "byte code")
     - originally proposed for interpretation
     - now often translated before execution (cf. JIT-compilation)
     - intermediate results in a *stack* (with postfix operations)
- *many* variations and elaborations for both kinds
  - addresses represented *symbolically* or as *numbers* (or both)
  - granularity/"instruction set"/level of abstraction: high-level op's available e.g., for array-access or: translation in more elementary op's needed.
  - operands (still) typed or not
  - ...

Various translations in the lecture

- AST here: tree structure *after* semantic analysis, let's call it AST$^+$ or just simply AST.
- translation AST $\Rightarrow$ P-code: appox. as in oblig 2
- we touch upon general problems/techniques in "translations"
- one (important) aspect ignored for now: *register allocation*



As mentioned earlier, the translation from typed ASTs to p-code corresponds to the task of the second oblig. The target code will be some stack-oriented byte-code format. The corresping 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 an 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 addresses for the arguments, and one target address for the result. In particular, binary operations that do calculations use 3, like addition or bitwise and. See equation (9.1).

We mentioned before that our intermediate code does not make use of addresses and registers (which is a common thing to do for intermediate code). That means, the instructions don't literally work with 3 addresses, but rather they involve 3 variables or constants. The code also not only makes use of "ordinary" variables (like the ones that originate from the source code), but the 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 assume, 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 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, very much 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.

However, one can do that more elegantly, specifying jumps and 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, it's rather that there is an extract *label instruction* which is part of the instruction set. Of course, it's equivalant. Adding a label instruction like `label L`, which means that one can use for intance `jmp L` to just effectivle to 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. They obviously also not make use of 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).

As far as oblig 2 is concerned. The instruction set in byte code of course supports jumps and conditional jumps. However, the instruction set does not offer *labels*. Instead one will have to deal with jumping the more low-level way jumping directly "addresses", where an index in an array cornensponds to the concept of address. That's less convenient that doing it symbolically, but not much so. When programming the code generator, one can (and will) of course remember and address or the index on the byte array in some properly named variable, and that serves the same purpose. This way, the address is, so to say, symbolically remembered in the meta-language, presumably Java, and is not part of the programming language itself, i.e., mentioned in the byte-code instructions.

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". and each time the code generator encounters the need to store an intermediate 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

- common (form of) IR

TA: Basic format

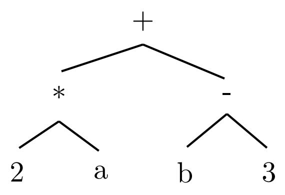
$$x = y \ \mathbf{op} \ z \tag{9.1}$$

- x, y, z: names, constants, temporaries ...
- some operations need fewer arguments

- example of a (common) **linear IR**
- *linear* IR: ops include *control-flow* instructions (like jumps)
- alternative linear IRs (on a similar level of abstraction): 1-address (or even 0) code (stack-machine code), 2 address code
- well-suited for optimizations
- modern architectures often have 3-address code like instruction sets (RISC-architectures)

3AC example (expression)

`2*a+(b-3)`



Three-address code

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

alternative sequence

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

We encountered the notion of *temporaries* already in connection with the activation records. There, the activation records for some function 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. The slide shows two versions that do the same thing. The two code listings are not radically different. The fact that both do the same captures the fact that the order of evaluation does not matter.

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.

3AIC instruction set

- basic format: $x = y\,\mathbf{op}\,z$

- but also:
  - $x = \mathbf{op}\, z$
  - $x = y$
- *operators*: +,-,*,/, $<$, $>$, `and`, `or`
- `read` x, `write` x
- `label` L (sometimes called a "pseudo-instruction")
- conditional jumps: `if_false` x `goto` L
- t_1, t_2, t_3 .... (or `t1, t2, t3, ...`): **temporaries** (or temporary variables)
  - assumed: *unbounded* reservoir of those
  - note: "non-destructive" assignments (single-assignment)

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 a representation of abstraction at the current of memory locations (ultimately addresses in main memory if registers cannot be used), labels are an representation of addresses, ultimately translated to relocatable addresses and ultimately to addresses in the code segment.

Illustration: translation to 3AIC

Source

```
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
```

Target: 3AIC

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

Variations in the design of 3A-code

- provide operators for `int`, `long`, `float` ....?
- how to represent program *variables*
  - names/symbols
  - pointers to the declaration in the symbol table?

  - (abstract) machine address?
- how to store/represent 3A *instructions*?
  - **quadruples**: 3 "addresses" + the op
  - *triple* possible (if target-address (left-hand side) is always a *new temporary*)

Quadruple-representation for 3AIC (in C)

```
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
```

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 illustrate 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 can be 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 heavily on stack-manipulations. Very roughly, where 3AIC uses temporaries to store intermediate results, p-code stores those on the stack. We will see details for both later, when we look how to compile to either intermediate code format.

So we cover 3AIC and "1AIC" (p-code), there is also 2AC / 2AIC, which we will not cover, at least not in this chapter. 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

- different common intermediate code / IR
- aka “one-address code”[2] or stack-machine code
- used prominently for Pascal
- remember: post-fix printing of syntax trees (for expressions) and “reverse polish notation”

P-code is an abbreviation for portable code. Some people 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...

Example: expression evaluation `2*a+(b-3)`

```
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
```

The code should be clear enough (with the help of the commentaries on the right-hand column). This first example is concerned with *expression evaluation*, in particular expressions without side effects. Expressions are dealt with in the mentioned “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 not part of this example. The next slide translates an *assignment* to 3AIC. In that example, we see both versions of the load-command.

[2] There’s also two-address codes, but those have fallen more or less in disuse for intermediate code.

P-code for assignments: `x := y + 1`

- assignments:
  - variables left and right: *L-values* and *R-values*
  - cf. also the values ↔ references/addresses/pointers

```
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
```

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 another (related) difference, which may be less obvious. For `x`, the *address* is loaded as part of a step, for `y` it's the content. We need the address of `x` to store back the result at the end of the generated code.

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.

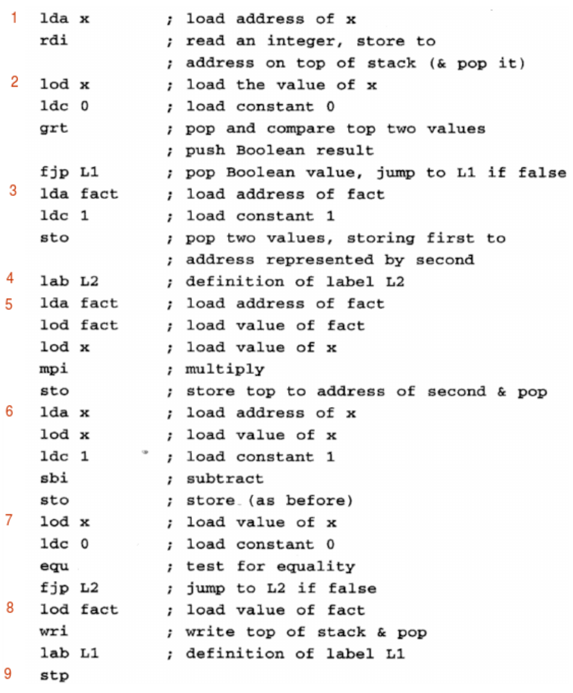
P-code of the faculty function

Source

```
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
```

P-code



```
1   lda x       ; load address of x
    rdi         ; read an integer, store to
                ; address on top of stack (& pop it)
2   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, jump to L1 if false
3   lda fact    ; load address of fact
    ldc 1       ; load constant 1
    sto         ; pop two values, storing first to
                ; address represented by second
4   lab L2      ; definition of label L2
5   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 & pop
6   lda x       ; load address of x
    lod x       ; load value of x
    ldc 1       ; load constant 1
    sbi         ; subtract
    sto         ; store (as before)
7   lod x       ; load value of x
    ldc 0       ; load constant 0
    equ         ; test for equality
    fjp L2      ; jump to L2 if false
8   lod fact    ; load value of fact
    wri         ; write top of stack & pop
    lab L1      ; definition of label L1
9   stp
```

9.5 Generating P-code

After having introduced the concept of p-code in Section 9.4, including (relevant parts of) the instruction set, we 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.

Assignment grammar

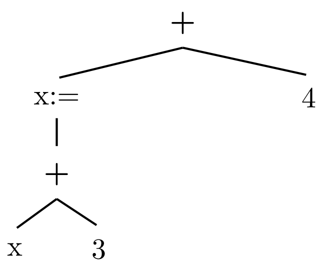
One way to describe the code generation is with an *attribute grammar*. So let's therefore fix a context-free grammar first, fixing the syntax, for which we later show appropriate semantic rules in the attribute grammar formalism.

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. In the previous example, the expression was side-effect free, but for the current example, that's not the case. That expressions like `(x:=x+3)+4` makes 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 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 have to do it otherwise. Consequently, the p-code in the example from the older section is *not* what would be generated here.

Grammar

$$\begin{array}{rcl}
\mathit{exp}_1 & \rightarrow & \textbf{id} := \mathit{exp}_2 \\
\mathit{exp} & \rightarrow & \mathit{aexp} \\
\mathit{aexp} & \rightarrow & \mathit{aexp}_2 + \mathit{factor} \\
\mathit{aexp} & \rightarrow & \mathit{factor} \\
\mathit{factor} & \rightarrow & (\,\mathit{exp}\,) \\
\mathit{factor} & \rightarrow & \textbf{num} \\
\mathit{factor} & \rightarrow & \textbf{id}
\end{array}$$

`(x:=x+3)+4`



As mentioned, the grammar covers only expression and assignments, i.e., straight-line code, but no control-structures.

As a side remark: we said that the intermediate code generation takes typically abstract syntax. Typical abstract syntax would not contain parentheses and the distinction between factors and terms etc. is more typical for grammars covering concrete syntax and parsing. But the question, whether the grammar describes typcially abstract or concrete syntax, is not too relevant for the principle of the translation here, and after all, one can use concrete syntax as abstract syntax trees, even if it often better design to make the AST a bit more abstract. Anyway, we don't bother to show the parentheses in the tree.

Generating p-code with A-grammars

- goal: p-code as *attribute* of the grammar symbols/nodes of the syntax trees
- *syntax-directed translation*
- technical task: turn the syntax tree into a *linear* IR (here P-code)

$\Rightarrow$
  - "linearization" of the syntactic tree structure
  - while translating the nodes of the tree (the syntactical sub-expressions) one-by-one

- not recommended at any rate (for modern/reasonably complex language): code generation *while* parsing[3]

The use of A-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's 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 principle core of 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 compositional 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

[3]One can use the a-grammar formalism also to describe the treatment of ASTs, not concrete syntax trees/parse trees.

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.

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.

A-grammar for statements/expressions

- focus here on expressions/assignments: leaving out certain complications
- in particular: control-flow complications
  - two-armed conditionals
  - loops, etc.
- also: code-generation "intra-procedural" only, rest is filled in as *call-sequences*
- A-grammar for intermediate code-gen:
  - rather simple and straightforwad
  - only 1 *synthesized* attribute: `pcode`

As mentioned, the code generated here is for straight-line code only and relatively simply, as can be seen on the a-grammar on the next slide.

A-grammar

- "string" concatenation: ++ (construct separate instructions) and ˆ (concat one instruction)

productions/grammar rules	semantic rules
$exp_1 \rightarrow \mathbf{id} := exp_2$	exp_1 .pcode = "**lda**"ˆ**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
$factor \rightarrow \mathbf{num}$	$factor$ .pcode = "**ldc**"ˆ**num**.strval
$factor \rightarrow \mathbf{id}$	$factor$ .pcode = "**lod**"ˆ**num**.strval

The op-codes are marked in red. 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`).

Linearization

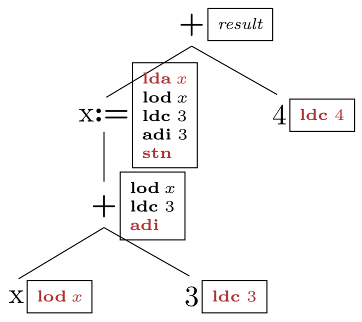
Let's address another small point here. As mentioned, we are dealing with a *linear* IR: like 3AIC and other formats, p-code is a linear IR. It is a language consisting of a linear sequence of simple commands (and uses jumps and labels for control, even though those parts are currently not in the focus). 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 (justing 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 ˆ 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 if of of quite simple structure). Therefore, in a last steps, one needs to flatten out the tree to a ultimate linear list. See Listings 9.3 and 9.4.

Why does one do so? 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 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 shown later reflects pretty truthfully the AG here.

`(x := x + 3) + 4`

Attributed tree



"result" attr.

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

- note: here `x:=x+3` has a side-effect *and "return" value* (as in C . . .):
- **stn** ("store non-destructively")
  - similar to **sto** , but *non-destructive*
    1. take top element, store it at address represented by 2nd top
    2. discard address, but not the top-value

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.

Implementation in a functional language

The following slides 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 transscripts of the grammars we have seen.

Overview: p-code data structures

Source

```
type symbol = string

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

Listing 9.1: Syntax of the source language (expressions with side effects)

Target

```
type instr   =  (* p-code instructions *)
    LDC of int
  | LOD of symbol
  | LDA of symbol
  | ADI
  | STN
  | STO


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

type program = instr list
```

Listing 9.2: Syntax of the target language

- symbols:
  - here: strings for *simplicity*
  - concretely, symbol table may be involved, or variable names already resolved in addresses etc.

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.

Two-stage translation

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

val linearize: Pcode.tree ->  Pcode.program

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

Listing 9.3: Code generation (interface)

```
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 9.4: Code generation

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. Appending lists in functional languages is typically not tail-recursive and one may be better off, efficiency-wise, to split it into two stages as shown.

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 (Listing 9.5).

Source language AST data in C

```
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 9.5: AST in C (for expressions with assignments)

Figure 9.1 shows schematically a small sample AST. The table summarizes the "attributes" per node.

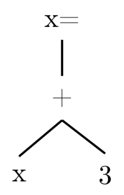


Figure 9.1: Sample AST

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

Code-generation via tree traversal (schematic)

```
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;
```

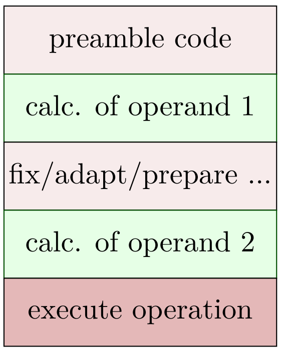
Listing 9.6: Schematic code generation `GenCode` in C

This sketch of a code skeleton basically says: the code generation is a recursive procedure, traversing a given abstract syntax tree. During traversal, it involves prefix-actions, postfix actions and maybe even infix-actions. 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 9.7 later for more complete code for `codeGen` for expressions.

Code generation from AST$^+$

- main "challenge": linearization
- here: relatively simple
- no control-flow constructs
- linearization here (see a-grammar):
  - string of p-code

  – not necessarily the ultimate choice (p-code might still need translation to “real” executable code)



Code generation

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 9.4, 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`). 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, for various reasons, the 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 9.3). 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 9.7: `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 faculty 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 9.8.

3AIC manual translation again

Source

```
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
```

Target: 3AIC

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

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 faculty, we see also here labelling commands (pseudo-instructions) and (conditional) jumps, as in the target code when translated to p-code.

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 abstract grammar was shown on page 15). In the following, we start by repeat the data structure for the source language (which is unchanged) and showing the data structures for the target language similar what we did for the p-code. The data structure can be seen as “abstract syntax” for the 3AIC. 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 format for the source code is unchanged, see Listing 9.1.

Three-address code data structures (some)

Data structures (target)

```
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
```

Listing 9.8: Syntax of the target language (3AIC)

- symbols: again strings for simplicity
- again “trees” not really needed (for simple language without more challenging control flow)

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.

Translation to three-address code

```
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 9.9: Code generation 3AIC (expressions)

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.

Three-address code by synthesized attributes

- similar to the representation for p-code
- again: purely synthesized
- semantics of executing expressions/assignments[4]
  - side-effect plus also
  - value
- *two* attributes (before: only 1)
  - `tacode`: instructions (as before, as string), potentially empty
  - `name`: “name” of variable or tempary, where result resides[5]
- evaluation of expressions: *left-to-right* (as before)

A-grammar

productions/grammar rules	semantic rules
$exp_1 \rightarrow \mathbf{id} = exp_2$	exp_1 `.name` $=$ exp_2 `.name`
	exp_1 `.tacode` $=$ exp_2 `.tacode` $++$ $\mathbf{id}$`.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`
$factor \rightarrow \mathbf{num}$	$factor$ `.name` $=$ $\mathbf{num}$`.strval`
	$factor$ `.tacode` $=$ ""
$factor \rightarrow \mathbf{id}$	$factor$ `.name` $=$ $\mathbf{num}$`.strval`
	$factor$ `.tacode` $=$ ""

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.

Another sketch of 3AI-code generation

The code-sketch shows the code generation in a C-like notation, following the discussed principles.

[4]That's one possibility of a semantics of assignments (C, Java).

[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.

```
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 9.10: Code generation 3AIC (expressions)

- "return" of the two attributes
  - name of the variable (a *temporary*): officially returned
  - the code: via *emit*
- note: *postfix* emission only (in the shown cases)

Generating code as AST methods

- possible: add `genCode` as *method* to the nodes of the AST
- e.g.: define an abstract method `String genCodeTA()` in the `Exp` class (or `Node`, in general all AST nodes where needed)

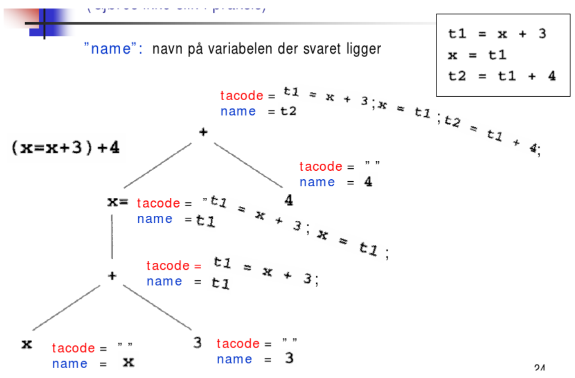
```
    String genCodeTA() { String s1,s2; String t = NewTemp();
      s1 = left.GenCodeTA();
      s2 = right.GenCodeTA();
      emit (t + "="  + s1 + op + s2);
      return t
    }
```

ASTs are trees, of course, and we have seen how one can realize the AST data structure in object-oriented, class-based languages, like Java etc., and probably most have chosen a corresponding representation in oblig 1. Of course, recursion over such data structure can be done straightforwardly, by adding a corresponding method. That's object-orientation "101": one adds a corresponding method to the classes, whose instances represent different nodes in the trees, and then calls them recursively, as shown in the code sketch.

Whether it is a good design from the perspective of modular compiler architecture and code maintenance, to clutter the AST with methods for code generation and god knows what else, e.g. type checking, pretty printing, optimization . . . , is a different question.

A better design, many would posit, is in this situation to separate the functionality from the tree structure, i.e., to separate the "algorithm" from the "data structure", not embedd the algorithm. Such a separation can be achieved in Java-like OO languages but a design-pattern called *visitor*. It allows to iterate over recurive stuctures "from the outside". It's a better design in our context of compilers; it allows to separate different modules from the central data structure and intermediate representation of ASTs (and might be useful for other intermediate representations as well). Since this is not a lecture about Java or C++ design patterns, but about (principles of) compilers, so we leave it like at that, especially since the "embedded solution" shown on the slide works ok as well. Some groups for oblig 1 actually did the effort to realize the print-function as visitor (at least 2020, and previous years, but not this year).

Attributed tree `(x:=x+3) + 4`



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

- note: room for optimization

To conclude this section, here the generated code for the example we have seen before, presented as attributes from the AG.

9.7 From P-code to 3A-Code and back: static simulation & macro expansion

In this intermezzo we shortly have a look 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 attemps 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.

"Static simulation"

- *illustrated* by transforming p-code ⇒ 3AC
- restricted setting: straight-line code
- cf. also *basic blocks* (or elementary blocks)
  - code without branching or other control-flow complications (jumps/conditional jumps…)
  - often considered as basic building block for static/semantic analyses,
  - e.g. basic blocks as nodes in *control-flow graphs*, the "non-semicolon" control flow constructs result in the edges
- terminology: static simulation seems not widely established
- cf. *abstract interpretation*, *symbolic execution*, etc.

The term "static simulation" seems like an oxymoron, a contradicton in itself. Simulation sounds like running a program, and static means, at compile time, before running a program. And, due to fundamental limitations (undecidablity of the halting problem), the compiler in general cannot simulate a program (for reasons of analysis or, here specifically, for translating it to a different representation). However, here we are in the quite restricted situation: straight-line code (especially no loops), which means the program terminates anyway, actually, the number of steps it does is known, it's the number of lines. So it's a finite problem, there are no issues with undecidability. Being finite, one can execute "mentally" one command after the other and know what will happen when running the program. That's what the compiler does for the translation and one can call it *static simulation*. Actually and as mentioned, the term "static simular" is not very widely used in compiler construction, that's why I put it into quotation marks.

The other mentioned techniques, like abstraction interpretation and symbolic execution, are well-established techniques and frameworks. Like static analysis here, they work by "mentally" executing the code step by step to achieve their result. They are used though for semantic *analysis*, not for compiling or translating one code representation into another, at least not directly. In a very general way, of course, all semantic analyses to some extent, statically "simulate" the code (the code on AST level, or intermediate code, or whatever). After all, the semantic analysis phase, generally, analyses the given code to *predict* what might happen at run-time, at least approximately, and the predictions helps to generate the code or generate better code, or optimize the given code by transforming it. This prediction is based to mentally "execute" the given code, one could say *simulate* the code execution. This "simulation" aspect is more felt or less in different techniques. Even data-flow analysis can be understood loosely as simulating, on an abstract level, the given program. Loosely insofar, that the data flow analysis typically does not need to follow the order of the statements as they appear in the program, but can treat them in a different orders. Therefore, the aspect of "simulation" or "execution" is typically less felt for data-flow analysis. We will look at data flow analysis in the following chapter for so-called *live variable analysis*. That's an important kind of data flow analysis, and also typical in the sense, that many other kind of data flow analyses work similarly. Another reason why the execution aspect feels not so pronouncedv in data flow analysis and likewise in abstract interpretation is that they work on "abstractions"; they abstract away from details of the concrete program and its behavior. Those techniques simulate or executes the behavior therefore on an (more or less) abstract level; the term *“abstract interpretation”* directly expresses that. Being abstract in the sense of ignoring details. As a consquence, the analytic predictions those techniques yield via “simulation” or via “abstractly executing the program” are not precise, but *approximative*.

The latter point is a *crucial* difference to what we do here! Translating from one (intermediate) code representation to a another one cannot ignore details or abstract away from anything. The transformation has to preserve the semantics, obviously.

We show two directions of such a translation: form p-code two 3AIC, and vice version. The reverse direction, from 3AIC to p-code can be done quite trivially, by macro expansion. That’s a technique not based on 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.

That will be easy. However, as it turns out, one could better. When comparing the result from translating *directly* from an abstract syntax tree to p-code via the indirect result, first to 3AIC and then macro-expanding that to p-code, it’s clear that the direct route results in better code.

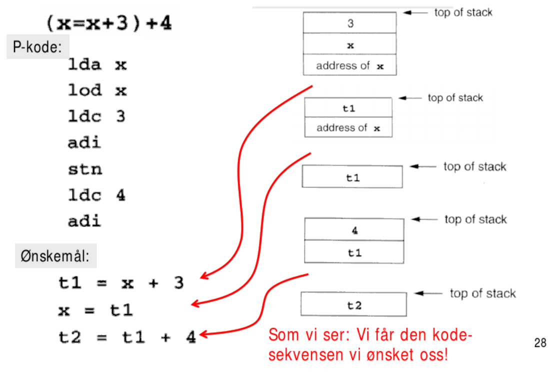
One can actually rememdy that. One just has to be more smart about how to translate 3AIC to p-code. The macro expansion translation is correct, but not very clever. If the translation is not purely syntactical, but the the semantics of the translated constructs into account, one can do much better. And then we are back at doing something that one may call static simulation.

We won’t show how to translated 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).

P-code $\Rightarrow$ 3AIC via “static simulation”

- difference:
  - p-code operates on the *stack*
  - leaves the needed “temporary memory” implicit
- given the (straight-line) p-code:
  - traverse the code = list of instructions from beginning to end
  - seen as “simulation”
    - conceptually at least, but also
    - concretely: the translation can make *use* of an actual stack

From P-code ⇒ 3AIC: illustration



```
(x=x+3)+4
P-kode:
    lda x
    lod x
    ldc 3
    adi
    stn
    ldc 4
    adi
Ønskemål:
    t1 = x + 3
    x = t1
    t2 = t1 + 4
```

The slide illustrates the concept 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 target 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 stages 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 of 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`.

P-code ⇐ 3AIC: macro expansion

- also here: simplification, illustrating the general technique, only
- main simplification:
  - register allocation
  - but: better done in just another optmization "phase"

The inverse direction of the translation is simpler, at least when doing it in a simple way. It does not need any static simulation of the architecture, i.e., considering the program's semantic, it can work simply on the *syntactic* structure of the input program. It simple expands each line by a corresponding sequence of p-code instructions. The is illustrated on the basic 3AIC instruction on the next slide and afterwards on the previous example.

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

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

Example: P-code ⇐ 3AIC (`(x:=x+3)+4`)

There are two different p-codes shown, translated in different ways. One indirectly, via the 3AIC, which is macro-expanded as illustrated. The second p-code is generated *directly* from the abstract syntax code. Clearly, the directly translated code is quite much shorter (and more efficient). 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 common place 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.

In a preview of code generation in the last chapter: similar information, which value is stored where, in particular in which register and which main-memory address, that style of information tracking will be employed in that context later as well.

source 3AI-code

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

Direct p-code

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

P-code via 3A-code by macro exp.

```
;--- 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
```

cf. indirect 13 instructions vs. direct: 7 instructions

As mentioned earlier, translating via macro expansion is correct, though it can be improved ("optimized"). We sketch a bit how that can be achieved.

Indirect code gen: source code ⇒ 3AIC ⇒ p-code

- as seen: *detour* via 3AIC leads to sub-optimal results (code size, also efficiency)
- basic deficiency: too many *temporaries*, memory traffic etc.
- several possibilities
  - avoid it altogether, of course (but remember JIT in Java)
  - chance for *code optimization* phase
  - here: more clever "macro expansion" (but sketch only)

  the more clever macro expansion: some form of *static simulation* again
- don't macro-expand the linear 3AIC
  - brainlessly into another *linear* structure (p-code), but
  - "statically simulate" it into a more *fancy* structure (a *tree*)

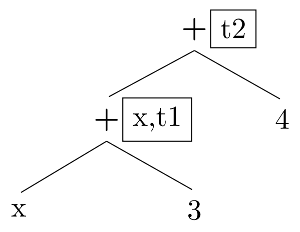
"Static simulation" into tree form (sketch)

- more fancy form of "static simulation" of 3AIC
- *result*: **tree** labelled with
  - operator, together with
  - variables/temporaries containing the results

Source

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

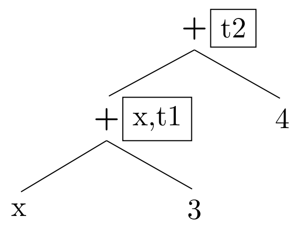
Tree



note: instruction `x = t1` from 3AIC: does *not* lead to more nodes in the tree

P-code generation from the generated tree

Tree from 3AIC



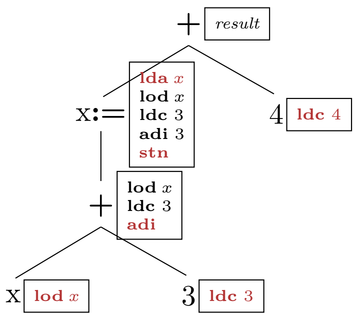
Direct code = indirect code

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

- with the thusly (re-)constructed tree

⇒ p-code generation
  - as before done for the AST
  - remember: code as synthesized attributes
- the "trick": reconstruct essential syntactic tree structure (via "static simulation") from the 3AI-code
- Cf. the macro expanded code: additional "memory traffic" (e.g. temp. t_1)

Compare: AST (with direct p-code attributes)



9.8 More complex data types

Next we drop one of the simplifications we have done so far, concerning the involved *data*. We have a lock 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. 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.

Status update: code generation

- so far: a number of simplifications
- data types:
  - integer constants only
  - no complex types (arrays, records, references, etc.)
- control flow
  - only expressions and
  - sequential composition

  ⇒ **straight-line code**

Address modes and address calculations

- so far
  - just standard "variables" (l-variables and r-variables) and temporaries, as in `x = x + 1`
  - variables referred to by their *names* (symbols)
- but in the end: variables are represented by *addresses*
- more complex *address calculations* needed

addressing modes in 3AIC:

- `&x`: *address* of `x` (not for temporaries!)
- `*t`: *indirectly* via `t`

addressing modes in P-code

- `ind i`: *indirect load*
- `ixa a`: *indexed address*

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 *two* arguments: the address of some place (in memory) and an *offset*. That should remind us to the way that arrays are layed out in memory (we had discussed that earlier). Indeed, HW-supported indexed access is one important reason, that arrays are a very efficient data structure. We will illustrate the new constructions on arrays (but also records) in the following.

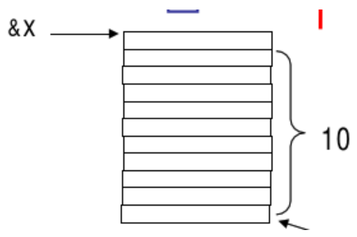
In 3AIC, we don't have indexed addressing, we have a C-like situation, with access to the addresses of variables. The `&x` operation corresponds to the `lda` instruction in p-code.

Loading *indirectly* (in 3AIC and 1AIC) 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 when using `*t` on the left-hand side of a 3AIC assignments.

Address calculations in 3AIC: `x[10] = 2`

- notationally represented as in C
- "pointer arithmetic" and address calculation with the available numerical operations

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

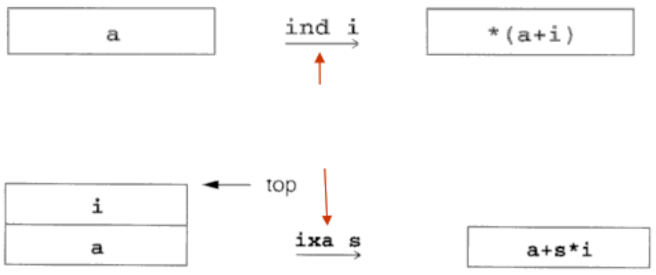


- 3-address-code data structure (e.g., quadrupel): *extended* (adding address mode)

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. It's a form of "pointer arithmetic". We will revisit the example in p-code; there, the translation will make use of an indexed access command `ixa`.

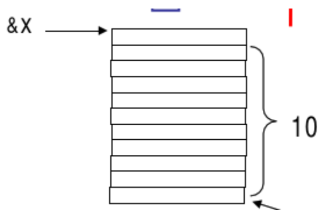
Address calculations in P-code: `x[10] = 2`

- tailor-made commands for address calculation



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

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



The efect of the two introduced commands `ixa` and `ind` is shown in the transitions in the picture, 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`). The difference correspond to the notions of l-values and r-values, we have seen before (but not in the context of array accesses). Also on the next slide, we see the difference between the two flavors of array-accesses (l- vs- r-value usage).

In the two pictures, the `a` is mnonic 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 discared from the stack: `sto` is the "destructive" write).

Array references and address calculations

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

- difference between left-hand use and right-hand use
- arrays: stored sequentially, starting at *base address*
- offset, calculated with a *scale factor* (dep. on size/type of elements)
- for example: for `a[i+1]` (with C-style array implementation)[6]

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

- `a` here *directly* stands for the base address

Array accesses in 3AI code

- *one* possible way: assume 2 additional 3AIC instructions
- remember: 3AIC can be seen as *intermediate code*, not as instruction set of a particular HW!
- 2 **new instructions**[7]

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

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

Source code

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

TAC

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

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 (like on this slide). 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).

[6]In C, arrays start at a 0-offset as the first array index is 0. Details may differ in other languages.
[7]Still in 3AIC format. Apart from the "readable" notation, it's just two op-codes, say `=[]` and `[]=`.

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 expressivity).

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".

Or "expanded": array accesses in 3AI code (2)

Expanding `t2=a[t1]`

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

Expanding `a[t2]=t1`

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

- "expanded" result for `a[i+1] = a[j*2] + 3`

```
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
```

Array accessses in P-code

Expanding `t2=a[t1]`

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

Expanding `a[t2]=t1`

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

- "expanded" result for `a[i+1] = a[j*2] + 3`

```
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
```

Extending grammar & data structures

- extending the previous grammar

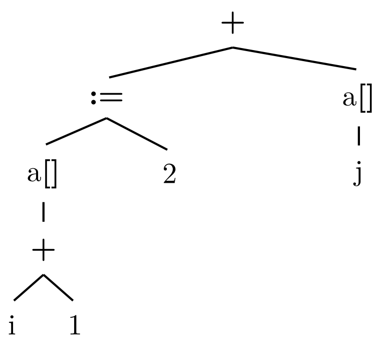
$$\begin{array}{rcl} exp & \rightarrow & subs = exp_2 \mid aexp \\ aexp & \rightarrow & aexp + factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \mathbf{num} \mid subs \\ subs & \rightarrow & \mathbf{id} \mid \mathbf{id}\,[\, exp\,] \end{array}$$

Extending the language (here with arrays) means extending the AST. That means we have to extend the tree definition from Listing 9.5. Actually, the extension is quite small: Compared to the the tree struction from Listing 9.5, 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 9.11: AST in C: additional `OpType`

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



Code generation for P-code

Listing 9.12 shows as C how one can generate code for the “array access” grammar from before. Compared to the correspinding procedure for code generation from Listing 9.7, the procedure `genCode` has one additional argument, a boolean flag. That has to do with the discinction we want to make (here) whether the argument is to be interpeted as address or not. And that in turn is related between so called L-values and R-values and the fact that the grammar allows “assignments” (written `x := exp2`) to be expressions themselvves. 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 9.7 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 9.12: Code generation 3AIC (arrays)

Access to records

Let's have also a short look to records. This time we don't show how to extend the abstract syntax tree declarations further or how to extend the `genCode` implementation in detail

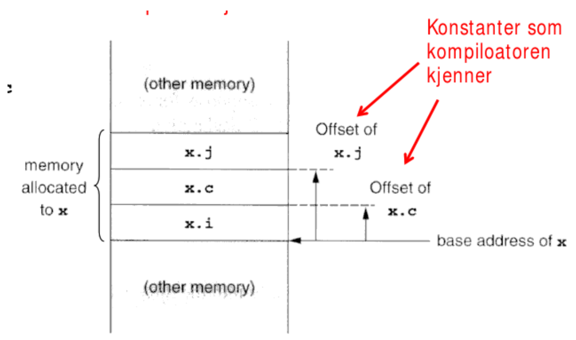
For dealing with records, one may consult also the remarks when discussing record types resp. the memory layout for different data types (in connection with the run-time environment). Records are not much more complex that arrays, it's only that the different slots are not "uniformely" sized. This one 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 compiler-time, but lies "in the future" in the phased design of the compiler. 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 9.13: Sample struct type declaration

Layout



- fields with (statically known) offsets from base address
- note:
  - goal: intermediate code generation *platform independent*
  - another way of seeing it: it's still IR, not *final* machine code yet.
- thus: introduce function `field_offset(x,j)`
- calculates the offset.
- can be looked up (by the code-generator) in the *symbol table*

⇒ call replaced by actual off-set

Records/structs in 3AIC

- note: typically, records are implicitly references (as for objects)
- in (our version of a) 3AIC: we can just use `&x` and `*x`

simple record access `x.j`

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

left and right: `x.j := x.i`

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

The second example shows record access a l-value and as r-value.

Field selection and pointer indirection in 3AIC

Next we cover an pointer indirection, actually in connection with records. 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 shows a typical example, a tree-like data structure. The following snippets then two typical examples making use of such trees, one on the left-hand side, one on the right-hand side of an assignment. The notation −> is C-specific, here used to “move” up or down the tree. The same example (the tree) will also be used to show the p-code translation afterwards.

C code

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

Treenode *p;
```

Listing 9.14: Some sample record type declaration (binary trees)

Assignments involving fields

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

3AIC

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

Structs and pointers in P-code

- basically same basic “trick”
- make use of `field_offset(x,j)`

3AIC

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

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

9.9 Control statements and logical expressions

So far, we have dealt with straight-line code only. The main "complication" were compound expressions, which do not exist in the intermediate code, neither in 3AIC nor in p-code. That required the introduction of temporaries resp. the use of the stack to store those intermediate results.

The core addition to deal with *control statements* here is the use of *labels*. Labels can be seen as "symbolic" respresentations 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*.

As a not-so-important side remark: Concretely in C, "booleans" and conditions operate also on more than just a boolean two-valued domain (containing `true` and `false` or `0` and `1`). In C, "everything" that's not `0` is treated as `1`. That may sounds not too "logical" but reflects how some hardware instructions and conditional jumps work. Doing some operations sets " hardware flags" which then are used for conditional jumps: jump-on-zero checks whether the corresponds flag is set accordingly. Furthermore, in functional languges, the phenomenon also occurs (but typically not called short-circuiting), and in general there, the dividing line between control and data is blurred anyway.

Control statements

- so far: basically *straight-line code*
- general (intra-procedural) control more complex thanks to *control-statements*
  - conditionals, switch/case
  - loops (while, repeat, for ...)
  - breaks, gotos, exceptions ...

important "technical" device: labels

- symbolic representation of addresses in static memory

- specifically named (= labelled) control flow points
- nodes in the *control flow graph*

- generation of labels (cf. also temporaries)

Intra-procedural means "inside" a procedure. *Inter*-procedural control-flow refers to calls and returns, which is handled by calling sequences (which also maintain, in standard C-like languages the call-stack of the RTE), as discussed in the chapter about run-time environments.

Concerning gotos: gotos (if the language supports them) are almost trivial in code generation, as they are basically available at machine code level. The "considered-harmful" qualification goes back to a famous (or infamous?) paper or letter by Dijkstra "Go To Statement Considered Harmful". Actually, when submitted the title of that piece was phrase differently, but the editor, Nikolaus Wirth, suggested a juicier one (Nikolaus Wirth is the guy behind Pascal, among other things). That letter was kind of the opening salvo or one important early salvo in the "structured programming wars"...

Loops and conditionals: linear code arrangement

Let's first fix the abstract syntax, extending the previous version. The additions are not very fancy, some 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 conditinals and loops in the abstract syntax; see later Figures 9.2 and 9.3

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if}\;(\;\textit{exp}\;)\;\textit{stmt}\;\;\textbf{else}\;\;\textit{stmt} \\ \textit{while-stmt} & \rightarrow & \textbf{while}\;(\;\textit{exp}\;)\;\textit{stmt} \end{array}$$

- challenge:
  - high-level syntax (AST) well-structured (= tree) which implicitly (via its structure) determines complex control-flow beyond SLC
  - low-level syntax (3AIC/P-code): rather flat, linear structure, ultimately just a *sequence* of commands

Arrangement of code blocks and cond. jumps

The two pictures show the "control-flow graph" of two structured commands (conditionals and loop). They should be clear enough. However, the pictures can also be read as containg 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. 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. Secondly, the two graphs use always the true-case as fall-through. Of course, the underlying intermediate code can

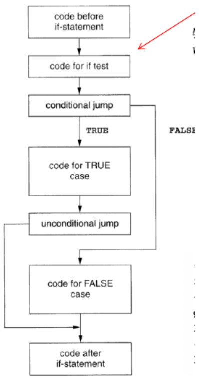


Figure 9.2: Conditional

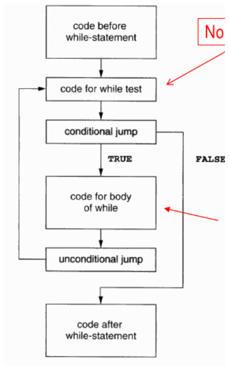


Figure 9.3: Conditional

support different formd of conditional jumps (like jump-on-zero and jump-on-non-zero) which may swap the situatiom. Our code will work with *jump-on-false* which explains the true-as-fall-through depiction.

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. 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 staight-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 very similar for both cases, both making use of labels).

To do the translation, we need to enhance the set of available "op-codes" (= available commands). We need a mechanism for *labelling* 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.

Jumps and labels for conditionals and loops

For conditionals **if** (E) **then** S_1 **else** S_2 and while loops **while** (E) S, the 3AIC is given in Listing 9.15, resp. in Listing 9.16.

```
<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 9.15: 3AIC for conditionals

```
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 9.16: 3AIC for while loops

For comparison, we show also the corresponding p-code in 9.17, resp. in Listing 9.18. 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 labelling and jumps to labels, in particular conditional jumps.

```
<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 9.17: 3AIC for conditionals

```
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 9.18: 3AIC for while loops

Boolean expressions

- two alternatives for treatment
  1. as *ordinary* expressions
  2. via *short-circuiting*
- ultimate representation in HW:
  - no built-in booleans (HW is generally untyped)
  - but “arithmetic” 0, 1 work equivalently & fast
  - bitwise ops which corresponds to logical $\wedge$ and $\vee$ etc
- comparison on “booleans”: $0 < 1$?
- boolean values vs. jump conditions

Short circuiting boolean expressions

The notation is C-specific, and a popular idiom for nifty C-hackers. For non-C users it may look a bit cryptic. A “popular” error in C-like languagues 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 “conjuction” 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 conjuct is **not** executed (to find out if it’s true or false), it’s jumped over. That’s known as “circuit evaluation”.

Short circuit illustration

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

- done in C, for example
- semantics must *fix* evaluation order
- note: logically equivalent $a \wedge b = b \wedge a$
- cf. to conditional expressions/statements (also left-to-right)

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

Pcode for `(x!=0) && (x==y)`

```
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
```

- new op-codes
  - **equ**
  - **neq**

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.

Grammar for loops and conditionals

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

- note: simplistic expressions, only *true* and *false*

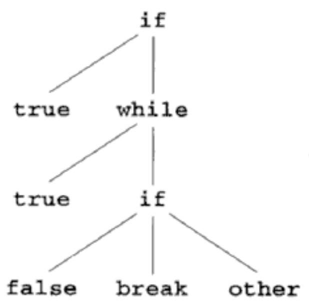
```
typedef enum {ExpKind, Ifkind, Whilekind,
              BreakKind, OtherKind} NodeKind;

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

type STreeNode * SyntaxTree;
```

Listing 9.19: C data structures for AST (control flow structures)

Translation to P-code

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

Syntax tree



P-code

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

Code generation

- extend/adapt `genCode`
- **break** statement:
  - absolute *jump* to *place afterwards*
  - *new argument*: label to jump-to when hitting a break
- assume: *label generator* `genLabel()`
- case for if-then-else
  - has to deal with one-armed if-then as well: test for NULL-ness
- side remark: **control-flow graph** (see also later)
  - labels can (also) be seen as *nodes* in the *control-flow graph*
  - `genCode` generates labels while traversing the AST
  - ⇒ implict generation of the CFG
  - also possible:
    - separately generate a CFG first
    - as (just another) IR
    - generate code from there

Code generation for for P-code

Listing 9.20 shows p-code generation for abstract syntax trees; the corresponding type declaration was shown earlier in Listing 9.19.

```
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 9.20: Code generation for p-code (control structures)

Listing 9.7

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. The most important or most high-level change in the procedure has to do with handling labels. In principle, we have seen what labels are and how to use them. Now, however, we have a concrete recursive procedure, traversing the tree. Now, the (small) challenge we have is: sometimes one has to inject a jump-command to some label which, at that point in the traversal, is not yet available, as not yet being generated. This is needed (for instance) when doing a break-statement in a loop. The way the code deals with it is that it 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 alterntaive ways to deal with this (mini-)challenge. Later we also have a look at an alternative ways, making use of two labels as argument.

More on short-circuiting (now in 3AIC)

- boolean expressions contain only two (official) values: true and false
- as stated: boolean expressions are often treated special: via short-circuiting
- short-circuiting especially for boolean expressions in *conditionals* and *while*-loops and similar
  - treat boolean expressions *different* from ordinary expressions
  - avoid (if possible) to calculate boolean value “till the end”
- short-circuiting: specified in the language definition (or not)

Example for short-circuiting

Source

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

3AIC

```
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
```

Alternative code generation for boolean expressions

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 9.20. In that older version, there was *no* short-circuiting. Now, in Listing 9.21, we want to include short-circuiting, and the part is handled by a separate sub-procedure `genBoolCode`; see Listing 9.22.

Note that `genBoolCode` takes to 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 9.20 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 9.22. 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 9.20, 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 proper 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 subexpression 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;
```

Listing 9.21: Alternative code generation for p-code (conditionals)

Anyway, Listing 9.21 shows a few cases, the one for "and" and "or", and also one comparison operator. The situation for "or" is also shown in Figure 9.4 (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 9.22: Alternative code generation for p-code (short-circuiting booleans)

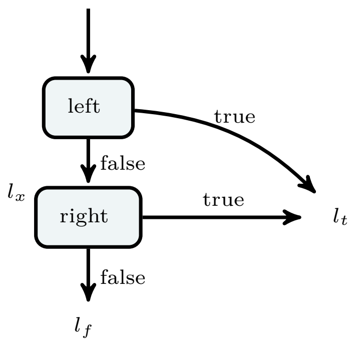


Figure 9.4: Short circuiting booleans, case "or"

Bibliography

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

Index