

Compiler: Codegeneration

Timo Heckel

Abstract—In the following we will take a look at the process of generating code out of preprocessed and optimized intermediate code to archive low level machine language. The question, how the codegenerator produces this language will be answered along with describing stack - and registermachines as two target-models for our output. Furthermore, algorithms for top-down parsing an abstract syntax tree and other purposeful methods of generating code are explained below. For better understanding, input-formats, memory management at runtime and evaluations orders are going to be shown.



1 INTRODUCTION

CODEGENERATION is an important part of compilers. In the following we are going to take a look at the codegeneration out of optimised intermediate code. The codegeneration results in executable machine-, assembly or object-code. How this works and which steps are needed to receive such an output is described in the following. At the beginning we need to define some important parts that are needed as an input for codegenerators. After that in '3. Processing', methods for processing and output-formats are going to be shown. Finally, a brief summary of the complete tasks of a code generator is provided.

1.1 Preliminary considerations

The codegenerator has some strict requirements. It must work efficiently but at the same time it needs to make effective use of resources on the target machine. The code which is going to be generated later must be without any errors and of high quality.

Inputs are very important for further processing. The compiler accepts intermediate code generated out of the sourceprogram. This code is produced by the front-end of the compiler and is delivered to the codegenerator.

We can now choose between different input styles which are going to be defined in 2

The intermediate code is delivered with informations about the runtime for data-objects which are very important for the compiler lateron.

It should be noted that details are depending on the final language - and the target operating system. Nevertheless, code-generation problems have to be eliminated before generating. Some possible problems for example are the type of the resourcemanagement, the set of commands, allocation of registers aswell as the sequence of the calculations.

2 INPUT

For further processing, the produced intermediate code of the front-end needs to be detailed and lexical/syntactical correct. This is necessary because we are going to use the intermediate code without any optimisations and transform it in the target format. Values of the names (dataobjects) are present (e.g bits, integer-numbers, pointer...) and required type-checks are performed earlier.

If these requirements are fulfilled, the selection of intermediate code can be placed.

2.1 Intermediate code styles

There are possibilities to display intermediate code with different representations. To generate a code, we need to choose the best-fitting Input - in future explanations we are focussing on abstract syntax-trees or tuples - but other representations are possible and may work too.

- The linear representation (flat, tuple-based, normally three-address-code) is for example the quadrupel. As the name indicates, every command and operator are splitted into a group of four elements. $A+B$ is going to be displayed as $(+, A, B, \text{tempVariable})$. More complex expressions are represented through a set of quadrupels. For this, A is going to be multiplied with the result of 100 plus the variable B . A possible quadrupel could be like this: $(+, 100, B, \text{tmp1})$, $(*, A, \text{tmp1}, \text{output})$.
- Another way of representing is the virtuell stack-based code. Pushdown-automaton-code as expressions for pushdown-automatons are using push and pop methods for placing or taking variables/constants to or from a stack. The resulting expressions are displayed through simple instructions. Example: $3 * x + 1 \Rightarrow \text{push } 3, \text{push } x, \text{multiply, push } 1, \text{add}$.
- Structured representations such as syntax trees or DAGs are used as a illustration for sentences and to show their structure. A syntax tree e.h. consists of stringed expressions and variables. Thus, the parent of a tree becomes an expression and the children will become a number or even a expression themself. So $2 + A$ could look like graphic 1.

Further as an extension, DAGs are a syntax tree like described above. It is a finite directed graph with no predefined directed cycles. But instead of having only childrens, it can have connections to other parents aswell as leading into a smarter form of connected syntaxtrees. The image in picture 2 shows an example of a DAG.

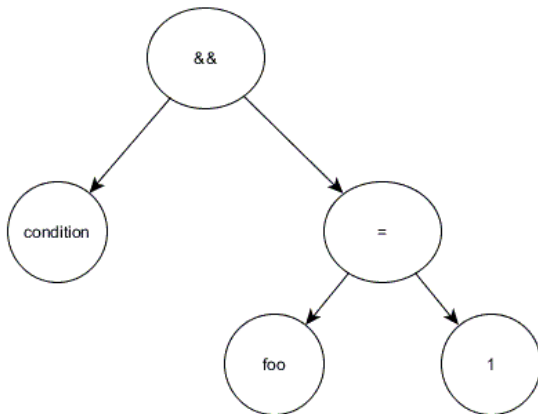


Figure 1. Example of a syntaxtree

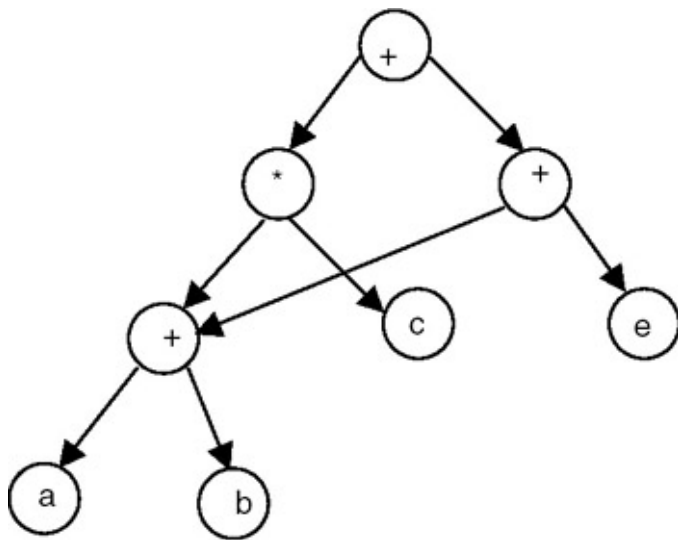


Figure 2. Example of a DAG syntaxtree

3 PROCESSING

After describing input variations and possible display methods of intermediate code this section is about how the compiler generates code. Each subsections describes a method used so we can summarize everything later. But first, what can we do with the pseudo-intermediate code shown in ???

The codegenerator understands intermediate code in e.g. in-memory data structure, or a special tuple- or stack-based code readable by the program. An intermediate representation of a program is more or less between the source-program and the destination language. IR needs to be fairly independent of source and target so its usability is maximised.

3.1 Why is IR used?

As mentioned above, we use intermediate code to be more independent. Compilers are made of a frontend aswell as a

backend. Imagine, we want to generate code from different sources for a machine. To do this, one can simply rewrite the front-end to generate IR from a different language - but the IR is still the same format. The backend used before can be reused and produces working machinecode. Also, rewriting the backend rather than the frontend results in more porability across different machines. This is more time-efficient because its only needed to rewrite some parts of the compiler and not the whole construction.

The step between IR and machinecode can also be used to perform machine independent optimizations of the input code. With this method, the compiler can be broken down to two simpler, more manageable pieces.

3.2 Considerations

Of course, as a prerequisite for our code generator, the preliminary considerations of the input and their own requirements apply. Once we have decided on a source language, the process of editing can begin.

3.2.1 Different procedures

Code generation can be either direct (as a routine that creates an object code from an input) or indirectly as an intermediate representation (= IR). IR is described as a level between source and destination. In the case of indirect codegeneration, a distinction is made between different levels, which are expressed in an ever-increasing approach to the target language.

In short it has three levels of tuple sophistication:

- **High-level:** Operations look like the source-code with arrays and other structured objects. It keeps the structure of the source language and it can be reconstructed. On high-level, there are no thoughts of using registers.
- **Medium-level:** Still independent of the target language but without structured objects. It breaks down complex declarations to only contain simple integer or float values. This level is perfect for optimizing code independent of the architecture.
- **Low-level:** No structured objects, operands are extremely close to the target language: now its architecture dependent so its also used for architecture dependent optimizations other then the medium level. The level deals heavily with issues such as stack frames and parameter passing.

3.3 Memory management / runtime

During code generation, various models of memory management are available for the machine. These models are defined in virtual machines or processors. One distinguishes between stack and register allocation.

3.4 Stackmachine

The stack machine model has a runtime stack as the only available memory. The implementation is done as an extensible array and has two pointers.

- Pointer (SP) points to the top-stack whereas

- Pointer (BP) points to the beginning of the area where variables (in this case the local ones) are located.

The stackmachine understands the set of commands shown in picture 3

Instruction		Actions
Push_Const	<i>c</i>	SP:=SP+1; stack[SP]:=c;
Push_Local	<i>i</i>	SP:=SP+1; stack[SP]:=stack[BP+i];
Store_Local	<i>i</i>	stack[BP+i]:=stack[SP]; SP:=SP-1;
Add_Top2		stack[SP-1]:=stack[SP-1]+stack[SP]; SP:=SP-1;
Subtr_Top2		stack[SP-1]:=stack[SP-1]-stack[SP]; SP:=SP-1;
Mult_Top2		stack[SP-1]:=stack[SP-1]*stack[SP]; SP:=SP-1;

Figure 3. Example of a stack commandset


Basically, a stack works with push, pop and mathematical arithmetic operations. An addition for example if performed by popping two numbers off the stack (with the LiFo-principle), adding them together and pushing them back on the stack after the calculation. Existing elements on the stack are not modified during this process allowed to be used in further processing after the operation.

An arithmetic operation like 'add' needs three memory operations so basically two read operations and one write back to the stack.

In order to be able to provide information for inputs which contain a plurality of arithmetic operations, the stack machine requires an accumulator which stores the uppermost position in the stack.

Results of arithmetic operations are always the accumulator - so calculations and/or calls are way faster because of this modification. Before an operation, we push the accumulator onto the stack after each compute. After the operation, we pop n-1 values out of the stack and delete any unused numbers.

Now, after every calculation we have a stack that is exactly like the output stack with addition of the accumulator which still points to the uppermost value. To visualize this method, we should take a look at image 4 shown below.



A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Figure 4. Example of a stack-operation with an accumulator included.

3.5 Registermachine

The register machine has, unlike the stack machine, arithmetic operations which can be performed explicitly on registers. (See also CISC). This model can in principle execute all operations on registers, like a real computer (PC). As a swap, the machine uses main memory, in which it stores unused intermediate results at runtime. This is one of the two types of commands:

- 1) The transfer between the main storage and the register (and bidirectional) and
- 2) arithmetic operations which are applied to registers.

Typically, an arithmetic operation is indicated with three subcommands: Register, destination register, constant and / or number. An example is provided in image 5.

Load_Mem	b, R1
Load_Mem	b, R2
Mult_Reg	R2, R1
Load_Const	4, R2
Load_Mem	a, R3
Load_Mem	c, R4
Mult_Reg	R4, R3
Mult_Reg	R3, R2
Subtr_Reg	R2, R1

Figure 5. Example of a register commandset

3.6 Selection of the evaluation order

Now that the code generator has received a syntax tree from the front end, the question arises in which order he processes it. The problem strongly affects the later resulting efficiency. One possible solution to this problem is to produce the code in the order specified by the intermediate code.

3.7 Backpatching

While processing the IR there might be statements that uses labels. So backpatching is the process where the codegenerator leaves empty statements when reaching goto-commands if the address is unknown during the first loop and filling these unknown in the second passing. This process is used during the generation of code for the specified machine.

3.8 Generating actual code

Having generated intermediate code, one is now faced with actual codegeneration to assembly. While the input looks like that:

```
li $t0, 3
lw $t1, (p)
lw $t2, (c)
add $t3, $t2, $t1
li $t4, 3
lw $t5, (b)
lw $t6, (c)
mul $t7, $t5, $t6
lw $t8, (y)
lw $t9, (p)
div $t10, $t8, $t9
```

Figure 6. Intermediate code in three-address-code representation

or even that:



Figure 7. Intermediate code in tree representation

How can we get code that actually looks like that which can be used in a runtime environment like MIPS Assembler?

```
p:=3
b:=p+c
m:=3
a:=b*c
x:=y/p
```

Figure 8. Example output in near-assembly language

4 OUTPUT FORMATS

Codegenerators can not only handle multiple inputs, but can also produce multiple output formats. We can divide in three main formats:

4.1 Absolute machinecode

Absolute machine code is stored in a fixed location in memory and can be executed immediately. A small program can be translated quickly and then executed.

4.2 Moveable machinecode

Movable machine language provides a separate translation of subprograms. A set of movable objects can be bound and loaded for execution. This gives a great flexibility because each subprogram can be translated individually (and called by objectmodule). If the moving is not automatically done by the target machine, the compiler needs to deliver informations for moving to allow including of individual translated modules.

4.3 Assembly

Assembly code can be easily generated through symbolical commands. Facilitations of the macrosystem assembler can be used. The negativ part of using assembly is the following, necessary assembling that is needed in order to allow the program to run. Though, this target language is good alternative for systems with less available memory on which the compiler needs to run multiple times.