Compiler: Codegeneration

generating code out of preprocessed and optimized intermediate code

Content

Introduction

- How do we receive output from a compiler?
- Define different intermediate representations (= IR)
- Method for processing and output formats

Preliminary considerations

- Codegenrator:
  - Strict requirements
  - efficient
  - Effective use of resources on the target machine

Components

- *Frontend*: deals with language, parsing, analyzing, optimizing and scanning
- Code generator is the **backend** of the compiler
  - Deals with the target language
  - Gets IR
  - Produces the right output for target machine

Input

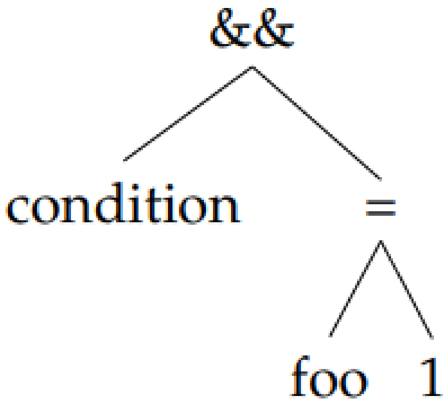
- Input is defined as IR
- IR = step between human-readable source and machine code
  - Used to be more independent
  - Backend can be reused
  - More portability across different machines
  - Time efficient

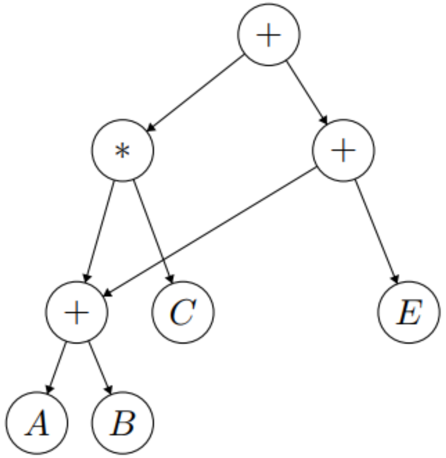
Important: For further processing the IR needs to be detailed and lexically / syntactically correct.

IR Styles

- Focus on syntax-trees / three-address-code
- Linear representation → tuple-based, normally three-address-code (= TAC)
  - A+B is going to be displayed as (+, A, B, Variable or Register)
  - Complex operations can be represented through set of quadruples
  - A * (100 + B) => (+,100,B,tmp1), (*,A,tmp1,output).
- Stack based code → simple instructions
  - Uses *push* and *pop* for placing or taking variables/constants to or from a stack
  - Example: [3 * x +1] => push 3, push x, multiply, push 1, add
- Structured representations → Syntax trees, directed acyclic graphs (DAG)
  - Consists of expressions and variables, every child or tree is a number or expression itself
  - DAG as extension for syntax trees, every node can have multiple parents

IR Styles: DAG example



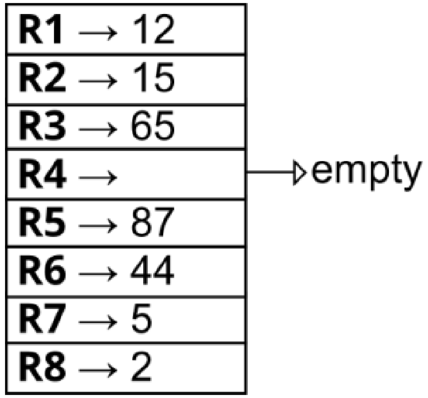


Example of a syntax tree: foo is set to one (foo = 1) and then foo is checked with a condition (foo && condition)

Example of a DAG syntax tree representing the expression (A+B)*C +((A+B) + E)

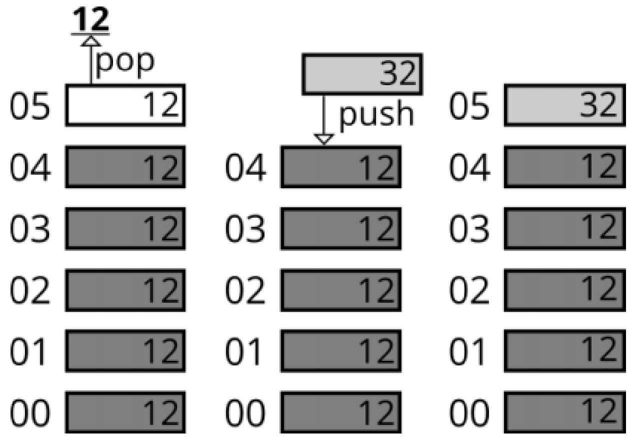
Processing – Considerations
Register

- Very fast computing memory
- Can store any kind of data (instructions, bit-sequences, characters…)
- Single memory location (accumulator)
- Implemented as register-files, flipflops, core memory…



- <u>Types:</u>
  - Accumulator (store data from memory)
  - General purpose registers (store data and results during execution)
  - Special purpose registers (instruction sets, program-counter)

Processing – Considerations
Stack (1/2)



- Popular dynamic data structure
- Functions
  - *Push* (places element on stack)
  - *Pop*/*Pull* (remove and return element from stack)
  - *Peek*/*Top* (return top element without removing)
- Used to save values which can later be processed
- Can only return the top element [LIFO (= last in, first out)]

Processing – Considerations
Stack (2/2)

- Extension: **stack frame**, a section in the stack containing values from parts of functions
- Frame pointer points to the top of the frame

Processing – Considerations
Stack frame

- Three stack frames for every recursive call
- Function-related variables will be saved
- Return informations of subprograms that are called before another one during runtime

```
def hello(x)
        if (x > 1)
                x--;
                hello(x);
        else
                return;
        end
end

hello(3)
```

Processing
Different Procedures

- Code generation can be
  - Direct, without IR
  - Indirect with IR
- IR = Level between source and destination
- Abstractions:
  - High-level → operations like source code with arrays and structured objects, no thoughts of using registers
  - Medium-level → Independent of the target but without structured objects, contains only simple integer or float values
  - Low-level → No structured objects, operands extremely close to target language, architecture dependent, deals with issues like stack frames

Processing
Code examples

```
double a[20][10];
.
.
.
for (int i = 0; i < n; i += di)
        a[i][j+2] = j;
.
```

```
        i := 0
L1:
        if i >= n goto L2
        t0 := a[i]
        t1 := j + 2
        t2 := t0[t1]
        *t2 := j
        i += di, goto L1
L2:
```

```
        i := 0
L1:
        if i >= n goto L2
        t0 := i * 80
        t1 := a + t0
        t2 := j + 2
        t3 := t2 * 8
        t4 := t1 + t3
        *t4 := j
        i := i + di
        goto L1
L2:
```

```
        r0 := 0
        r1 := j
        r2 := n
        r3 := di
        r4 := a
L1:
        if r0 >= r2 goto L2
        r5 := r0 * 80
        r6 := r4 + r5
        r7 := r1 + 2
        r8 := r7 * 8
        r9 := r6 + r8
        f0 := tofloat r1
        *r9 := f0
        r0 := r0 + r3
        goto L1
L2:
```

Processing

Memory management, runtime and destination machine

- During code generations various **models of memory management** are available
- Distinguishes between **stack** and **register machine**

Processing

Stack machine

Instruction		Actions
Push_Const	c	SP:=SP+1; stack[SP] := c;
Push_Local	i	SP:=SP+1; stack[SP] := stack[BP+i];
Store_Local	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

- Runtime stack as only available memory
- Implementation as an array with to pointer
  - Stack pointer (= SP) points to the top of the stack
  - Begin pointer (= BP) points to the beginning area where variables are located
- The stack machine understands a set of commands
  - Push_Const increments the stack pointer and adds c to the current location of the SP.
  - Push_Local also increments the stack pointer but then stores the value on position [BP+i]
  - Store_Local pops and stores result back in local storage

Processing
Stack machine

- works with push, pop and arithmetic operations
- Upgrade: single register called accumulator
  - Save the last result in the accumulator for further processing
  - calculations and/or calls are way faster
  - after every calculation the stack is exactly like the one without an accumulator but the accumulator still points to the uppermost value

Code	Accumulator	Stack
acc $\leftarrow$ 3	3	<init>
push acc	3	3, <init>
acc $\leftarrow$ 7	7	3, <init>
push acc	7	7, 3, <init>
acc $\leftarrow$ 5	5	7, 3, <init>
acc $\leftarrow$ acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc $\leftarrow$ acc + top_of_stack	15	3, <init>
pop	15	<init>

Code	Stack
push 5	5, <init>
push 7	7, 5 <init>
add_top2	12 <init>
push 3	12, 3, <init>
add_top2	15, <init>

Processing
Register machine

- Gets the name because it uses one or more registers that can hold a single positive integer value
  - Can execute all operations on them
  - Register: two basic operations → transfer between main storage and the register aswell as arithmetic operations.
  - Typically, arithmetic operationi is indicated with three subcommands:
    - Register, destination register, constant and or number

```
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
```

Processing
Register machine vs stack machine

- Stack code easier to generate (but harder for register allocation because of the temp locations)
- Register machine code is harder to generate but in the end more efficient
- Stack machine needs push and pop while a register machine only need one line for adding a value.
- Overheads while pushing and popping are non-existent in a register
  - Instructions execute way faster this way
- Register based optimisations
- Register instruction is typically larger

Processing
Selection of the evaluation order

- Define in which order the backend processes the IR
- Strongly effects the efficiency of the target code

Processing
Selection of the evaluation order

The compiler gets the following intermediate calculation from

$(a + b) - (c + d) * e$

$$\begin{aligned} t1 &:= a+b \\ t2 &:= c+d \\ t3 &:= e*t2 \\ t4 &:= t1-t3 \end{aligned}$$

Not optimized

$$\begin{aligned} t2 &:= c+d \\ t3 &:= e*t2 \\ t1 &:= a+b \\ t4 &:= t1-t3 \end{aligned}$$

Not optimized

Processing
Selection of the evaluation order

And we receive the following output:

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

NOT OPTIMIZED

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
ADD b,R0
SUB R1,R0
MOV R0,t4
```

OPTIMIZED

Processing
Function: getreg

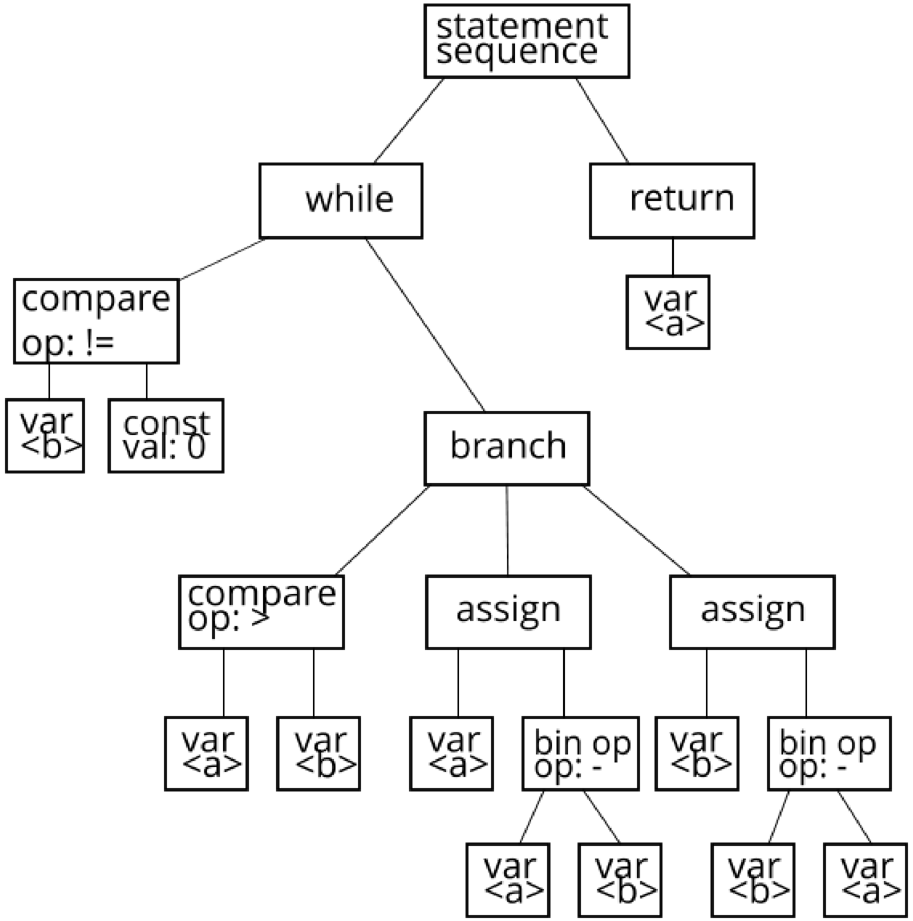
- Used to determine the status of a register and possible locations of values
- Returns a register that can be used later

Processing
Generating actual code

- Input program:

```
if a = 0 then
    result = b
else
    while b != 0
        if a > b then
            a = a - b
        else
            b = b - a
        end
    end
end
```

Processing
Generating actual code




```
a := ? ;defined in function call
b := ? ;also defined in function call

L1:
    if a = 0 goto L5
    goto L2
L2: ;running through loop
    if b = 0 goto L5

    if a > b goto L3
    if a < b goto L4


L3:
    a = a - b, goto L2

L4:
    b = b - a, goto L2

L5:
    ;end of program, returning.
```

Processing
Generating actual code

- We receive output like this:

```
li $t0, 3
lw $t1, (p)
lw $t2, (c)
add $t3, $t2, $t1
...
```

- Complex and hard to read

Processing
Generating actual code

- Call a function called ’getreg’ → returns register or memory location
- Check the address-descriptor → get location for the variable
- the instruction for <u>[op ’z’, ’L’]</u> is going to get generated
  - find a register for that code
  - prefer register over memory
- Modify the register-descriptor to free up unused registers
- Load statement:
  - Update the register descriptor
  - Update the address descriptor

Output formats
Absolute machine code

- Fixed location in memory, can be executed immediately
- A program can be translated quickly and then executed
- Jump to an exact location or address and/or read from an exact address
- Needs placeholder, filled with absolute addresses in further processing by a linker

Output formats
Moveable machine code

- provides a separate translation of subprograms
- A set of movable objects can be bound and loaded for execution from RAM
- relative address is different from the absolute address
- great flexibility, each subprogram can be translated individually

Output formats
Assembly

- low level programming language
- Easy to generate because of high abstraction level
- Near machine code level and instructions
- Facilitations of the macrosystem assembler
- Assembling is needed
- good alternative for systems with less available memory

Postprocessing

- Peephole optimizations
- Machine code optimization