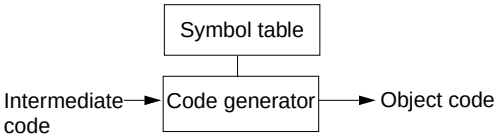
Code generation

- Difficult to generate good code
- Simple to generate bad code
- There are code-generator generators

Requirements for code generation

- Correctness
- High quality
- Efficient use of the resources of the target machine
- Quick code generation

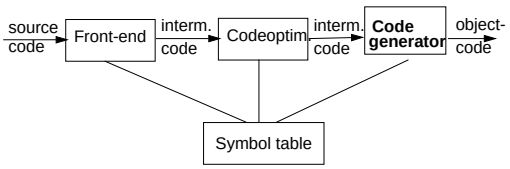
Different forms of intermediate code and object code



- Syntax tree
- Postfix code
- Triples
- Quadruples
- Code for abstract stack machine

- Absolute machine code
- Relocatable machine code
- Assembler code

Code generator environment



Input:

- Internal form

Prerequisites:

- No lexical, syntactic or semantic errors.
- The symbol table contains all information required, e.g. type and size, addresses, offset etc.

Output:

- Object code (absolute, relocatable or assembler)

Disadvantages and advantages of different types of object code

Absolute code

Generated code is placed directly in memory and execution starts immediately.

Example:

PASSGO: good for small jobs (student compiler)

Turbo Pascal: has a switch for absolute code or relocatable code.

+ Quick
– Can not call modules in other languages
– Can not be compiled separately

Relocatable code

Most common, linking and loading needed.

– Slower
+ Can be compiled separately
+ Can call modules in other languages
+ Flexible

Assembly code

– Slowest (assembling, linking, loading required)
+ Makes code generation much simpler (you do not need to handle future references)

Target machine

Generation of code for the statement: `A := B + C`

- **Stack machine**

```
PUSH A
PUSH B
PUSH C
ADD
STORE
```

- **Register machine**

```
MOVE B,R0
ADD C,R0
MOVE R0,A
```


Problems with code generation

1. Choice of instructions

   Example: `A := A + 1`

   ```
   INC  A        or        MOVE A,R0
                           ADD  1,R0
                           MOVE R0,A
   ```

2. Choice of order

   How should arithmetic expressions be calculated so that `LOAD` and `STORE` instructions are minimised?

   Consider numerically instable cases (*overflow, underflow*).

3. Register use

   - Very difficult problem (messy when an operation requires several registers).
   - Very important for the quality of the code.
   - Certain registers are used for special purposes, e.g. the stack pointer.
   - Keep variables in registers as much as possible (problems when trouble-shooting).

Machine model

- 8 registers `R0-R7`
- Machine operations are similar to PDP-11:

  Op source, destination

Example:

```
MOVE A,B          ; B := A
ADD  A,B          ; B := B + A
SUB  A,B          ; B := B - A
```

- Cost table

Source	Destination	Cost
register	register	1
register	memory	2
memory	register	2
memory	memory	3

Example: `A := B + C`

1.
```
MOVE B,R0       ;2
ADD  C,R0       ;2
MOVE R0,A       ;2   ⇒ Total cost = 6
```

2.
```
MOVE B,A        ;3
ADD  C,A        ;3   ⇒ Total cost = 6
```

3.
```
ADD  R2,R1      ;If R2 contains B and
                ;R1 contains C
MOVE R1,A       ;and C is not alive after
                ;this statement
                ;   ⇒ Total cost = 3
```

4.
```
ADD  R2,R1      ;Same conditions as in (3),
                ; but the value of A is in R1
                ;   ⇒ Total cost = 1
```

There is a lot to be gained with good register allocation.

Examples of code generation algorithms

1. Macro-expansion of internal form
2. "A simple code generation algorithm"
   (using address and register descriptors)
3. Code generation from DAGs
4. Code generation using code templates (pattern matching)

Macro-expansion of internal form

Each quadruple is translated to one or more instructions.

+ very simple
- poor quality code (slow and requires much memory)
- poor use of registers

A simple code generation algorithm (ASU p. 535)

Prerequisites:

Input: sequence of quadruples grouped in *basic blocks*.

Output: assembly code (or machine code)

- The result is kept in registers as long as possible and is moved into memory only
  1. if the register is needed for another calculation
  2. at the end of a *basic block*
- ***Basic block***: sequence of statements which can only be traversed sequentially from the first instruction to the last.
- A variable `x` **is used locally** after a point *p* if `x`'s value is used within the block after *p* before an assignment to `x` (if any) is made.
- All variables (except temporary variables) are assumed to be **alive** (i.e. they can be used before they are assigned a value) after a *basic block*.

`reg(R)`: *register descriptor*, specifies the content of register `R`

`adr(A)`: address descriptor, specifies where the value of `A` is (possibly in both register and memory)

Code for quadruple: `op B C A`

is generated using the following algorithm:

1. `L := GETREG()` (defined below)
2. `B' := adr(B)` prefer register if several addresses
   if `B'<>L` generate `MOV B',L`
3. `C':= adr(C)`
   generate `op C',L`
   `adr(A):= L`
   if `L` is a register, `reg(L):= A`
4. If `B` and/or `C` are not used locally or are alive after the block, free the registers where `B` and/or `C` are.

When all the quadruples in a *basic block* have been traversed,* `MOV` is generated for the non-temporary variables that are found only in the register.

Definition of `GETREG()`:

a) If `adr(B)` is a register and there is no other variable in this register and if `B` is not alive after the block or is used locally:

```
L := adr(B)
adr(B) := nowhere
return L
```

b) Otherwise return an empty register

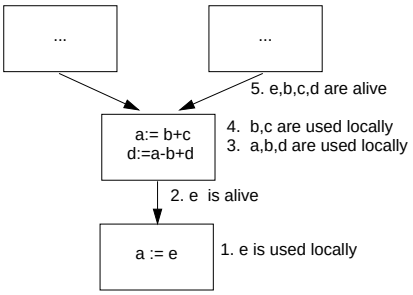
- Otherwise (if there are no more registers)

  If `A` is used locally, empty a register by generating a `MOV` to a temporary

c) for some other variable and return the register.

d) Otherwise return `adr(A)`

Example of "is used locally" and "is alive"



Exercise 1:

Generate code for the following statement using the algorithm presented above:

```
X := Y * Z + U + V * W
```

i.e. for the following quadruples:

op	B	C	A
*	Y	Z	T1
+	T1	U	T2
*	V	W	T3
+	T2	T3	X

The machine is assumed to have 2 registers

Exercise 2:

Generate code for the following quadruples. To understand the algorithm better, assume there are only two registers, R0 and R1, which can be used:

```
t1 := m + n
t2 := a + b
t3 := k + t2
 z := t1 - t3
```

Show the contents of the address- and register-*descriptors*, and calculate the cost of each quadruple and finally the cost of the whole *basic block*.

Assume that all variables are alive after the block, but temporary variables are not alive after the block.

The code should be as shown below, but details have not been included. Check that you get the same result.

```
MOV m, R0     m ⟶ R0
ADD n, R0     t1 in R0
MOV a, R1     a ⟶ R1
ADD b, R1     t2 in R1
MOV R0, t1  ;empty R0, it is needed for something else now!
MOV k, R0   ;t3 will be in R0 after next add
ADD R1, R0  ;free R1, as it is not used again in the block
MOV t1, R1  ;t1 is in memory, load it into a register
SUB R0, R1  ;calculated t1-t3, the result is in R1
MOV R1, z   ;end of block=> save R1 in z's memory address
```

Total cost of the block = 18 (10 instructions)

A heuristic improvement of the algorithm

You can improve the number of `LOAD` and `STORE` operations by changing the place of some quadruples. The replacement can be performed as the calculations are not dependent on each other. For example, compare the quadruple sequences below:

Given quads.	Replaced quads.
`t1 := m + n`	`t2 := a + b`
`t2 := a + b`	`t3 := k + t2`
`t3 := k + t2`	`t1 := m + n`
`z := t1 - t3`	`z := t1 - t3`

The code for the new quadruple sequence is:

```
MOV a, R0
ADD b, R0
MOV k, R1
ADD R0,R1
MOV m, R0
ADD n, R0
SUB R1, R0
MOV R0,  z
```

Cost=14 (8 instructions)

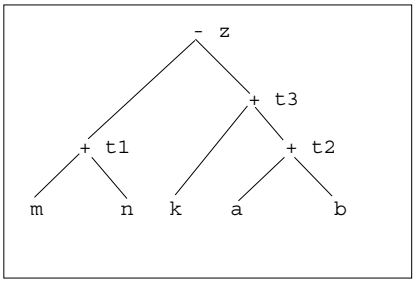
Why?

What we did above was to delay the code generation for the left argument, for each quadruple, as late as possible. In this case the only possible delay will be the code generation for `t1`, which is performed just before the code generation for `z`.

NODE-LISTING-algorithm

You can use a DAG (Directed Acyclic Graph) and the `NODE-LISTING` algorithm in ASU, page 560 to get the required order for the quadruples which code is to be generated for.

Study the `NODE-LISTING` algorithm and check that the order of the quadruples corresponds to the replaced sequence above.



DAG for the given sequence of quadruples

Code generation from a DAG

DAG: *Directed Acyclic Graph*

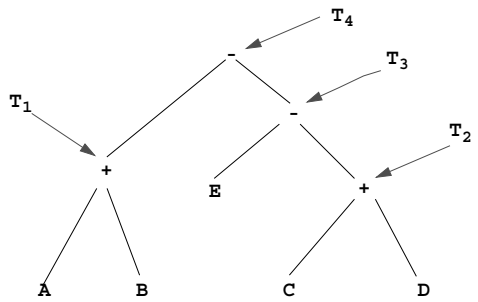
An abstract syntax tree is a special case of a DAG.

Input: DAG (quadruples in tree form!)

Output: assembly code (or machine code)

Example:

$T_1 = A + B$
$T_2 = C + D$
$T_3 = E - T_2$
$T_4 = T_1 - T_3$



The algorithm has two phases:

- Calculate the need for registers for each sub-tree
- Traverse the tree and generate code. The register need guides the traversal.

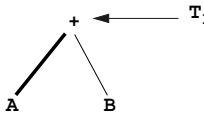
Phase 1: Calculate the register needs for each sub-tree

For binary trees (each operator has 2 operands).

n = a node

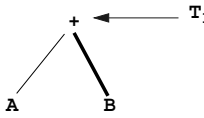
LABEL(n) = the register needs for the sub-tree with node **n**.

- If **n** is a left leaf ⇒ **LABEL(n):= 1**



MOVE A,R0

- If **n** is a right leaf ⇒ **LABEL(n):=** 0



ADD B,R0

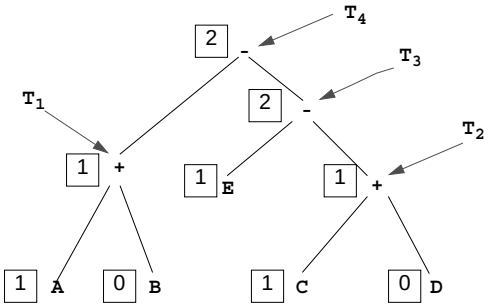
- If left and right children have different register needs:

```
LABEL(n):= max(LABEL(n.left),
               LABEL(n.right))
```

- If left and right children have the same register needs:

```
LABEL(n):= LABEL(n.left) + 1
```


Example:



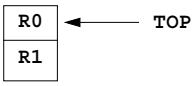
x denotes the register needs for the node

Easy to calculate the register needs using syntax-directed translation with bottom-up parsing (postorder traversal).

Phase 2: Code generation

Data structures:

RSTACK: the register stack, initialised with all available registers.



TSTACK: Stack for temporary variables.

Procedures:

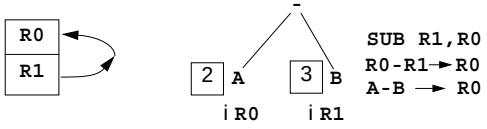
Gencode(n):

Recursive procedure which generates code for sub-trees with root **n**.

The result is placed in **RSTACK[TOP]**

Swap(RSTACK)

swaps the top two elements at the top of the stack.




```
SUB R1,R0
R0-R1→R0
A-B → R0
```

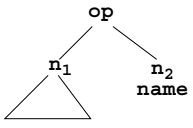

Gencode(n): 5 different cases (0 – 4) depending on the register needs for the sub-trees:

Case 0: n = left leaf ⇒

```
Print('MOVE ', name, RSTACK[TOP]);
```

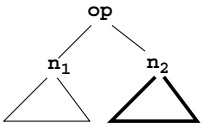
Case 1: If **n** is a node with children n_1 and n_2 (left and right children, resp.) and **LABEL**(n_2) = 0 ⇒

```
Gencode(n1);
Print(op, name, RSTACK[TOP]);
```



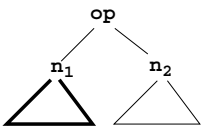
Case 2: If $1 \le$ **LABEL**$(n_1) <$ **LABEL**(n_2) and **LABEL**$(n_1) <$ **r** where **r** is the number of registers in the machine.

```
Swap(RSTACK);
Gencode(n2);
savereg := Pop(RSTACK);
Gencode(n1);
Print(op, savereg, RSTACK[TOP]);
Push(savereg);
Swap(RSTACK);
```



Case 3: If $1 \le$ **LABEL**$(n_2) \le$ **LABEL**(n_1) and **LABEL**$(n_2) <$ **r** where **r** is the number of registers in the machine.

```
Gencode(n1);
savereg := Pop(RSTACK);
Gencode(n2);
Print(op, RSTACK[TOP], savereg);
Push(savereg);
```



Case 4: Both n_1 and n_2 have register needs $\ge$ **r** $\Rightarrow$ store the result in the temporary stack **TSTACK**.

```
Gencode(n2); { recursive call }
T := Pop(TSTACK);
Print('MOVE ', RSTACK[TOP], T);
Gencode(n1);
Print(op, T, RSTACK[TOP]);
Push(T);
```

(See Fig 9.26 in ASU)

Code generation using code templates, i.e. code generation using pattern matching

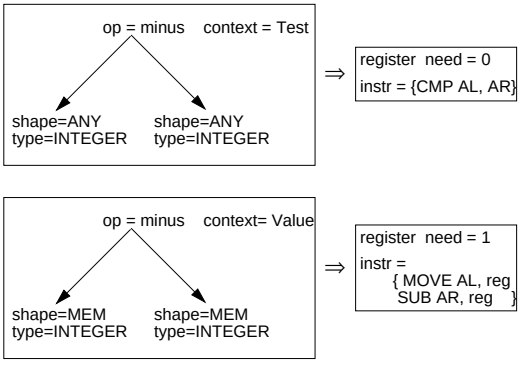
- The target machine is described using a set of code templates.
- Corresponding instructions are attached to each code template.
- The tree is matched with the code templates top-down.
- If there is a ”complicated” code template which corresponds to the whole tree $\Rightarrow$ write the corresponding instructions.
- Otherwise match with the children of the node.

NB! Code can be generated in different contexts.

```
IF (A-B)=0 THEN A:=B {CMP A,B}    Test

D:=(A-B)*C           {MOV A,R0
                      SUB B,R0}   Value
```


Examples of code templates:



AL = *Address left*, AR = *Address right*.

Shape can be in register, memory or on the stack.

The Graham-Glanville method (1978)

Idea: use a LR-parser for the matching process.

- Quick matching
- Compact specification of target machine using a CFG — Context free grammar.