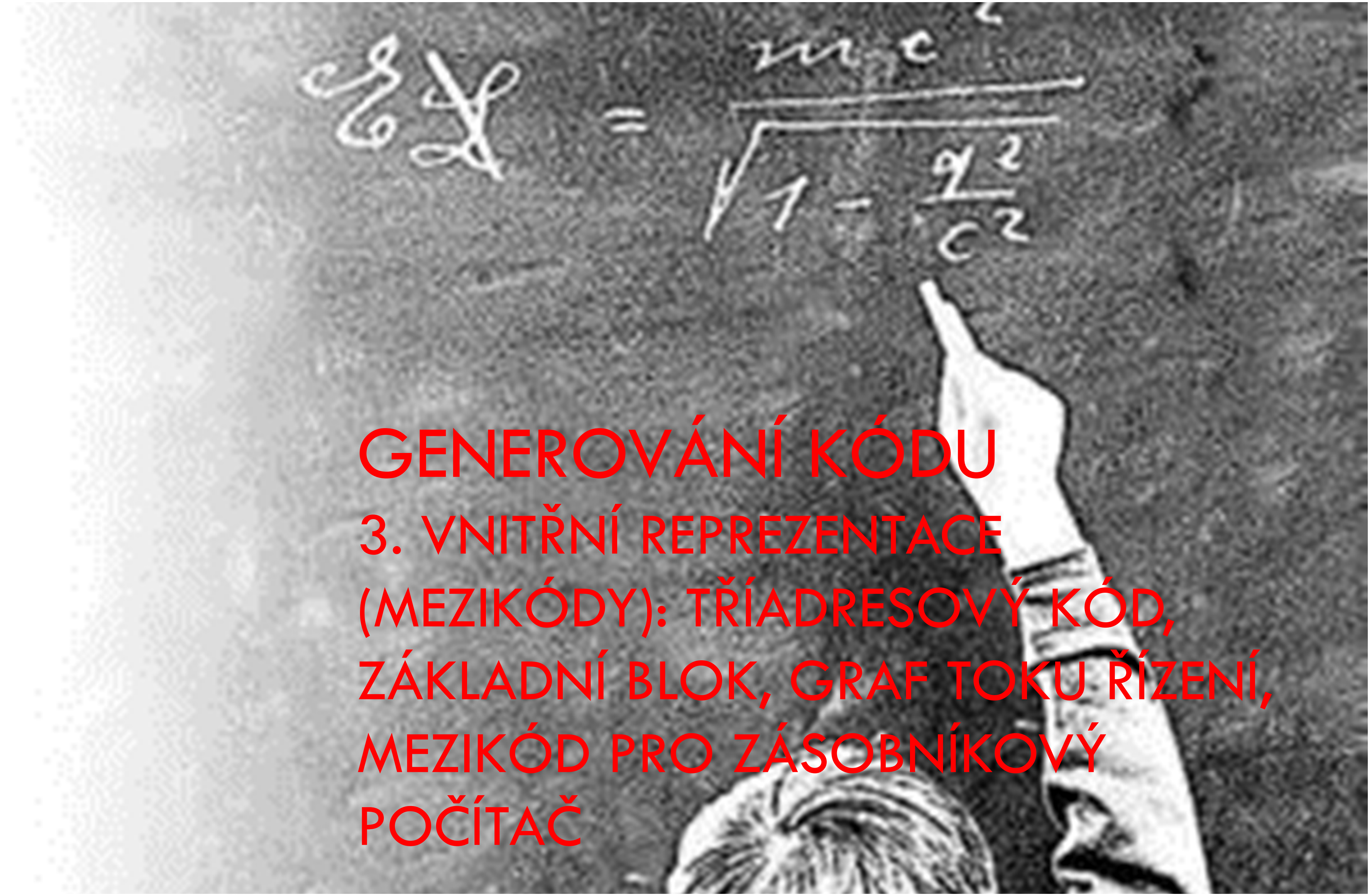


GENEROVÁNÍ KÓDU

3. VNITŘNÍ REPREZENTACE (MEZIKÓDY): TŘÍADRESOVÝ KÓD, ZÁKLADNÍ BLOK, GRAF TOKU ŘÍZENÍ, MEZIKÓD PRO ZÁSOBNÍKOVÝ POČÍTAČ

2011 Jan Janoušek
MI-GEN

EVROPSKÁ UNIE
Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

Three address code (3AC)

Introduction

- linear code

- ways of generating:
  - Syntax/directed translation produced by the frontend (the translation is defined by an attribute translation grammar)
  - From an AST by traversing the AST

Main properties

- A sequence of simple instructions with **at most one** operation on the right hand side, the result of each instruction is assigned to a temporary

  ie. x = y **op** z

- an unfinite number of temporaries (something like registers and memory cells)

- Example: x+y*z

t1 = y * z
t2 = x + t1

Another example

a = b*-c+b*-c

t1 := -c	t1 := -c
t2 := b * t1	t2 := b * t1
t3 := -c	a := t2 + t2
t4 := b * t3	
t5 := t2 + t4	
a := t5	

Parameters of operations in 3AC

- Names: source-programs names – usually implemented as pointers to the symbol table
- Compiler-generated temporaries
- Constants

Further properties

- Arrays:

x [y] = z
x = y [z]

- Labels:

L:

3AC instructions

- assignments with binary operator: x = y **op** z
- assignments with unary operator: x = **op** y
- copy instructions: x = y
- Control flow instructions:
  - ifFalse x goto L
  - ifTrue x goto L
  - goto L

3AC instructions

- Procedure calls

param *x1*

param *x2*

…

param *xn*

call *p*, *n*

- Addresses and pointer assignments:

x=&y

x=*y

*x=y

Example: translation of a statement

Statement

If *expr* then *stmt1*

is translated to this 3AC code:

code to compute *expr* into *x*
ifFalse *x* goto *after*
code for *stmt1*
after:

Most common implementations of 3AC

- Quadruples
- Triples
- Indirect triples

Quadruples

best for easy optimization

Names of temporaries

	op	arg1	arg2	res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Triples

Does not contain names of temporaries, which are replaced by position of instructions

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Indirect triples

Positions of instructions are connected with pointers to triples (the right table)

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

	stmt
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Generating 3AC

Generating 3AC

- As a syntax-directed translation:
  - can be described by attribute grammar – the resulting code is contained in a synthesized attribute,
  - the implementation is usually joined together with the parsing phase in the frontend
- can be performed by walking (traversing) the abstract syntax tree, which is generated by the frontend

Syntax-directed generating 3AC

- Attributes for generating 3AC, which are added to each nonterminal symbol
  - place – the name of a temporary containg the result
  - code – the resulting sequence of 3AC instructions for the nonterminal
  - label – absolute or relative address to the 3AC code

Example

- $E \rightarrow E_R + T$

  ```
  E.p = newtemp
  E.c = concat(E_R.c, T.c,
     gen(E.p=E_R.p + T.p))
  ```

- $E \rightarrow T$

  ```
  E.p = T.p
  E.c = T.c
  ```

- $T \rightarrow T_R * F$

  ```
  T.p = newtemp
  T.c = concat(T_R.c, F.c, gen(T.p
     = T_R.p * F.p))
  ```

- $T \rightarrow F$

  ```
  T.p = F.p
  T.c = F.c
  ```

- $F \rightarrow (E)$

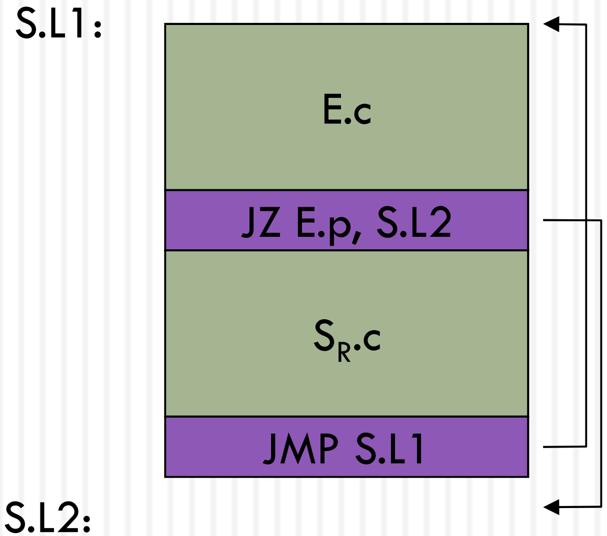
  ```
  F.p = E.p
  F.c = E.c
  ```

- $F \rightarrow \mathbf{id}$

  ```
  F.p = id.p
  F.c = ""
  ```

Another example - while

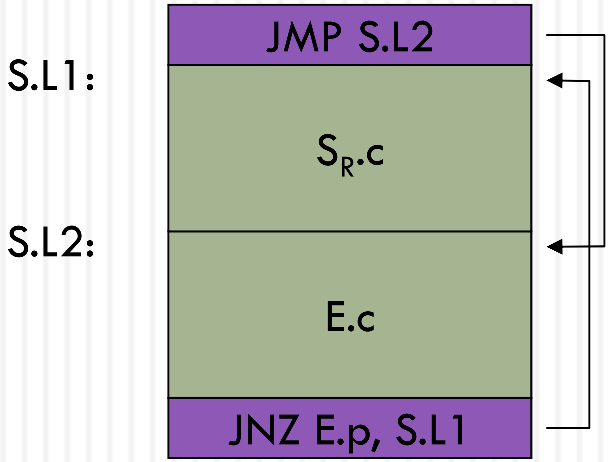
- $S \rightarrow$ **while** E **do** S_R




```
S.L1 = curradr
S.L2 = curradr + E.c.size
   + SR.c.size + 2
S.c = concat(E.c, gen(JZ
   E.p, S.L2), SR.c,
   gen(JMP S.L1))
```

while example once more and better

- S→ **while** E **do** S_R




```
S.L1 = curradr + 1
S.L2 = curradr +
  SR.c.size + 1
S.c = concat(gen(JMP
  S.L2), SR.c, E.c,
  gen(JNZ E.p, S.L1))
```

IR: Basic blocks, flow graph

- Important for target code generation and optimizations

Basic Block (BB)

- Maximal sequence of consecutive 3AC instructions where the flow of control can only enter and can only leave the block through the first instruction and the last instruction of the block, respectively.

- The first instruction of BB is called leader.

- For each leader, its basic block consists of itself and all instructions up to but not including the next leader

Partitioning a code into BBs

- Fixing leaders
  - The first 3AC instruction is a leader.
  - Any instruction that is the target of a conditional or unconditional jump is a leader.
  - Any instructions that immediately follows a a conditional or unconditional jump is a leader.

Flow graph

- Graph representation of IR.
- Nodes are BB. Nodes represents particular computations.
- Suitable for optimisations as well as for representations.

Flow graph

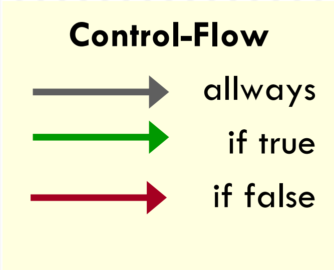
- Oriented edges represent the flow of control.
- There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B:
  - There is a jump from the end of B to the beginning of C.
  - C immeditely follows B in the original order of 3AC instructions and B does not end with a unconditional jump.

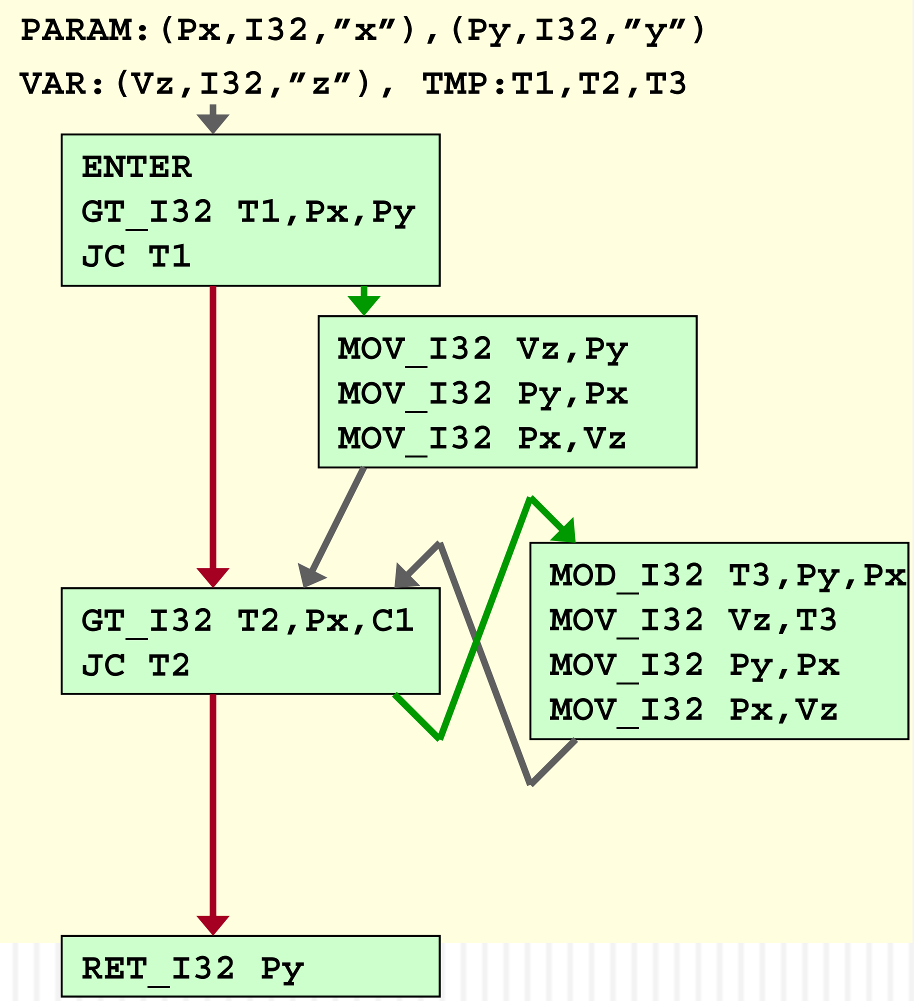
An example of BBs and flow graph

```
int gcd( int x,
int y)
{ int z;
…
  if ( x > y )
  {
    z = y;
    y = x;
    x = z;
  }
  while ( x > 0 )
  {
    z = y % x;
    y = x;
    x = z;
  }
  return y;
}
```

An example of BBs and flow graph

```
int gcd( int x,
int y)
{ int z;
…
  if ( x > y )
  {
    z = y;
    y = x;
    x = z;
  }
  while ( x > 0 )
  {
    z = y % x;
    y = x;
    x = z;
  }
  return y;
}
```





Stack-based IRs

Just another kind of IR to mention: **stack based** intermediate code

```
int gcd(int a,int b)
{
while (a != b) {
      if (a > b)
            a -= b;
      else
            b -= a;

      }
return a;
}
```

```
# javap -c Gcd
Method int gcd(int, int)
0     goto 19
3     iload_1 // Push a
4     iload_2 // Push b
5     if_icmple 15 // if a <= b goto 15
8     iload_1 // Push a
9     iload_2 // Push b
10    isub // a - b
11    istore_1 // Store new a
12    goto 19
15    iload_2 // Push b
16    iload_1 // Push a
17    isub // b - a
18    istore_2 // Store new b
19    iload_1 // Push a
20    iload_2 // Push b
21    if_icmpne 3 // if a != b goto 3
24    iload_1 // Push a
25    ireturn // Return a
```

stack based IR - Advantages

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers (stack based)
- Often compact

stack based IR - Disadvantages

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization

stack based IR

- Recently again popular and frequently used:
  - Java bytecode – interpreted and/or compiled by Java Virtual Machine
  - Common Intermediate Language (CIL), formerly called Microsoft Intermediate Language (MSIL) – interpreted and/or compiled by .NET runtime or MONO runtime