6. Code Generation

6.1 Overview
6.2 The .NET Common Language Runtime (CLR)
6.3 Items
6.4 Expressions
6.5 Assignments
6.6 Jumps and Labels
6.7 Control Structures
6.8 Methods

Responsibilities of the Code Generation

Generation of machine instructions

- selecting the right instructions
- selecting the right addressing modes

Translation of control structures (if, while, ...) into jumps

Allocation of stack frames for local variables

Maybe some optimizations

Output of the object file

Common Strategy

1. **Study the target machine**
   registers, data formats, addressing modes, instructions, instruction formats, ...
2. **Design the run-time data structures**
   layout of stack frames, layout of the global data area, layout of heap objects, layout of the constant pool, ...
3. **Implement the code buffer**
   instruction encoding, instruction patching, ...
4. **Implement register allocation**
   irrelevant for Z#, because we have a stack machine
5. **Implement code generation routines** (in the following order)
   - load values and addresses into registers (or onto the stack)
   - process designators (x.y, a[i], ...)
   - translate expressions
   - manage labels and jumps
   - translate statements
   - translate methods and parameter passing

6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

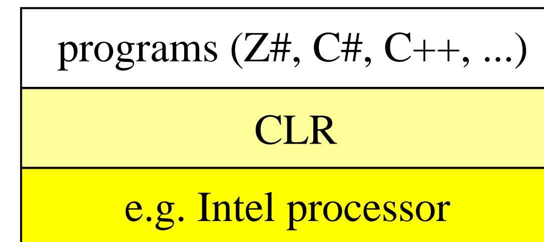
6.7 Control Structures

6.8 Methods

Architecture of the CLR

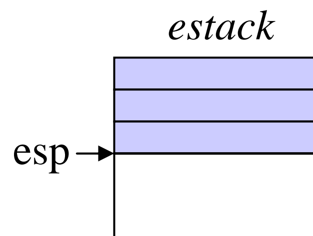
What is a virtual machine (VM)?

- a CPU implemented in Software
- Commands will be interpreted / JIT-compiled
- other examples: Java-VM, Smalltalk-VM, Pascal P-Code



The CLR is a stack machine

- no registers
- instead it has an *expression stack* (onto which values are loaded)



max. size is stored in metadata of each method

esp ... expression stack pointer

The CLR executes JIT-compiled bytecode

- each method is compiled right before the first execution (= just-in-time)
- operands are addressed symbolically in IL (informationen is stored in the metadata)

How a Stack Machine Works

Example

statement **i = i + j * 5;**

assume the following values of *i* and *j*

locals

0	3	i
1	4	j

Simulation

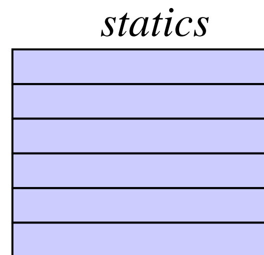
instructions	*stack*	
ldloc.0	3	load variable from address 0 (i.e. *i*)
lldloc.1	3 4	load variable from address 1 (i.e. *j*)
ldc.i4.5	3 4 5	load constant 5
mul	3 20	multiply the two topmost stack elements
add	23	add the two topmost stack elements
stloc.0		store the topmost stack element to address 0

At the end of every statement the expression stack is empty!

Data Areas of the CLR

Global variables

statics



- are mapped to static fields of the program class in the CLR
- global variables live during the whole program (= while the program class is loaded)
- global variables are addressed by medadata tokens
  e.g. ldsfld T_{fld} loads the value of the field referenced by T_{fld} onto *estack*

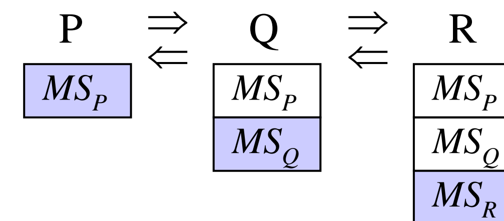
Metadata token are 4 Byte values that reference rows in metadata tables.

token type (1 Byte)	index into metadata table (3 Byte)

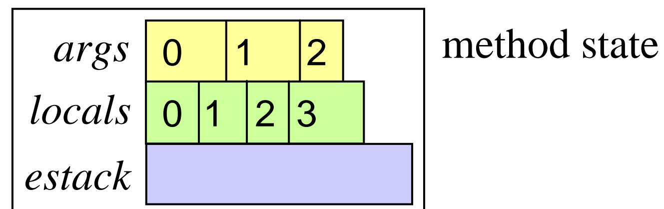
Data Areas of the CLR

Method state

- manages separate areas for
  - arguments (*args*)
  - local variables (*locals*)
  - expression stack (*estack*)
- each method call has its own method state (*MS*)
- method states managed in a stack and therefore also called *stack frames*.



- each parameter and each local variable occupy a slot of typedependent size.
- addresses are consecutive number reflecting the order of declaration
  e.g. *ldarg.0* loads the value of the first method argument onto *estack*
  ldloc.2 load the value of the third local variable onto *estack*

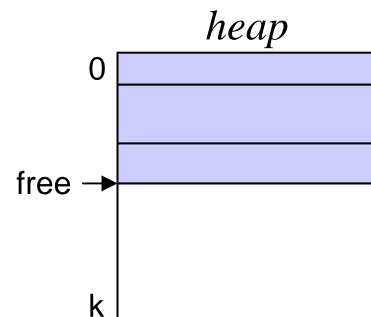


method state

Data Areas of the CLR

Heap

- contains class objects and array objects



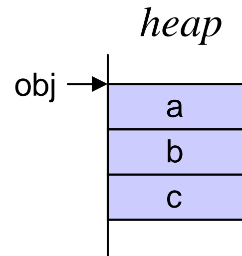
- new objects are allocated at the position *free* (and *free* is incremented); this is done by the CIL instructions *newobj* and *newarr*
- objects are deallocated by the garbage collector

Data Areas of the CLR

class objects

```
class X {
    int a, b;
    char c;
}
X obj = new X;
```

heap

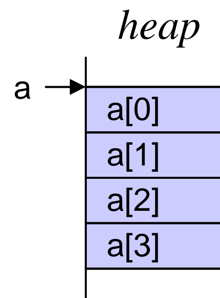


- addressed by field token relative to *obj*

array objects

```
int[] a = new int[4];
```

heap



- addressed by index value relativ to *a*

- Z# supports only vectors, i.e. one dimensional arrays with index starting at 0
- only vectors can be handled by the special CIL instructions *newarr*, *ldlen*, *ldelem*, *stelem*

Code Buffer

- Z# has only static methods that belong to type of the main program class
- each method consists of a stream of CIL instructions and metadata, like *.entrypoint*, *.locals*, *.maxstack*, access attributes, ...

System.Reflection.Emit.ILGenerator manages the CIL stream.
We can access the IL stream of the currently compiled method via the static field

```
internal static ILGenerator il;
```

of class *Code*.

```
class ILGenerator {
    /* overloaded (see next slide) */
    virtual void Emit (OpCode op, ...);
    /* for method calls */
    void EmitCall (
        OpCode op,
        MethodInfo meth,
        Type[] varArgs);
    ...
}
```

Handles for instruction are defined in calss *OpCodes*.
Wir verwenden Abkürzungen in Klasse *Code*.

```
static readonly OpCode
    LDARG0 = OpCodes.Ldarg_0, ...
    LDARG  = OpCodes.Ldarg_S, ...
    LDC0   = OpCodes.Ldc_I4_0, ...
    ADD    = OpCodes.Add, ...
    BEQ    = OpCodes.Beq, ...
    ... ;
```

e.g.: Generating *ldloc.2*

```
Code.il.Emit(Code.LDLOC2);
```

Emit methods of ILGenerator

```
class ILGenerator {                          // use for the following IL instructions:
    void Emit (OpCode);                      // ldarg.n, ldloc.n, stloc.n, ldnull, ldc.i4.n, ld.i4.m1
                                             // add, sub, mul, div, rem, neg,
                                             // ldlen, ldelem... , stelem... , dup, pop, ret, throw
    void Emit (OpCode, byte);                // ldarg.s, starg.s, ldloc.s, stloc.s
    void Emit (OpCode, int);                 // ldc.i4
    void Emit (OpCode, FieldInfo);           // ldsfld, stsfld, ldfld, stfld
    void Emit (OpCode, LocalBuilder);        // ldloc.s, stloc.s
    void Emit (OpCode, ConstructorInfo);     // newobj
    void Emit (OpCode, Type);                // newarr
    void Emit (OpCode, Label);               // br, beq, bge, bgt, ble, blt, bne.un

    /* for method calls */
    void EmitCall (OpCode, MethodInfo, Type[]);  // call (set last argument (varargs) to null)
    ...
}
```

Metadata

describes the properties of the components of an assembly (types, fields, methods, ...).
Class *Code* has fields for managing important metadata items:

```
static AssemblyBuilder assembly; // the program assembly
static ModuleBuilder module;        // the program module
static TypeBuilder program;         // the main class
static TypeBuilder inner;           // the currently compiled inner class
```

The method *CreateMetadata* of class *Code* creates metadata objects from symbol nodes:

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Prog:
         AssemblyName aName = new AssemblyName(); aName.Name = sym.name;
         assembly = AppDomain.CurrentDomain.DefineDynamicAssembly(
                              aName, AssemblyBuilderAccess.Save);
         module = assembly.DefineDynamicModule(sym.name + "Module", sym.name + ".exe");
         program = module.DefineType(sym.name, TypeAttributes.Class | TypeAttributes.Public);
         inner = null;
         break;
      ...
}
```

Metadata: local variables

The method *DeclareLocal* of *ILGenerator* returns a System.Reflection.Emit.LocalBuilder that builds the metadata for local variables.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Local:
         il.DeclareLocal(sym.type.sysType);
         break;
      ...
}
```

- additional field in Struct nodes

  ```
  internal Type sysType;
  ```

  for creating the corresponding CLR types for a Symbol node.

Metadata: types

The method *DefineType* of *ModuleBuilder* returns a System.Reflection.Emit.TypeBuilder that builds the metadata for types.
Inner classes from Z# are mapped to outer types in the CLR.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      inner = module.DefineType(sym.name, TypeAttributes.Class | TypeAttributes.NotPublic);
      sym.type.sysType = inner;

      // define default contructor (simply calls base constructor)
      sym.ctor = inner.DefineConstructor( MethodAttributes.Public, CallingConventions.Standard,
                                          new Type[0]);
      il = sym.ctor.GetILGenerator();
      il.Emit(LDARG0);
      il.Emit(CALL, typeof(object).GetConstructor(new Type[0]));
      il.Emit(RET);
      break;
      ...
}
```

no-arg constructor of System.Object

- additional field for Symbol nodes

  `internal ConstructorBuilder ctor;`

  for generating invocations to constructor when creating new objects (*newobj*).

Metadata: global variables & object fields

The CLR handles both as fields (static fields of program class; instance fields of inner class type).
The method *DefineField* of *TypeBuilder* returns a System.Reflection.Emit.FieldBuilder
that generates the metadata for fields.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Global:
         if (sym.type != Tab.noType)
            sym.fld = program.DefineField(sym.name, sym.type.sysType, FieldAttributes.Assembly
                                                                        | FieldAttributes.Static);
         break;
      case Symbol.Kinds.Field:
         if (sym.type != Tab.noType)
            sym.fld = inner.DefineField(sym.name, sym.type.sysType, FieldAttributes.Assembly);
         break;
      ...
}
```

- additional fields for Symbol nodes

  `internal FieldBuilder fld;`

  for generating field accesses (*ldsfld*, *stsfld*, *ldfld*, *stfld*).

Metadata: methods & arguments

The method *DefineMethod* of *TypeBuilder* returns a System.Reflection.Emit.MethodBuilder that generates the metadata for methods and their arguments.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Meth:
         Type[] args = new Type[sym.nArgs];
         Symbol arg;
         for (arg = sym.locals; arg != null && arg.kind == Symbol.Kinds.Arg; arg = arg.next)
            args[arg.adr] = arg.type.sysType;
         sym.meth = program.DefineMethod( sym.name,  MethodAttributes.Static
                                                     | MethodAttributes.Family,
                                          sym.type.sysType, args);
         il = sym.meth.GetILGenerator();
         if (sym.name == "Main") assembly.SetEntryPoint(sym.meth);
         break;
      ...
}
```

- additional field for Symbol nodes

  `internal MethodBuilder meth;`

  for generating method calls (*call*).

Instruction Set of the CLR

Common Intermediate Language (CIL) (here we need only a subset)

- very compact: most instructions are just 1 byte long
- mostly untyped; type indications refer to the result type

untyped	*typed*
ldloc.0	ldc.i4.3
starg.1	ldelem.i2
add	stelem.ref

Instruction format

very simple compared to Intel, PowerPC or SPARC

```
Code         = { Instruction }.
Instruction  = opcode [ operand ].
```

opcode ... 1 or 2 byte
operand ... of primitive type or metadata token

Examples

0 operands	add	has 2 implicit operands on the stack
1 operand	ldc.i4.s 9	

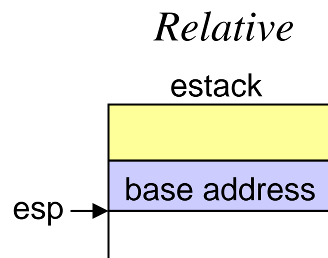
Instruction Set of the CLR

Addressing modes

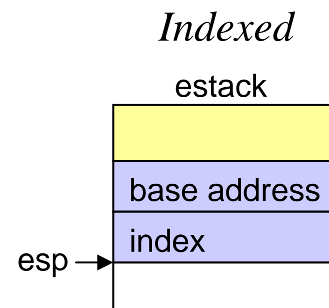
How can operands be accessed?

addressing mode	*example*	
• **Immediate**	ldc.i4 123	for constants
• **Arg**	ldarg.s 5	for method arguments
• **Local**	ldlocs.s 12	for local variable
• **Static**	ldsfld *fld*	for statische Felder (*fld* = metadata token)
• **Stack**	add	for loaded values on *estack*
• **Relative**	ldfld *fld*	for object fields (object reference is on *estack*)
• **Indexed**	ldelem.i4	for array elements (array reference & index are on *estack*)

Relative



Indexed



Instruction Set of the CLR

Loading and storing of method arguments

ldarg.s b	... ..., val	Load push(args[b]);
ldarg.*n*	... ..., val	Load (n = 0..3) push(args[n]);
starg.s b	..., val ...	Store args[b] = pop();

Operand types

b ... unsigned byte
i ... signed integer
T ... metadata token

Loading and storing of local variables

ldloc.s b	... ..., val	Load push(locals[b]);
ldloc.*n*	... ..., val	Load (n = 0..3) push(locals[n]);
stloc.s b	..., val ...	Store locals[b] = pop();
stloc.*n*	..., val ...	Store (n = 0..3) locals[n] = pop();

Instruction Set of the CLR

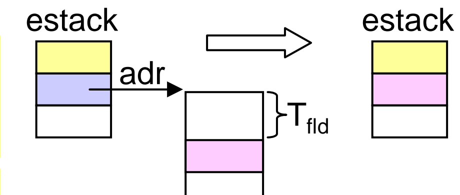
Loading and storing of global variables

ldsfld	T_{fld}	... ..., val	Load static variable push(statics[T_{fld}]);
stsfld	T_{fld}	..., val ...	Store static variable statics[T_{fld}] = pop();

Loading and storing of object field

ldfld	T_{fld}	..., obj ..., val	Load object field obj = pop(); push(heap[obj+T_{fld}]);

stfld	T_{fld}	..., obj, val
		...
			Store object field
			val = pop(); obj = pop();
			heap[obj+T_{fld}] = val;

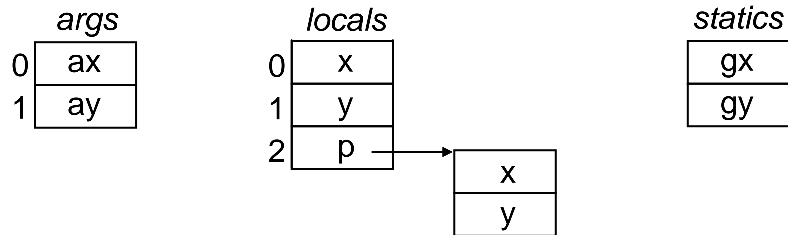


Instruction Set of the CLR

Loading of constants

ldc.i4 i	... ..., i	Load constant push(i);
ldc.i4.*n*	... ..., *n*	Load constant (n = 0..8) push(n);
ldc.i4.m1	... ..., -1	Load minus one push(-1);
ldnull	... ..., null	Load null push(null);

Example: loading and storing



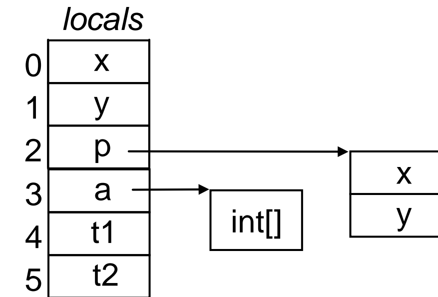
	Code	*Bytes*	*estack*
ax = ay;	ldarg.1	1	ay
	starg.s 0	2	-
x = y;	ldloc.1	1	y
	stloc.0	1	-
gx = gy;	ldsfld T_{fld_gy}	5	gy
	stsfld T_{fld_gx}	5	-
p.x = p.y;	ldloc.2	1	p
	ldloc.2	1	p p
	ldfld T_{fld_y}	5	p p.y
	stfld T_{fld_x}	5	-

Instruction Set of the CLR

Arithmetics

add	..., val1, val2 ..., val1+val2	Add push(pop() + pop());
sub	..., val1, val2 ..., val1-val2	Subtract push(-pop() + pop());
mul	..., val1, val2 ..., val1*val2	Multiply push(pop() * pop());
div	..., val1, val2 ..., val1/val2	Divide x = pop(); push(pop() / x);
rem	..., val1, val2 ..., val1%val2	Remainder x = pop(); push(pop() % x);
neg	..., val ..., -val	Negate push(-pop());

Examples: arithmetics



	Code	*Bytes*	*Stack*
x + y * 3	ldloc.0	1	x
	ldloc.1	1	x y
	ldc.i4.3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3
x++;	ldloc.0	1	x
	ldc.i4.1	1	x 1
	add	1	x+1
	stloc.0	1	-
p.x++	ldloc.2	1	p
	dup	1	p p
	ldfld T_{px}	5	p p.x
	ldc.i4.1	1	p p.x 1
	add	1	p p.x+1
	stfld T_{px}	5	-

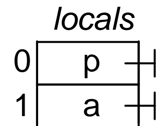
	Code	Bytes	Stack
x--;	ldloc.0	1	x
	ldc.i4.m1	1	x -1
	add	1	x-1
	stloc.0	1	-
a[2]++	ldloc.3	1	a
	ldc.i4.2	1	a 2
	stloc.s 5	2	a
	stloc.s 4	2	-
	ldloc.s 4	2	a
	ldloc.s 5	2	a 2
	ldloc.s 4	2	a 2 a
	ldloc.s 5	2	a 2 a 2
	ldelem.i4	1	a 2 a[2]
	ldc.i4.1	1	a 2 a[2] 1
	add	1	a 2 a[2]+1
	stelem.i4	1	-

Instruction Set of the CLR

Object creation

newobj T_{ctor}	... [arg0, ..., argN] ..., obj	New object creates a new object of the type specified by the constructor token and then executes the constructor (arguments are on the stack)
newarr T_{eType}	..., n ..., arr	New array creates an array with space for n Elements of the type specified by the type token

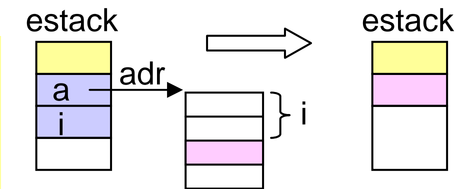
Examples: object creation



	Code	*Bytes*	*Stack*
Person p = new Person;	newobj $T_{P()}$	5	p
	stloc.0	1	-
int[] a = new int[5];	ldc.i4.5	1	5
	newarr T_{int}	5	a
	stloc.1	1	-

Instruction Set of the CLR

Array access

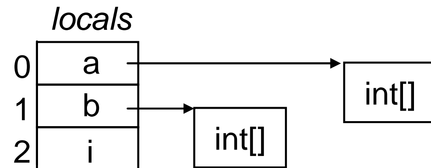


ldelem.u2
ldelem.i4
ldelem.ref
	..., adr, i
	..., val
		<u>Load array element</u>
		i = pop(); adr = pop();
		push(heap[adr+i]);
		result type on *estack*:
		char, int, object reference

stelem.i2
stelem.i4
stelem.ref
	...,adr, i, val
	...
		<u>Store array element</u>
		val=pop(); i=pop(); adr=pop()/4+1;
		heap[adr+i] = val;
		type of element to be stored:
		char, int, object reference

ldlen	..., adr ..., len	<u>Get array length</u>

Example: array access



a[i] = b[i+1];

Code	*Bytes*	*Stack*
ldloc.0	1	a
ldloc.2	1	a i
ldloc.1	1	a i b
ldloc.2	1	a i b i
ldc.i4.1	1	a i b i 1
add	1	a i b i+1
ldelem.i4	1	a i b[i+1]
stelem.i4	1	-

Instruction Set of the CLR

Stack manipulation

pop	..., val ...	Remove topmost stack element dummy = pop();
dup	..., val ..., val, val	Duplicate topmost stack element x = pop(); push(x); push(x);

Jumps

br i	... ...	Branch unconditionally pc = pc + i

b<*cond*> i	..., x, y
	...
		Branch conditionally
		(<*cond*> = eq | ge | gt | le | lt | ne.un)
		y = pop(); x = pop();
		if (x cond y) pc = pc + i;

pc marks the current instruction;
i (jump distance) relative to the start of the next instruction

Example: jumps

locals

0	x
1	y

	Code	*Bytes*	*Stack*
if (x > y) {	ldloc.0	1	x
...	ldloc.1	1	x y
}	ble ...	5	-
...			

Instruction Set of the CLR

Method call

call T_{meth}	... [arg0, ... argN] ... [retVal]	Call method pops the arguments off caller *estack* and puts them into *args* of the callee; pops return value off callee *estack* and pushes it onto caller *estack*
ret	... ...	Return from method

Sonstiges

throw	..., exc ...	Throw exception

Instruction Set of the CLR

I/O

The functionality of the Z# keywords *read* and *write* is provided by separate methods of the main program class. These use the class *System.Console* for In-/Output.

```
static char read ();
static int readi ();
static void write (char, int);
static void write (int, int);
```

These methods are added to the main program class by the compiler.

```
internal static void CreateMetadata (Symbol sym) { ...
   switch (sym.kind) { ...
      case Symbol.Kinds.Program: ...
         BuildReadChar(); BuildReadInt(); BuildWriteChar(); BuildWriteInt(); break;
   }
}
```

and are then accessible via static fields of class *Code*:

```
internal static MethodBuilder readChar, readInt, writeChar, writeInt;
```

In the parser we can generate a call to these methods like this:

```
Code.il.EmitCall(Code.CALL, Code.readChar, null);
```

Example

C# source	CIL code
void Main () int a, b, max, sum; {	static void Main ()
if (a > b)	0: ldloc.0 1: ldloc.1 2: ble 7 (=14)
max = a;	7: ldloc.0 8: stloc.2 9: br 2 (=16)
else max = b;	14: ldloc.1 15: stloc.2
while (a > 0) {	16: ldloc.0 17: ldc.i4.0 18: ble 15 (=38)
sum = sum + a * b;	23: ldloc.3 24: ldloc.0 25: ldloc.1 26: mul 27: add 28: stloc.3
a--;	29: ldloc.0 30: ldc.i4.1 31: sub 32: stloc.0
}	33: br -22 (=16)
}	38: return

addresses

a ... 0
b ... 1
max ... 2
sum ... 3

6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

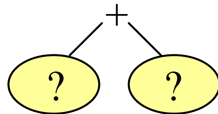
6.6 Control Structures

6.6 Methods

Operands During Code Generation

Example

we want to add two values



desired code pattern

load operand 1
load operand 2
add

Depending on the operand kind we must generate different load instructions

operand kind	*instruction to be generated*
• constant	ldc.i4 x
• method argument	ldarg.s a
• local variable	ldloc.s a
• global variable	ldsfld T_{fld}
• object field	getfield a
• array element	ldelem
• loaded value on the stack	---

We need a descriptor, which gives us all the necessary information about operands.

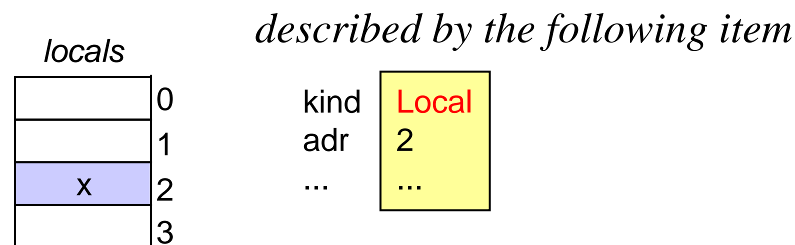
Items

Descriptors holding the kind and the location of operands.

Example

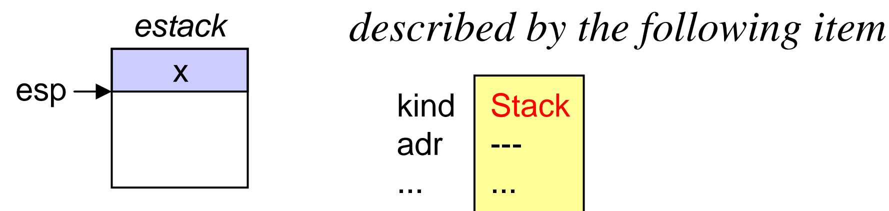
Local variable *x* in locals area of a method state

described by the following item



After loading the value with *ldloc.2* it is on *estack*

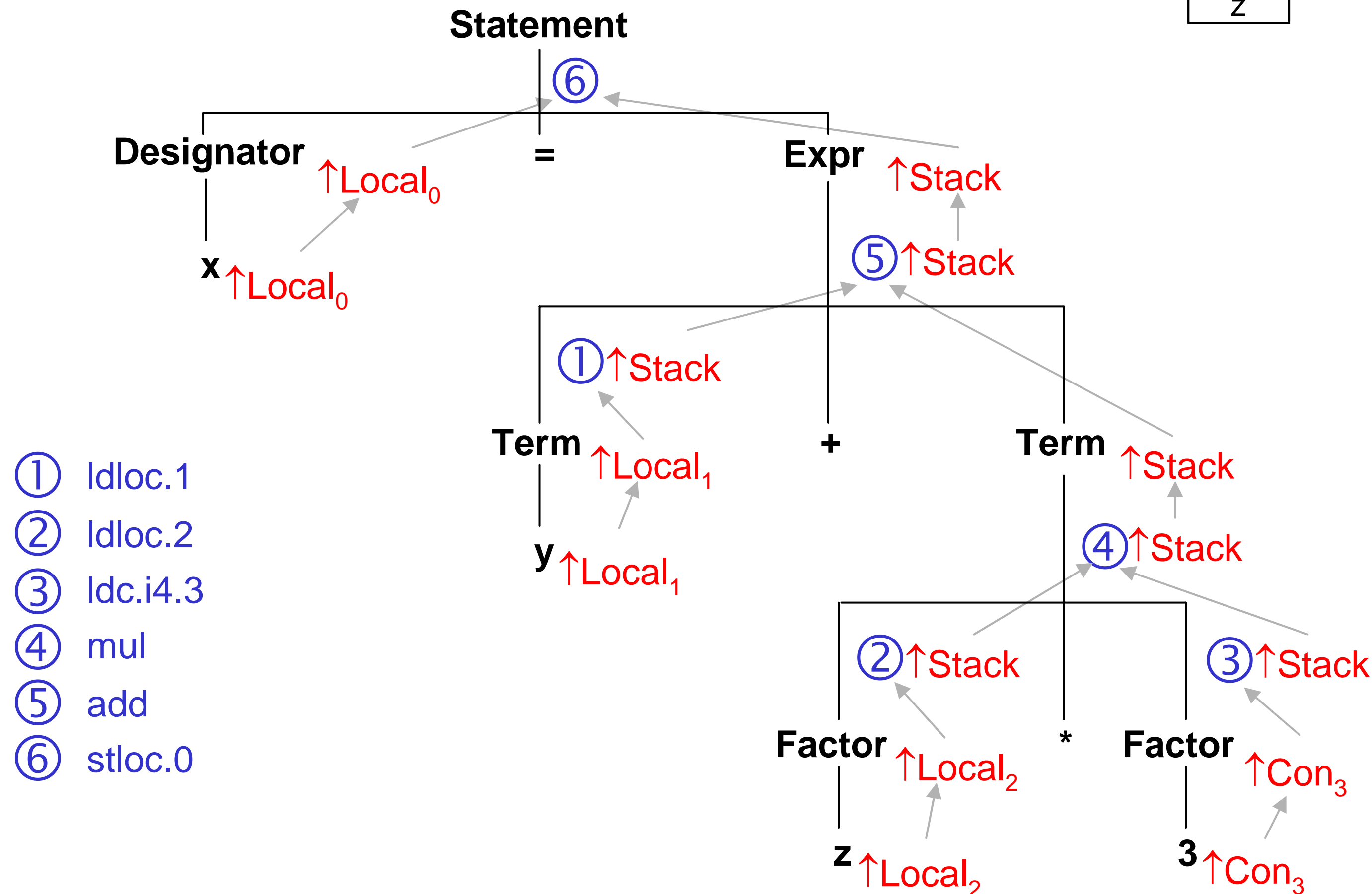
described by the following item



Example: Processing of Items

Most parsing methods return items (as a result of their translation work)

Example: translating the assignment x = y + z * 3;



Item Kinds

operand kinds	*item kind*	*info about operands*	
constant	**Const**	constant value	
argument	**Arg**	address	*args* adr
local variable	**Local**	address	*locals* adr
global variable	**Static**	field symol node	*statics* T_{fld}
object field	**Field**	field symbol node	*estack* esp adr T_{fld}
value on stack	**Stack**	---	*estack* esp
array element	**Elem**	---	*estack* esp adr idx idx
method	**Meth**	method symbol node	

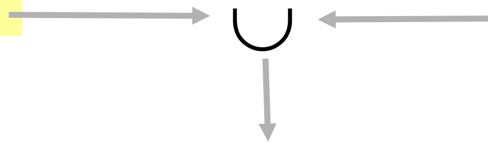
How Do we Find the Necessary Item Kinds?

addressing modes
depending on the target machine

- Immediate
- Arg
- Local
- Static
- Stack
- Relative
- Indexed

object kinds
depending on the source language

- Con
- Var
- Type
- Meth



item kinds

- Con
- Arg
- Local
- Static
- Stack
- Fld
- Elem
- Meth

We do not need *Type* items in Z#,
because types do not occur as operands
(no type casts)

Class Item

```
class Item {
    enum Kinds { Const, Arg, Local, Static, Stack, Field, Elem, Meth }

    Kinds   kind;
    Struct  type;   // type of the operands
    int     val;    // Const: constant value
    int     adr;    // Arg, Local: address
    Symbol  sym;    // Field, Meth: Symbol node from symbol table
}
```

Constructors for creating Items

```
public Item (Symbol sym) {
    type = sym.type; val = sym.val; adr = sym.adr; this.sym = sym;
    switch (sym.kind) {
        case Symbol.Kinds.Const: kind = Kinds.Const; break;
        case Symbol.Kinds.Arg: kind = Kinds.Arg; break;
        case Symbol.Kinds.Local: kind = Kinds.Local; break;
        case Symbol.Kinds.Global: kind = Kinds.Static; break;
        case Symbol.Kinds.Field: kind = Kinds.Field; break;
        case Symbol.Kinds.Meth: kind = Kinds.Meth; break;
        default: Parser.Error("cannot create Item");
    }
}
```

create an Item from a symbol table node

```
public Item (int val) {
    kind = Kinds.Const; type = Tab.intType; this.val = val;
}
```

creates an Item from a constant

Loading Values (1)

given: a value described by an item (Const, Arg, Local, Static, ...)
wanted: load the value onto the expression stack

```
public static void Load (Item x) { // method of class Code
    switch (x.kind) {
        case Item.Kinds.Const:
            if (x.type == Tab.nullType) il.Emit(LDNULL);
            else LoadConst(x.val);
            break;
        case Item.Kinds.Arg:
            switch (x.adr) {
                case 0: il.Emit(LDARG0); break;
                ...
                default: il.Emit(LDARG, x.adr); break;
            }
            break;
        case Item.Kinds.Local:
            switch (x.adr) {
                case 0: il.Emit(LDLOC0); break;
                ...
                default: il.Emit(LDLOC, x.adr); break;
            }
            break;
        ...
```

Case analysis

depending on the item kind we have to generate different load instructions

Loading Values (2)

```
public static void Load (Item x) {  // cont.
    ...
    case Item.Static:
      if (x.sym.fld != null) il.Emit(LDSFLD, x.sym.fld); break;
    case Item.Stack: break; // nothing to do (already loaded)
    case Item.Field:  // assert: object base address is on stack
      if (x.sym.fld != null) il.Emit(LDFLD, x.sym.fld); break;
    case Item.Elem: // assert: array base address and index are on stack
      if (x.type == Tab.charType) il.Emit(LDELEMCHR);
      else if (x.type == Tab.intType) il.Emit(LDELEMINT);
      else if (x.type.kind == Struct.Kinds.Class) il.Emit(LDELEMREF);
      else Parser.Error("invalid array element type");
      break;
    default: Error("cannot load this value");
  }
  x.kind = Item.Kinds.Stack;
}
```

resulting item is always a *Stack* item

```
internal static void LoadConst (int n) {  // method of class Code
  switch (n) {
    case -1: il.Emit(LDCM1); break;
    case 0: il.Emit(LDC0); break;
    ...
    default: il.Emit(LDC, n); break;
  }
}
```

Example: Loading a Constant

Description by an ATG

```
Factor <↑Item x>
=  number               (.  x = new Item(token.val);  // x.kind = Const
                            Code.Load(x);              // x.kind = Stack
                         .)
```

Visualisation

val　　x = new Item(token.val);　　Code.Load(x);

output

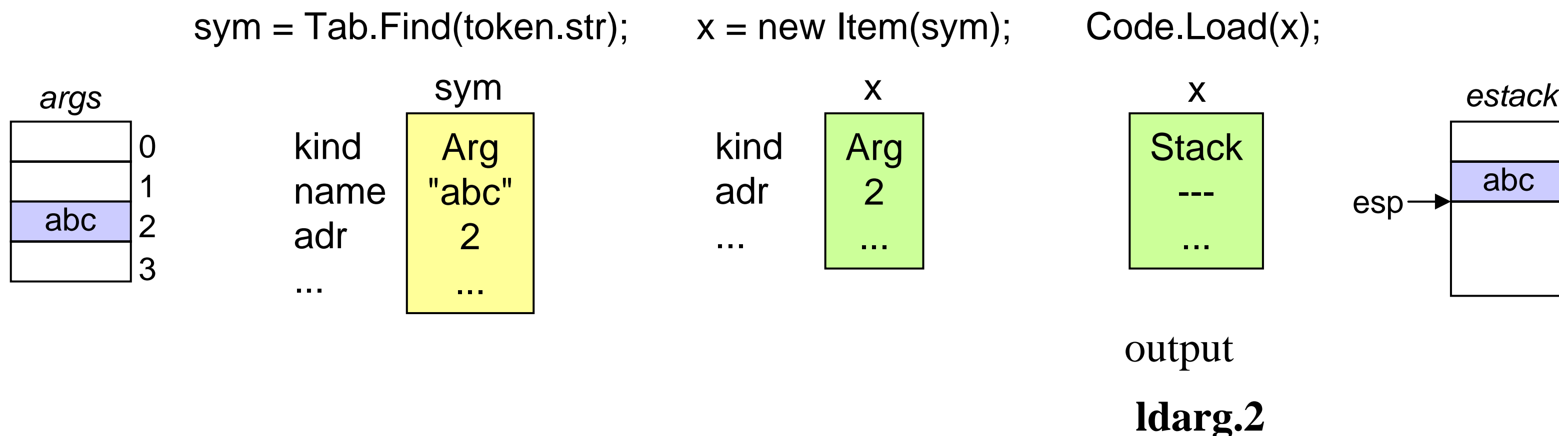
ldc.i4 17

Example: Loading an Argument

Description by an ATG

```
Designator<↑Item x>
=  ident                (. Symbol sym = Tab.Find(token.str); // sym.kind = Const | Arg | Local | Global
                           x = new Item(sym);                // x.kind = Const | Arg | Local | Static
                           Code.Load(x);                     // x.kind = Stack
                        .) .
```

Visualisation

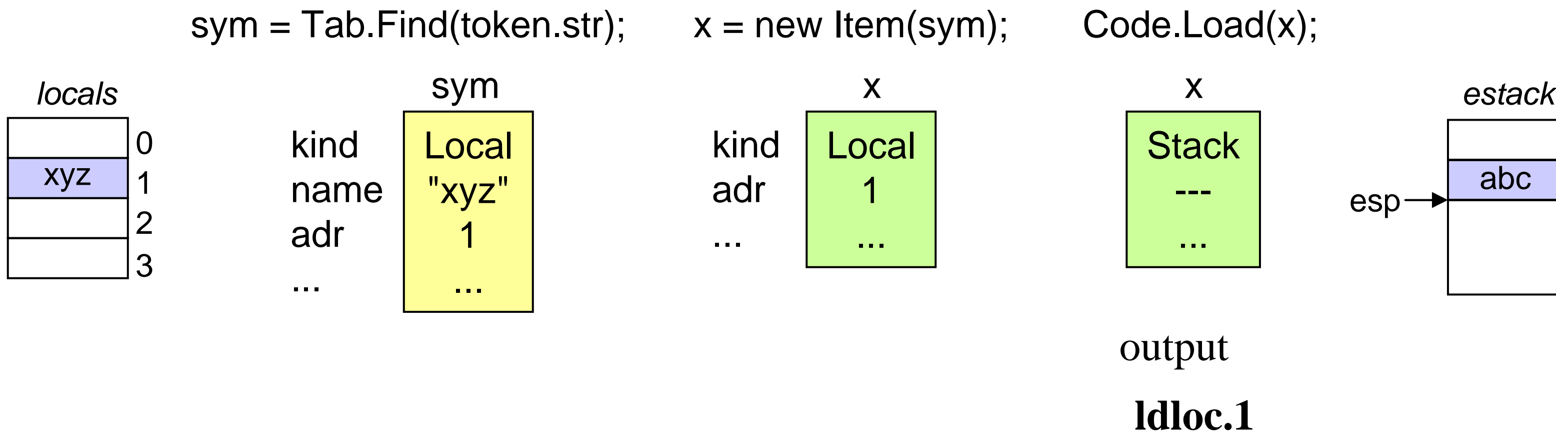


Example: Loading a Local Variable

Description by an ATG

```
Designator<↑Item x>
=  ident           (. Symbol sym = Tab.Find(token.str); // sym.kind = Const | Arg | Local | Global
                      Item x = new Item(sym);            // x.kind = Const | Arg | Local | Static
                      Code.Load(x);                      // x.kind = Stack
                    .) .
```

Visualisation



Loading Object Fields

var.f

Context conditions (make sure that your compiler checks them)

Designator = Designator "." ident .

- The type of *Designator* must be a class.
- *ident* must be a field of *Designator*.

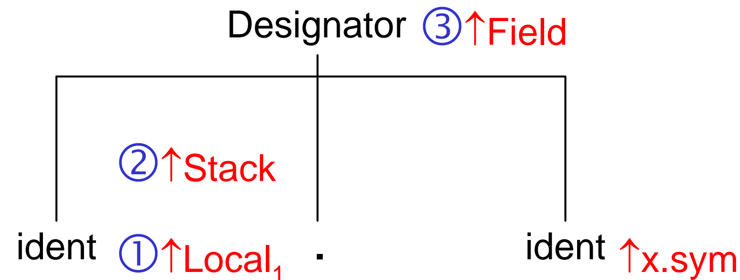
Description by an ATG

```
Designator<↑Item x> (. string name; .)
= ident              (. name = token.str;
                        Item x = new Item(Tab.Find(name)); .)
  { "." ident        (. if (x.type.kind == Struct.Kinds.Class) {
                            Code.Load(x);
                            x.sym = Tab.FindField(token.str, x.type);
                            x.type = x.sym.type;
                        } else Error(name + " is not an object");
                        x.kind = Item.Kinds.Field;
                      .)
  | ...
  }.
```

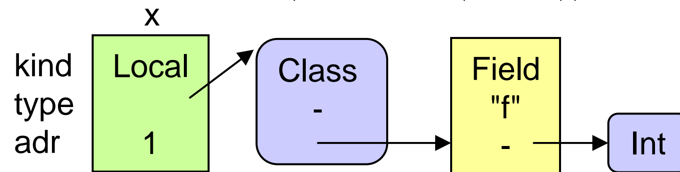
looks up *token.str* in the field list of *x.type*

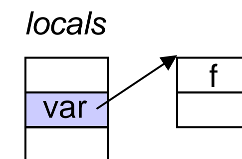
creates a *Field* item

Example: Loading an Object Field

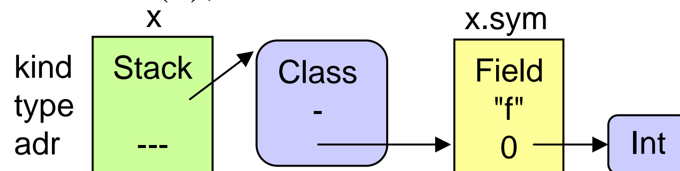


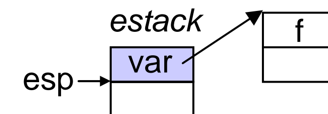
① Item x = new Item(Tab.Find(name));





② Code.Load(x);

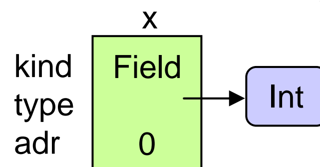


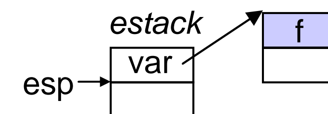


output

ldloc.1

③ create from *x* and *x.sym* a *Field* item





Loading Array Elements

a[i]

Context conditions

```
Designator = Designator "[" Expr "]" .
```

- The type of *Designator* must be an array.
- The type of *Expr* must be *int*.

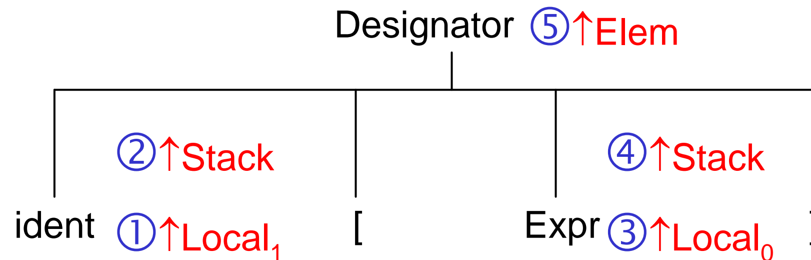
Description by an ATG

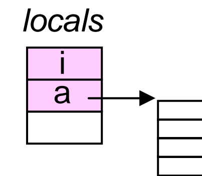
```
Designator<↑Item x>   (. string name; Item x, y; .)
= ident               (. name = token.str; x = new Item(Tab.find(name)); .)
  [
    "["               (. Code.Load(x); .)
    Expr<↑y>          (. if (x.type.kind == Struct.Arr) {
                             if (y.type != Tab.intType) Error("index must be of type int");
                             Code.Load(y);
                             x.type = x.type.elemType;
                         } else Error(name + " is not an array");
                         x.kind = Item.Elem;
                       .)
    "]"
  ] { … }.
```

create an *Elem* item

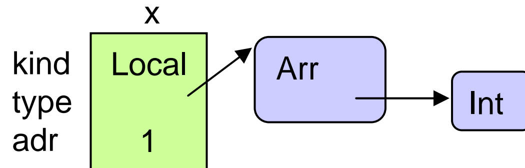
index check is done in the CLR

Example: Loading an Array Element

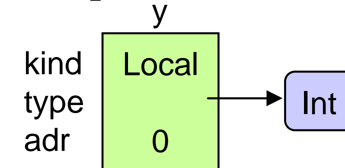




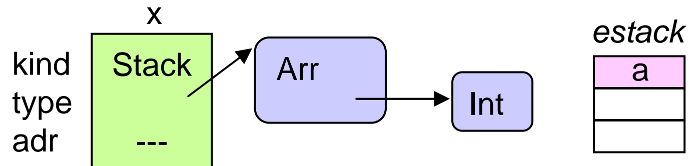
① Item x = new Item(Tab.Find(name));



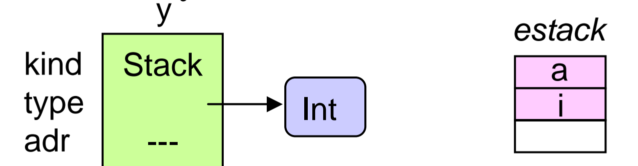
③ y = Expr();



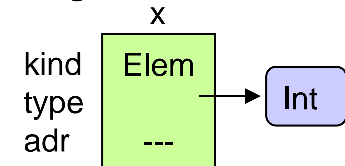
② Code.Load(x); **ldloc.1**



④ Code.Load(y); **ldloc.0**



⑤ erzeuge aus *x* ein *Elem*-Item



6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

6.7 Control Structures

6.8 Methods

Compiling Expressions

Scheme for x + y + z

load x
load y
add
load z
add

Context conditions

Expr = "-" Term.
• *Term* must be of type *int*.

Expr = Expr Addop Term.
• *Expr* and *Term* must be of type *int*.

Description by an ATG

```
Expr<↑Item x>          (. Item y; OpCode op; .)
= ( "-" Term<↑x>       (. if (x.type != Tab.intType) Error("operand must be of type int");
                          if (x.kind == Item.Kinds.Const) x.val = -x.val;
                          else { Code.Load(x); Code.il.Emit(Code.NEG); }
                        .)
  | Term<↑x>
  )
  { ( "+"              (. op = Code.ADD; .)
    | "-"              (. op = Code.SUB; .)
    )                  (. Code.Load(x); .)
    Term<↑y>           (. Code.Load(y);
                          if (x.type != Tab.intType || y.type != Tab.intType)
                            Error("operands must be of type int");
                          Code.il.Emit(op);
                        .)
  }.
```

Compiling Terms

Term = Term Mulop Factor.
• *Term* and *Factor* must be of type *int*.

```
Term<↑Item x>               (.  Item y; OpCode op; .)
= Factor<↑x>
    { ( "*"                 (.  op = Code.mul; .)
      | "/"                 (.  op = Code.div; .)
      | "%"                 (.  op = Code.rem; .)
      )                     (.  Code.Load(x); .)
      Factor<↑y>            (.  Code.Load(y);
                                if (x.type != Tab.intType || y.type != Tab.intType)
                                   Error("operands must be of type int");
                                Code.il.Emit(op);
                             .)
    }.
```

Compiling Factors

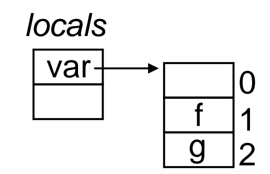
Factor = "new" ident.
• *ident* must denote a class.

Factor = "new" ident "[" Expr "]".
• *ident* must denote a type. • The type of *Expr* must be *int*.

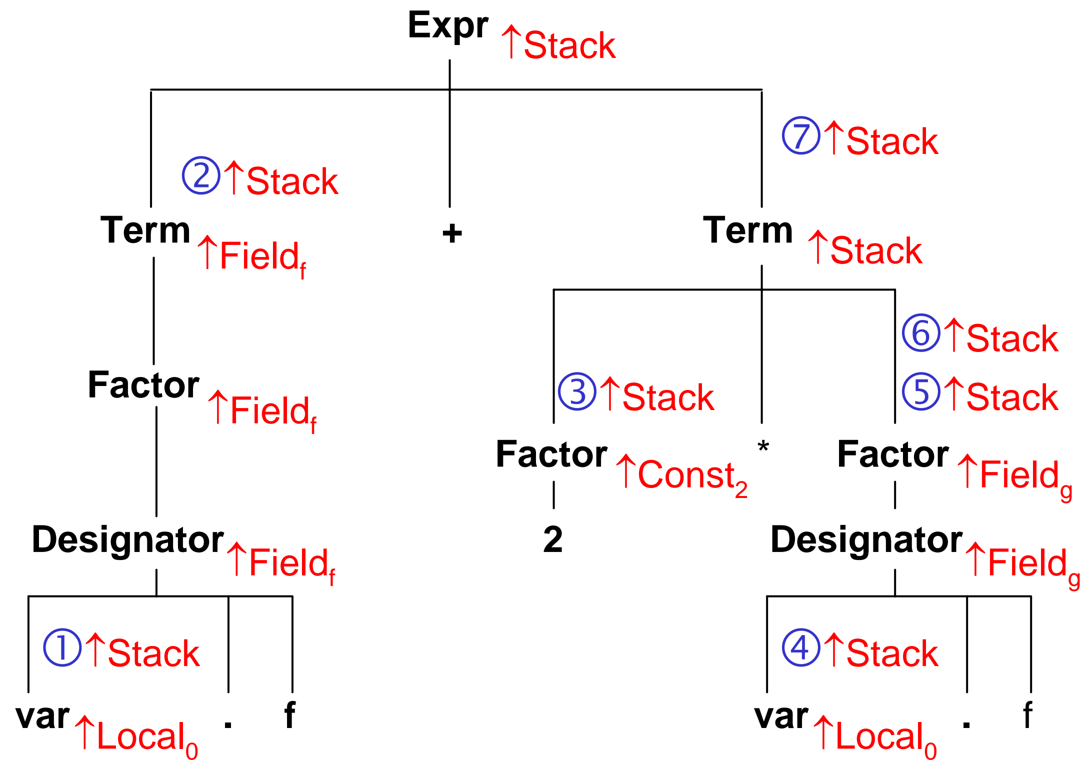
```
Factor<↑Item x>
= Designator<↑x>         // functions calls see later
| number                 (. x = new Item(token.val); .)
| charConst              (. x = new Item(token.val); x.type = Tab.charType; .)
| "(" Expr<↑x> ")"

| "new" ident            (. Symbol sym = Tab.find(token.str);
                            if (sym.kind != Symbol.Kinds.Type) Error("type expected");
                            Struct type = sym.type;
                          .)
  ( "[" Expr<↑x> "]"     (. if (x.type != Tab.intType) Error("array size must be of type int");
                            Code.Load(x);
                            Code.il.Emit(Code.NEWARR, type.sysType);
                            type = new Struct(Struct.Kinds.Arr, type);
                          .)
  |                      (. if (type.kind == Struct.Kinds.Class)
                               Code.il.Emit(Code.NEWOBJ, sym.ctor);
                            else { Error("class type expected"); type = Tab.noType; }
                          .)
  )                      (. x = new Item(Item.Kinds.Stack); x.type = type; .)
.
```

Example

var.f + 2 * var.g



1. ldloc.0
2. ldfld T_{fld_f}
3. ldc.i4.2
4. ldloc.0
5. ldfld T_{fld_g}
6. mul
7. add



6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

6.7 Control Structures

6.8 Methods

Code Patterns for Assignments

5 cases depending on the kind of the designator on the left-hand side

blue instructions are generated by *Designator*!

arg = expr;	localVar = expr;	globalVar = expr;	obj.f = expr;	a[i] = expr;
... *load expr* ... **starg arg**	... *load expr* ... **stloc localVar**	... *load expr* ... **stsfld $T_{globalVar}$**	**ldloc obj** ... *load expr* ... **stfld T_f**	**ldloc a** **ldloc i** ... *load expr* ... **stelem.i2** **i4** **ref**

depending on the element type (char, int, object reference)

Compiling Assignments

Context condition

Statement = Designator "=" Expr ";".

- *Designator* must denote a variable, an array element or an object field.
- The type of *Expr* must be assignment compatible with the type of *Designator*.

Description by an ATG

```
Assignment              (.  Item x, y; .)
=  Designator<↑x>       // this call may already generate code
   "=" Expr<↑y>         (.  Code.Load(y);
                            if (y.type.AssignableTo(x.type))
                               Code.Assign(x, y);  // x: Arg | Local | Static | Field | Elem
                            else Error("incompatible types in assignment");
                         .)
   ";".
```

Assignment compatibility

y is assignment compatible with *x*, if

- *x* and *y* have the same types (*x.type* == *y.type*), or
- *x* and *y* are arrays with the same element types, or
- *x* has a reference type (class or array) and *y* is *null*

6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

6.7 Control Structures

6.8 Methods

Conditional and Unconditional Jumps

Unconditional jumps

```
br offset
```

Conditional jumps

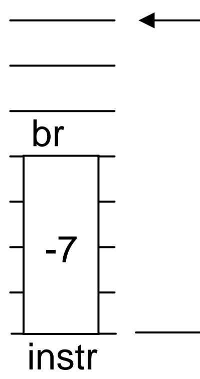
```
... load operand1 ...
... load operand2 ...
beq offset
```

if (operand1 == operand2) br offset

beq	jump on equal
bge	jump on greater or equal
bgt	jump on greater than
ble	jump on less or equal
blt	jump on less than
bne.un	jump on not equal

Forward and Backward Jumps

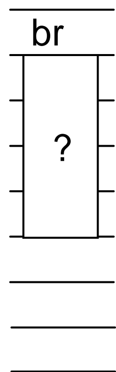
Backward jumps



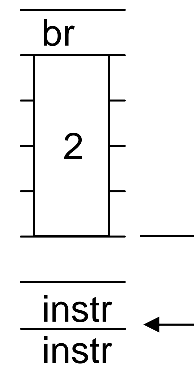
target address (label) is already known
(because the instruction at this position has already been generated)

jump distance
- 4 bytes long (CIL also has short form with 1 byte addresses
- relative to the beginning of the next instruction (= end of jump instruction)

Forward jumps



target address still unknown
⇒ leave it empty
⇒ remember "fixup address"



patch it when the target address becomes known (fixup)

Struct System.Reflection.Emit.Label

Representation of Jump Labels

```
struct Label { ... }
```

Labels are managed by *ILGenerator*.

```
class ILGenerator {
    ...
    Label DefineLabel ();          // create a yet undefined jump label
    void MarkLabel (Label);        // define the label at the current position in the IL stream
}
```

Use

```
Label label = Code.il.DefineLabel();
...
Code.il.Emit(Code.BR, label);    // jump to the yet undefined label (forward jump)
..
Code.il.MarkLabel(label);        // now the branch to label leads here
```

Conditions

Conditions

code pattern

```
if ( a > b ) ...
```

Condition (the part `a > b`)

```
load a
load b
ble ...
```

- *Condition* returns the compare operator;
  the comparison is then done in the jump instruction

Cond item

Condition returns an item of a new kind *Item.Kinds.Cond* with the following contents:

- token code of compare operator:
  ```
  public int relop;  // Token.EG, .GE, .GT, .LE, .LT, .NE
  ```
- jump labels for jumps that lead out of Condition
  - `public Label tLabel` : for true-jumps
  - `public Label fLabel` : for false-jumps

  necessary for conditions, which contain&& and || operators

True-jumps and false-jumps

true-jump	jump if the condition is true	a > b ⇒ bgt ...
false-jump	jump if the condition is false	a > b ⇒ ble ...

Cond Items

```
class Item {
   public enum Kinds { Const, Arg, Local, Static, Field, Stack, Elem, Meth, Cond }

   Kinds    kind;
   Struct   type;      // type of the operand
   int      val;       // Const: constant value
   int      adr;       // Arg, Local: address
   int      relop;     // Cond: token code of the operator
   Label    tLabel;    // jump label for true jumps
   Label    fLabel;    // jump label for false jumps
   Symbol   sym;       // Field, Meth: symbol table node
}
```

New constructor (for *Cond* items)

```
public Item (int relop, Struct type) {
   this.kind = Kinds.Cond;
   this.type = type;
   this.relop = relop;
   tLabel = Code.il.DefineLabel();
   fLabel = Code.il.DefineLabel();
}
```

Generating Conditional Jumps

With methods of class *Code*

```
class Code {
   static readonly OpCode[] brTrue = { BEQ, BGE, BGT, BLE, BLT, BNE };
   static readonly OpCode[] brFalse = { BNE, BLT, BLE, BGT, BGE, BEQ };
   ...
   internal static void TJump (Item x) {
      il.Emit(brTrue[x.relop - Token.EQ], x.tLabel);
   }

   internal static void FJump (Item x) {
      il.Emit(brFalse[x.relop - Token.EQ], x.fLabel);
   }
   ...
}
```

Use

```
"if" "(" Condition<↑x> ")"          (. Code.FJump(x); .)
```

Generating Unconditional Jumps

With a method of class *Code*

```
class Code {
   ...
   internal static void Jump (Label lab) {
      il.Emit(BR, lab);
   }
   ...
}
```

Use

```
Label label = Code.il.DefineLabel();
...
Code.Jump(label);
```

6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

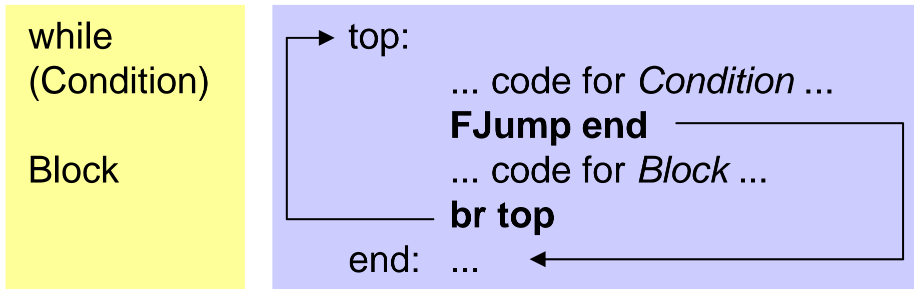
6.6 Jumps and Labels

6.7 Control Structures

6.8 Methods

while Statement

Desired code pattern



Description by an ATG

```
WhileStatement              (. Item x; .)
= "while"                   (. Label top = Code.il.DefineLabel();
                               top.here();
                             .)
  "(" Condition<↑x> ")"     (. Code.FJump(x); .)
  Block                     (. Code.Jump(top);
                               Code.il.MarkLabel(x.fLabel);
                             .)
  .
```

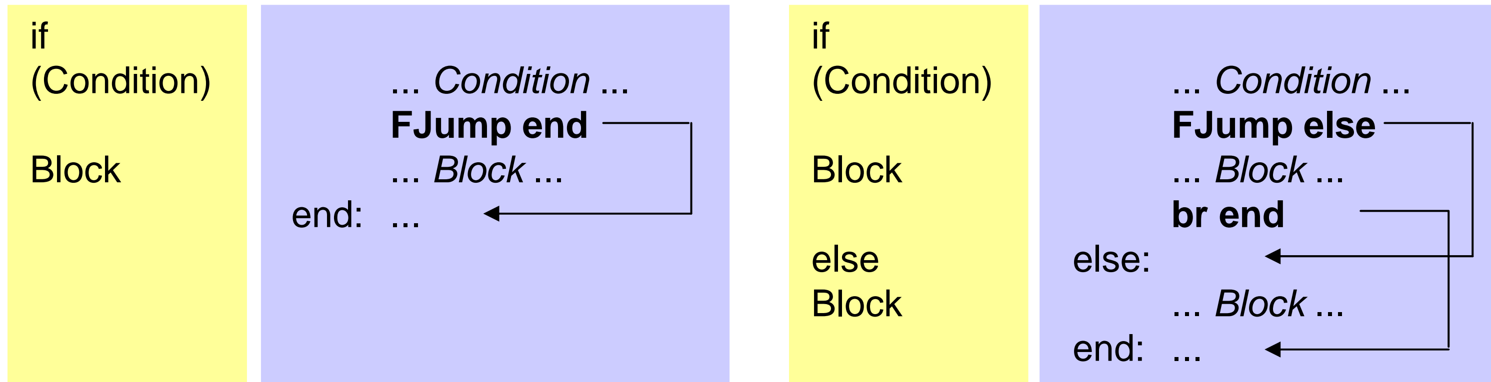
Example

while (a > b) a = a - 2;

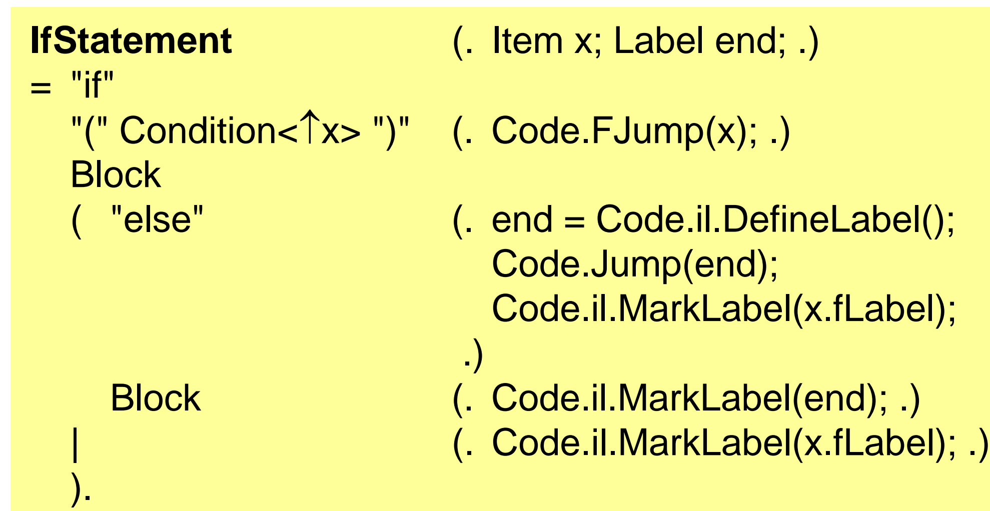
```
10   ldloc.0        <---- top
11   ldloc.1
12   ble 9 (=26)
17   ldloc.0
18   ldc.i4.2
19   sub
20   stloc.0
21   br -16 (=10)
26   ...            <---- x.fLabel
```

if Statement

Desired code pattern

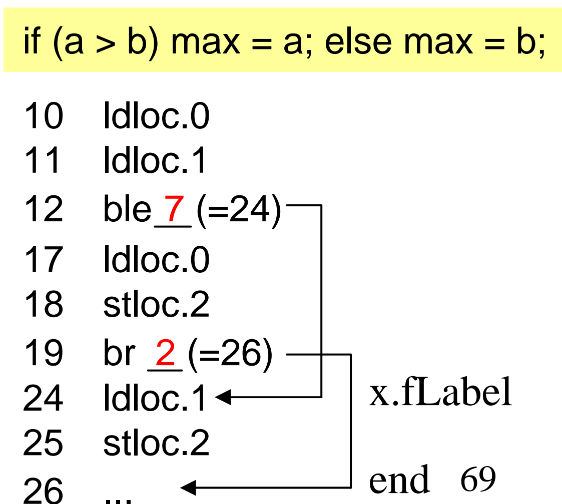


Description by an ATG



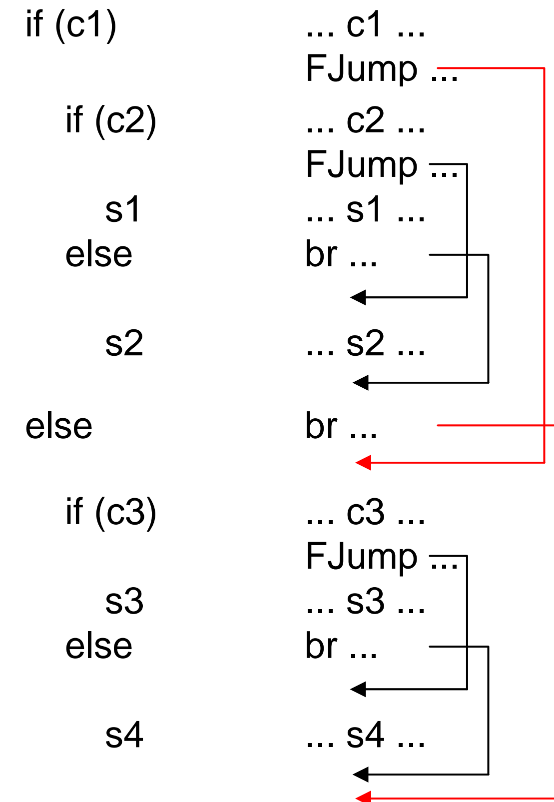
```
IfStatement                (. Item x; Label end; .)
= "if"
  "(" Condition<↑x> ")"   (. Code.FJump(x); .)
  Block
  (  "else"                (. end = Code.il.DefineLabel();
                              Code.Jump(end);
                              Code.il.MarkLabel(x.fLabel);
                            .)
     Block                 (. Code.il.MarkLabel(end); .)
  |                        (. Code.il.MarkLabel(x.fLabel); .)
  ).
```

Example



Works Also for Nested ifs

```
IfStatement                 (. Item x; Label end; .)
= "if"
  "(" Condition<↑x> ")"     (. Code.FJump(x); .)
  Block
  (  "else"                 (. end = Code.il.DefineLabel();
                               Code.Jump(end);
                               Code.il.MarkLabel(x.fLabel);
                             .)
     Block                  (. Code.il.MarkLabel(end); .)
  |                         (. Code.il.MarkLabel(x.fLabel); .)
  ).
```



break Statement

For jumping out of a loop

- define a label *breakLab* at the end of the loop
- if a break occurs in the loop: Code.Jump(breakLab);

Nested loops

- every loop needs its own *newBreakLab*
- *newBreakLab* must be passed to nested statements as an attribute

```
Statement<↓Label breakLab>
=  "while"                    (.  Label newBreakLab = Code.il.DefineLabel(); ... .)
   "(" Condition<↑x> ")"      (. ... .)
   Block<↓newBreakLab>        (. Code.il.MarkLabel(newBreakLab); .)

|  "{" { Block<↓breakLab> } "}"

|  "if"
   ...
    Block<↓breakLab>
   ...

|  "break"                    (.  if (breakLab.Equals(undef)) Error("break outside a loop");
                                  Code.Jump(breakLab);
                               .)

|  ... .
```

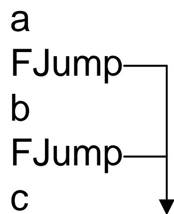
static readonly Label **undef**;
because structs cannot be null (yet)

Question: What would we have to do in order to use break with a label name?

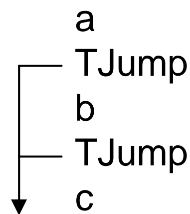
Short-Circuit Evaluation of Boolean Expressions

- Boolean expression may contain && and || operators
- The evaluation of a Boolean expression stops as soon as its result is known

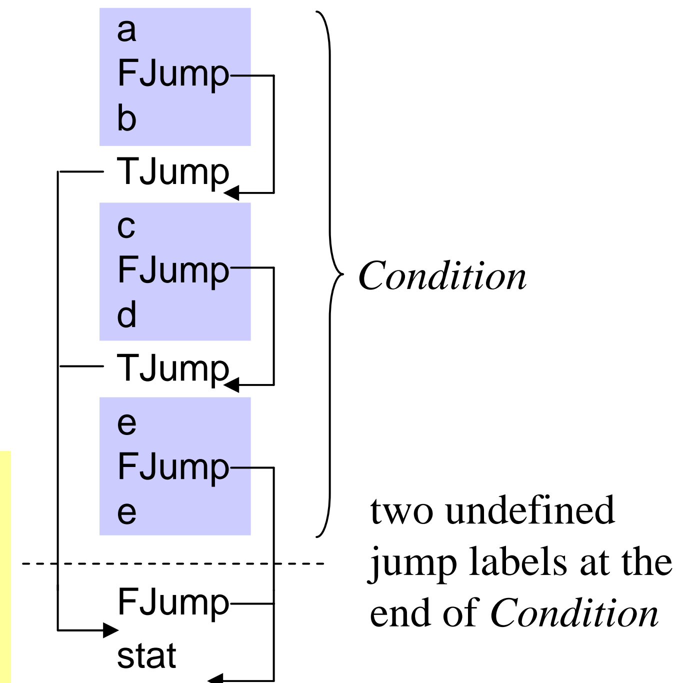
a && b && c



a || b || c



if (a && b || c && d || e && f) stat;



Needs a small change in the ATG of if statements

```
IfStatement
=  "if"
   "(" Condition<↑x> ")"        (. Code.FJump(x);
                                   Code.il.MarkLabel(x.tLabel);
                                 .)
   Block                        (. Code.il.MarkLabel(x.fLabel); .)
   .
```

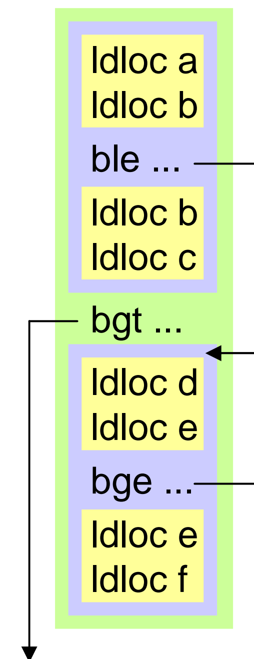
Compiling Boolean Expressions

```
CondFactor<↑Item x>   (. Item y; int op; .)
= Expr<↑x>            (. Code.Load(x); .)
  Relop<↑op>
  Expr<↑y>            (. Code.Load(y);
                         if (!x.type.CompatibleWith(y.type))
                           Error("type mismatch");
                         else if (x.type.IsRefType() &&
                                  op!=Token.EQ && op!=Token.NE)
                           Error("only equality checks ...");
                         x = new Item(op, x.type);
                       .)
  .
```

```
CondTerm<↑Item x>       (. Item y; .)
= CondFactor<↑x>
  {  "&&"               (. Code.FJump(x); .)
     CondFactor<↑y>     (. x.relop = y.relop; x.tLabel = y.tLabel; .)
  }.
```

```
Condition<↑Item x>      (. Item y; .)
= CondTerm<↑x>
  {  "||"               (. Code.TJump(x);
                           Code.il.MarkLabel(x.fLabel);
                         .)
     CondTerm<↑y>       (. x.relop = y.relop; x.fLabel = y.fLabel; .)
  }.
```

```
a>b  &&  b>c  ||  d<e  &&  e<f
```



6. Code Generation

6.1 Overview

6.2 The .NET Common Language Runtime (CLR)

6.3 Items

6.4 Expressions

6.5 Assignments

6.6 Jumps and Labels

6.7 Control Structures

6.8 Methods

Procedure Call

Code pattern

M(a, b);	ldloc a ldloc b call T_{meth}	parameters are passed on the *estack*

Description by an ATG

```
Statement                (. Item x, y; ... .)
= Designator<↑x>
   (  ActParList<↓x>     (. Code.il.EmitCall(Code.CALL, x.sym.meth, null);
                            if (x.type != Tab.noType) Code.il.Emit(Code.POP);
                          .)
   |  "=" Expr<↑y> ";"   (. ... .)
   )
| ... .
```

Function Call

Code pattern

c = M(a, b);	ldloc.s a	parameters are passed on the *estack*
	ldloc.s b	
	call T_{meth}	
	stloc.s c	function value is returned on the *estack*

Standard functions

ord('x')

- *ActParList* loads 'x' onto the *estack*
- the loaded value gets the type of *ordSym* (= *intType*) and *kind* = *Item.Kinds.Stack*

Description by an ATG

```
Factor<↑Item x>
=  Designator<↑x>
   [   ActParList<↓x>      (.  if (x.type == Tab.noType) Error("procedure called as a function");
                               if (x.sym == Tab.ordSym || x.sym == Tab.chrSym) ; // nothing to do
                               else if (x.sym == Tab.lenSym)
                                   Code.il.Emit(Code.LDLEN);
                               else if (x.kind == Item.Kinds.Meth)
                                   Code.il.Emit(Code.CALL, x.sym.meth, null);
                               x.kind = Item.Stack;
                            .)
   ]
| ... .
```

Method Declaration

```
MethodDecl              (. Struct type; int n; .)
= (  Type<↑type>
  |  "void"             (. type = Tab.noType; .)
  )
  ident                 (. curMethod = Tab.Insert(Symbol.Kinds.Meth, token.str, type);
                           Tab.openScope();
                         .)
  "(" [ FormPars ] ")"  (. curMethod.nArgs = Tab.topScope.nArgs;
                           curMethod.locals = Tab.topScope.locals;
                           Code.CreateMetadata(curMethod);
                           if (curMethod.name == "Main") {
                              if (curMethod.type != Tab.noType) Error("method Main must be void");
                              if (curMethod.nArgs != 0) Error("method Main must not have parameters");
                           }
                         .)
  { VarDecl<↓Symbol.Kinds.Local>
  }                     (. curMethod.nLocs = Tab.topScope.nLocs;
                           curMethod.locals = Tab.topScope.locals;
                         .)
  Block<↓undef>         (. if (curMethod.type == Tab.noType) Code.il.Emit(Code.RET);
                           else {  // end of function reached without a return statement
                              Code.il.Emit(Code.NEWOBJ, Code.eeexCtor);
                              Code.il.Emit(Code.THROW);
                           }
                           Tab.closeScope();
                        .)
  .
```

global variable

no-arg constructor of
System.ExecutionEngineException

Formal Parameters

- are entered into the symbol table (as *Arg* symbols of the method scope)
- the method scope counts their number (in *nArgs*)

```
FormPars = FormPar { "," FormPar } .
```

```
FormPar                 (. Struct type; .)
= Type<↑type>
  ident                 (. Tab.Insert(Symbol.Kinds.Arg, token.str, type); .)
  .
```

Actual Parameters

- load them to *estack*
- check if they are assignment compatible with the formal parameters
- check if the number of actual and formal parameters corresponds

```
ActPars<↓Item m>   (. Item ap; int aPars = 0, fPars = 0; .)
=  "("             (. if (m.kind != Item.Kinds.Meth) { Error("not a method"); m.sym = Tab.noSym; } .)
                      fPars = m.sym.nArgs;
                      Sym fp= m.sym.locals;
                    .)
   [  Expr<↑ap>    (. Code.Load(ap); aPars++;
                      if (fp != null && fp.kind == Symbol.Kinds.Arg) {
                         if (!ap.type.AssignableTo(fp.type)) Error("parameter type mismatch");
                         fp = fp.next;
                      }
                    .)
      { "," Expr<↑ap> (. Code.Load(ap); aPars++;
                      if (fp != null && fp.kind == Symbol.Kinds.Arg) {
                         if (!ap.type.AssignableTo(fp.type)) Error("parameter type mismatch");
                         fp = fp.next;
                      }
                    .)
      }
   ]               (. if (aPars > fPars) Error("more actual than formal parameters");
                      else if (aPars < fPars) Error("less actual than formal parameters");
                    .)
   ")" .
```

return Statement

```
Statement<↓Label break>
= ...
|  "return"
   (                    (.  if (curMethod.type == Tab.noType)
                                Error("void method must not return a value");
                         .)
      Expr<↑x>          (.  Code.Load(x);
                            if (x.type.AssignableTo(curMethod.type))
                                Error("type of return value must be assignable to method type");
                         .)
   |                    (.  if (curMethod.type != Tab.noType) Error("return value expected"); .)
   )                    (.  Code.il.Emit(Code.RET); .)
   ";" .
```