



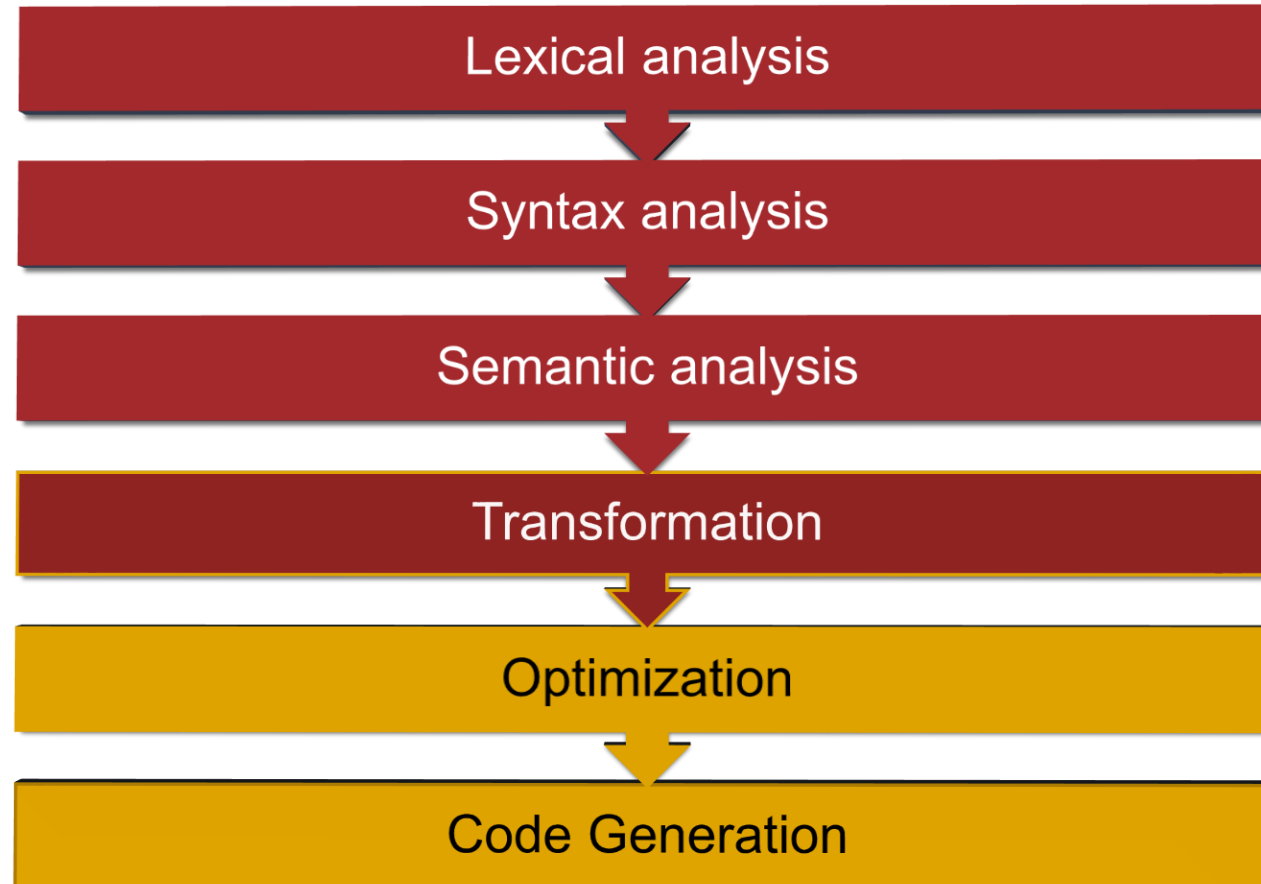
# MODEL-BASED SOFTWARE DEVELOPMENT

## LECTURE VI.

### OPTIMIZATION, OBFUSCATION, CODE GENERATION

Dr. Gergely Mezei,  
Dr. Ferenc Somogyi

# THE CLASSIC PHASES OF COMPILATION



# TODAY'S AGENDA

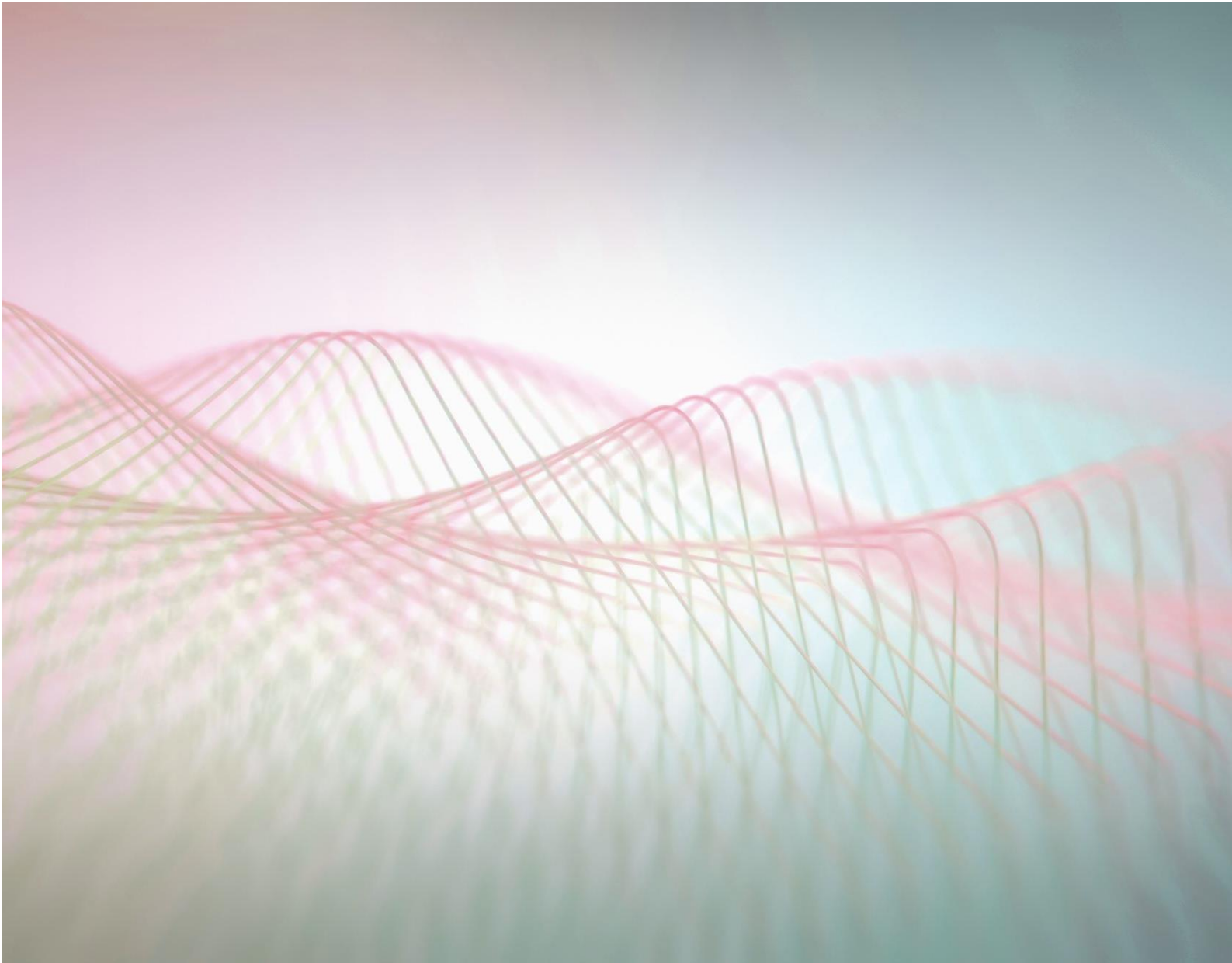
**I. OPTIMIZATION**

**II. OBFUSCATION**

**III. CODE GENERATION**

**IV. EDITOR SUPPORT**





# DATA-FLOW OPTIMIZATION

# DATA-FLOW OPTIMIZATION

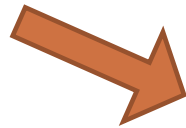
- Common subexpression elimination
- Constant/copy propagation
- Available expression analysis
- Dead code elimination
- Live-variable analysis

**Only between basic blocks (local)!**

# COMMON SUBEXPRESSION ELIMINATION

- Avoid computing the same expression twice
  - If the operands have not changed and the operation is free of side effects

```
...  
a = b ⊙ c + 5  
e = (b ⊙ c) * 2  
...
```



```
...  
tmp = b ⊙ c  
a = tmp + 5  
e = tmp * 2  
...
```

# CONSTANT PROPAGATION

- If a variable is assigned a constant value, then the variable can be switched with the value

```
...  
a = 7  
b = a ⊗ 3  
...  
if (a > 4) { write("."); }  
...
```



```
...  
a = 7  
b = 7 ⊗ 3  
...  
write(".");  
...
```

# COPY PROPAGATION

- If two variables have the same value, they can be switched

```
...  
a = b ⊙ c  
d = a  
e = d * 5 + 8  
...
```



```
...  
a = b ⊙ c  
d = a  
e = a * 5 + 8  
...
```



# OPTIMIZATION - EXAMPLE

```
...  
b = a * a;  
c = a * a;  
d = b + c;  
e = 1;  
f = b + b;  
g = f * e;  
...
```

# AVAILABLE EXPRESSION ANALYSIS

- *Available expression*: if there exists a variable that contains the value of the expression, then the expression can be switched with the variable
- Available expression analysis
  - > Start with an empty set
  - > For each  $a = b \odot c$  expression
    - Remove expressions that contain  $a$
    - Add the expression  $a = b \odot c$

# AVAILABLE EXPRESSION ANALYSIS

```
a = b;
```

```
c = b;
```

```
d = a + b;
```

```
e = a + b;
```

```
d = b;
```

```
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{}  
a = b;  
  
c = b;  
  
d = a + b;  
  
e = a + b;  
  
d = b;  
  
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{}  
a = b;  
{ a = b }  
c = b;  
  
d = a + b;  
  
e = a + b;  
  
d = b;  
  
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = b;  
{ a = b, c = b }  
d = a + b;  
  
e = a + b;  
  
d = b;  
  
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = b;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = a + b;  
  
d = b;  
  
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = b;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = a + b;  
{ a = b, c = b, d = a + b, e = a + b }  
d = b;  
  
f = a + b;
```



# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = b;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = a + b;  
{ a = b, c = b, d = a + b, e = a + b }  
d = b;  
{ a = b, c = b, d = b, e = a + b }  
f = a + b;
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = b;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = a + b;  
{ a = b, c = b, d = a + b, e = a + b }  
d = b;  
{ a = b, c = b, d = b, e = a + b }  
f = a + b;  
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = a;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = a + b;  
{ a = b, c = b, d = a + b, e = a + b }  
d = b;  
{ a = b, c = b, d = b, e = a + b }  
f = a + b;  
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
  
c = a;  
{ a = b, c = b }  
  
d = a + b;  
{ a = b, c = b, d = a + b }  
  
e = d;  
{ a = b, c = b, d = a + b, e = a + b }  
  
d = b;  
{ a = b, c = b, d = b, e = a + b }  
  
f = a + b;  
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
  
c = a;  
{ a = b, c = b }  
  
d = a + b;  
{ a = b, c = b, d = a + b }  
  
e = d;  
{ a = b, c = b, d = a + b, e = a + b }  
  
d = a;  
{ a = b, c = b, d = b, e = a + b }  
  
f = a + b;  
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# AVAILABLE EXPRESSION ANALYSIS

```
{ }  
a = b;  
{ a = b }  
c = a;  
{ a = b, c = b }  
d = a + b;  
{ a = b, c = b, d = a + b }  
e = d;  
{ a = b, c = b, d = a + b, e = a + b }  
d = a;  
{ a = b, c = b, d = b, e = a + b }  
f = e;  
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# DEAD CODE ELIMINATION

- If the left side of an assignment is never read, then the assignment can be removed

```
...  
a = b ⊙ c  
d = a  
e = a * 5 + 8  
...
```



```
...  
a = b ⊙ c  
d = a  
e = a * 5 + 8  
...
```

# LIVENESS ANALYSIS

- Dead code analysis: **liveness analysis**
- A variable is *live* at a given point of the program, if its value is read at least once before being overwritten
- Dead code elimination using liveness analysis:
  - > For each variable, liveness analysis
  - > Delete every assignment that has a non-live variable on the left side
- Compute the statements of the basic block in reverse-order
- Some variables may be considered to be live by default (e.g. if it influences output)



# LIVENESS ANALYSIS

```
a = b;
```

```
c = a;
```

```
d = a + b;
```

```
e = d;
```

```
d = a;
```

```
f = e;
```

# LIVENESS ANALYSIS

```
a = b;
```

```
c = a;
```

```
d = a + b;
```

```
e = d;
```

```
d = a;
```

```
f = e;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

```
a = b;
```

```
c = a;
```

```
d = a + b;
```

```
e = d;
```

```
d = a;
```

```
{ b, d, e }
```

```
f = e;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

```
a = b;
```

```
c = a;
```

```
d = a + b;
```

```
e = d;
```

```
{ a, b, e }
```

```
d = a;
```

```
{ b, d, e }
```

```
f = e;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

a = b;

c = a;

d = a + b;

{ a, b, d }

e = d;

{ a, b, e }

d = a;

{ b, d, e }

f = e;

{ b, d }

# LIVENESS ANALYSIS

a = b;

c = a;

*{ a, b }*

d = a + b;

*{ a, b, d }*

e = d;

*{ a, b, e }*

d = a;

*{ b, d, e }*

f = e;

*{ b, d }*

# LIVENESS ANALYSIS

a = b;

{ a, b }

c = a;

{ a, b }

d = a + b;

{ a, b, d }

e = d;

{ a, b, e }

d = a;

{ b, d, e }

f = e;

{ b, d }

# LIVENESS ANALYSIS

*{ b }*

a = b;

*{ a, b }*

c = a;

*{ a, b }*

d = a + b;

*{ a, b, d }*

e = d;

*{ a, b, e }*

d = a;

*{ b, d, e }*

f = e;

*{ b, d }*



# LIVENESS ANALYSIS

*{ b }*

a = b;

*{ a, b }*

c = a;

*{ a, b }*

d = a + b;

*{ a, b, d }*

e = d;

*{ a, b, e }*

d = a;

*{ b, d, e }*

*{ b, d }*

# LIVENESS ANALYSIS

*{ b }*

a = b;

*{ a, b }*

*{ a, b }*

d = a + b;

*{ a, b, d }*

e = d;

*{ a, b, e }*

d = a;

*{ b, d, e }*

*{ b, d }*

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
e = d;
```

```
d = a;
```

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
e = d;
```

```
d = a;  
{ b, d }
```

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
e = d;
```

```
{ a, b }
```

```
d = a;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
{ a, b, d }
```

```
e = d;
```

```
{ a, b }
```

```
d = a;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

```
a = b;  
{ a, b }  
d = a + b;  
{ a, b, d }  
e = d;  
{ a, b }  
d = a;  
{ b, d }
```

# LIVENESS ANALYSIS

{ *b* }

a = b;

{ *a, b* }

d = a + b;

{ *a, b, d* }

e = d;

{ *a, b* }

d = a;

{ *b, d* }



# LIVENESS ANALYSIS

{ *b* }

a = b;

{ *a, b* }

d = a + b;

{ *a, b, d* }

{ *a, b* }

d = a;

{ *b, d* }

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
d = a;
```

# LIVENESS ANALYSIS

```
a = b;
```

```
d = a + b;
```

```
d = a;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

`a = b;`

`d = a + b;`

`{ a, b }`

`d = a;`

`{ b, d }`

# LIVENESS ANALYSIS

```
a = b;
```

```
{ a, b }
```

```
d = a + b;
```

```
{ a, b }
```

```
d = a;
```

```
{ b, d }
```

# LIVENESS ANALYSIS

$\{ b \}$

$a = b;$

$\{ a, b \}$

$d = a + b;$

$\{ a, b \}$

$d = a;$

$\{ b, d \}$

# LIVENESS ANALYSIS

$\{ b \}$

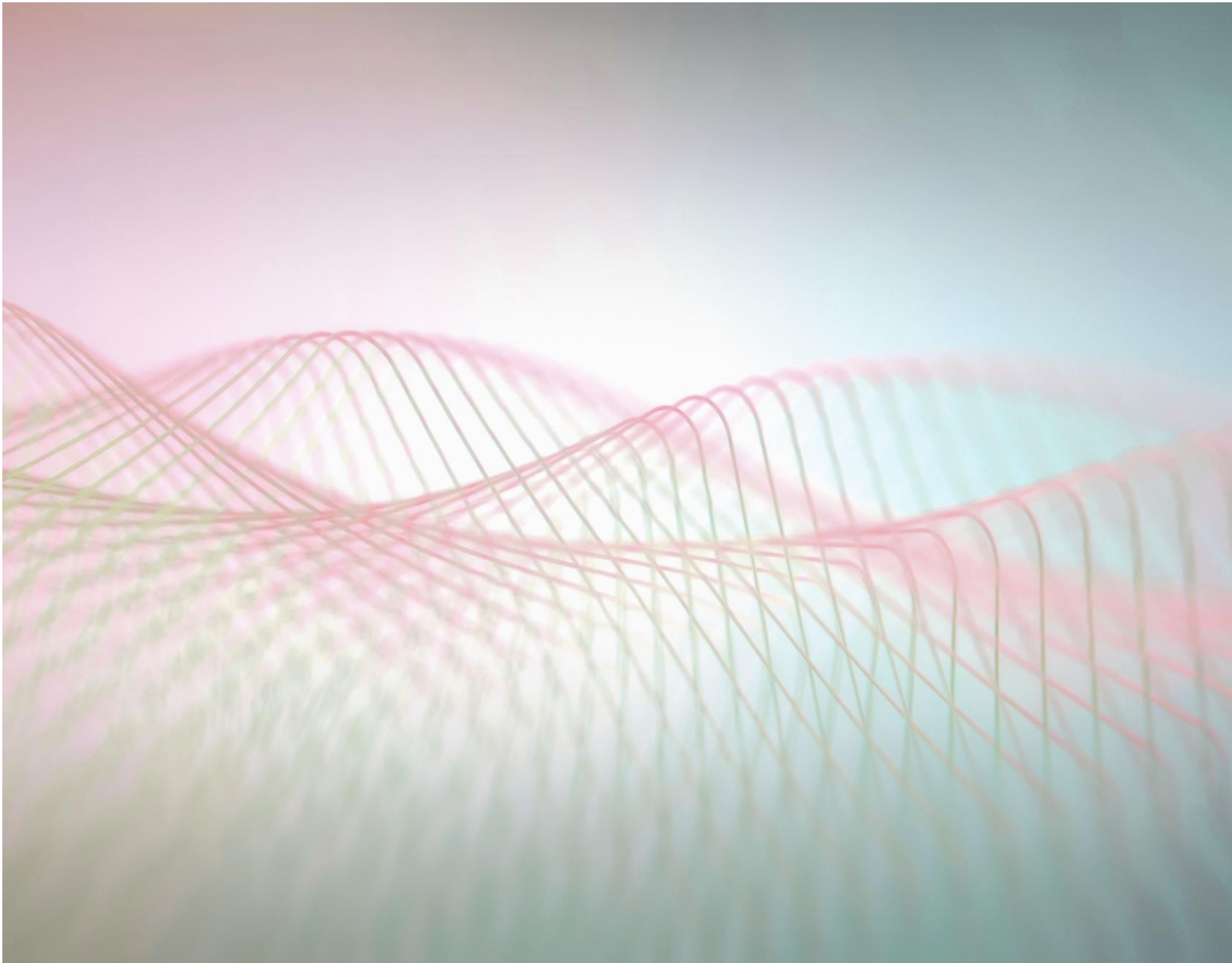
$a = b;$

$\{ a, b \}$

$\{ a, b \}$

$d = a;$

$\{ b, d \}$



# CODE MOTION



# CODE MOTION

- Code sinking
- Code factoring
- Code scheduling
- Loop invariant code motion

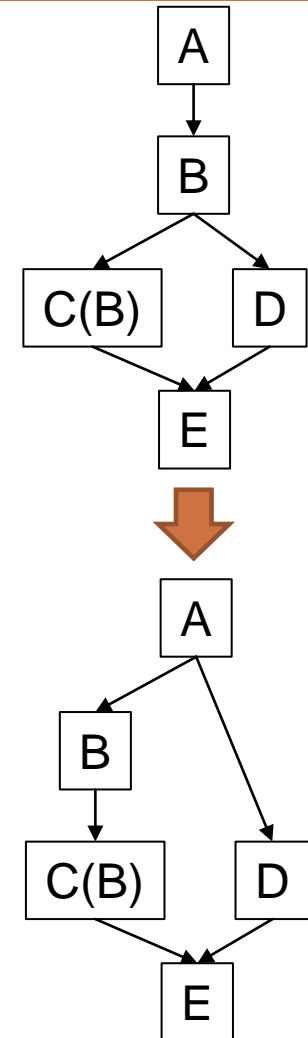
# CODE SINKING

- Execute statements only where they are necessary
  - > Similar to dead code, but can be used for statements with side effects as well

```
...  
  b=5;  
  if (a>5) { write(b); } else { write("none"); }  
...
```



```
...  
  if (a>5) { b=5; write(b); } else { write("not ok"); }  
...
```



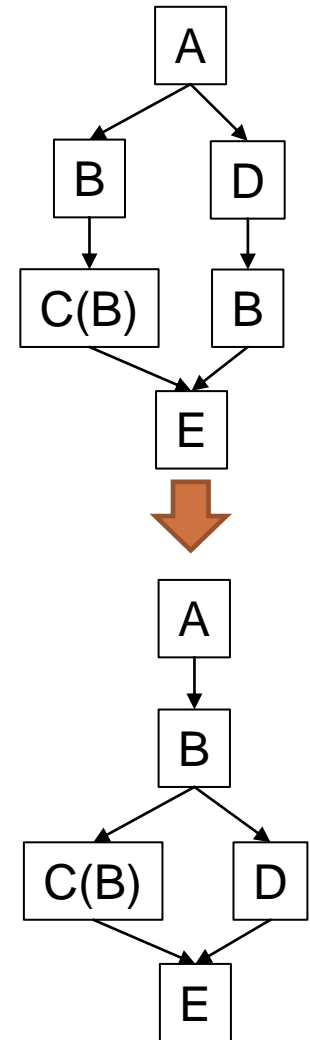
# CODE FACTORING

- Execute common statements in the common execution path
  - > The “reverse” of code sinking
  - > Often, the code is factorized into smaller fragments

```
...  
if (a>b) { c=5; write(c); } else { write("none"); c=5; }  
...
```



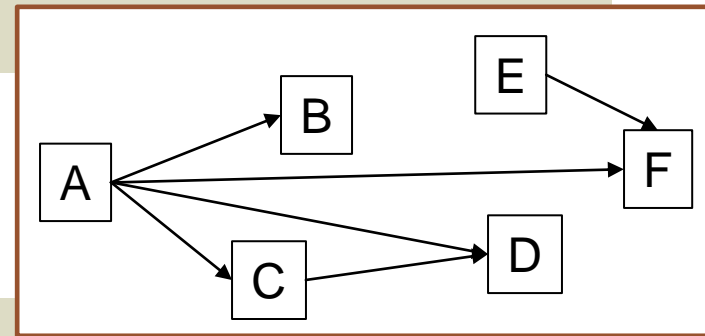
```
...  
c=5;  
if (a>b) { write(c); } else { write("none"); }  
...
```



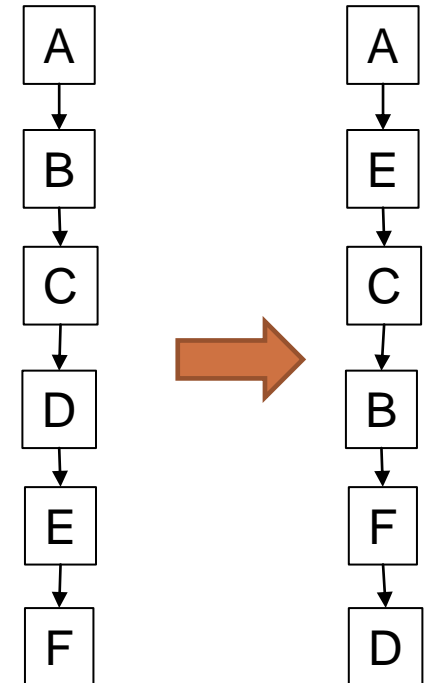
# CODE SCHEDULING

- Avoid long waiting queues!
  - > Parallel / effective execution can be faster in a different order of statements
  - > Optimize dependency graph

```
...  
a=f(); b=a+1; c=f2(a); d=a+c; e=f3(); f=e+a;  
...
```



```
...  
a=f(); e=f3(); c=f2(a); b=a+1; f=e+a; d=a+c;  
...
```



# LOOP INVARIANT CODE MOTION

- Move invariant code before the loop!
  - > Unnecessary to compute in every iteration

```
...  
int i = 0;  
while (i < j) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
a = b + c;  
while (i < j) {  
    d[i] = i + a * a;  
    ++i;  
}  
...
```

# LOOP INVARIANT CODE MOTION

- Move invariant code before the loop!
  - > Can also move partly invariant code

```
...  
int i = 0;  
while (i < j) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
a = b + c;  
int const a' = a * a;  
while (i < j) {  
    d[i] = i + a';  
    ++i;  
}  
...
```

# LOOP INVARIANT CODE MOTION

- Move invariant code before the loop!
  - > It is not guaranteed that the program enters the loop at all!
  - > Must be handled

```
...
int i = 0;
while (i < j) {
    a = b + c;
    d[i] = i + a * a;
    ++i;
}
...
```



```
...
int i = 0;
if (i < j) {
    a = b + c;
    int const a' = a * a;
    while (i < j) {
        d[i] = i + a';
        ++i;
    }
}...

```

# LOOP INVARIANT CODE MOTION

- Move invariant code before the loop!
  - > What if the loop condition has a side effect?

```
...  
int i = 0;  
while (fn(i)) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
if (fn(i)) {  
    a = b + c;  
    int const a' = a * a;  
    while (fn(i)) {  
        d[i] = i + a';  
        ++i;  
    }  
}...
```



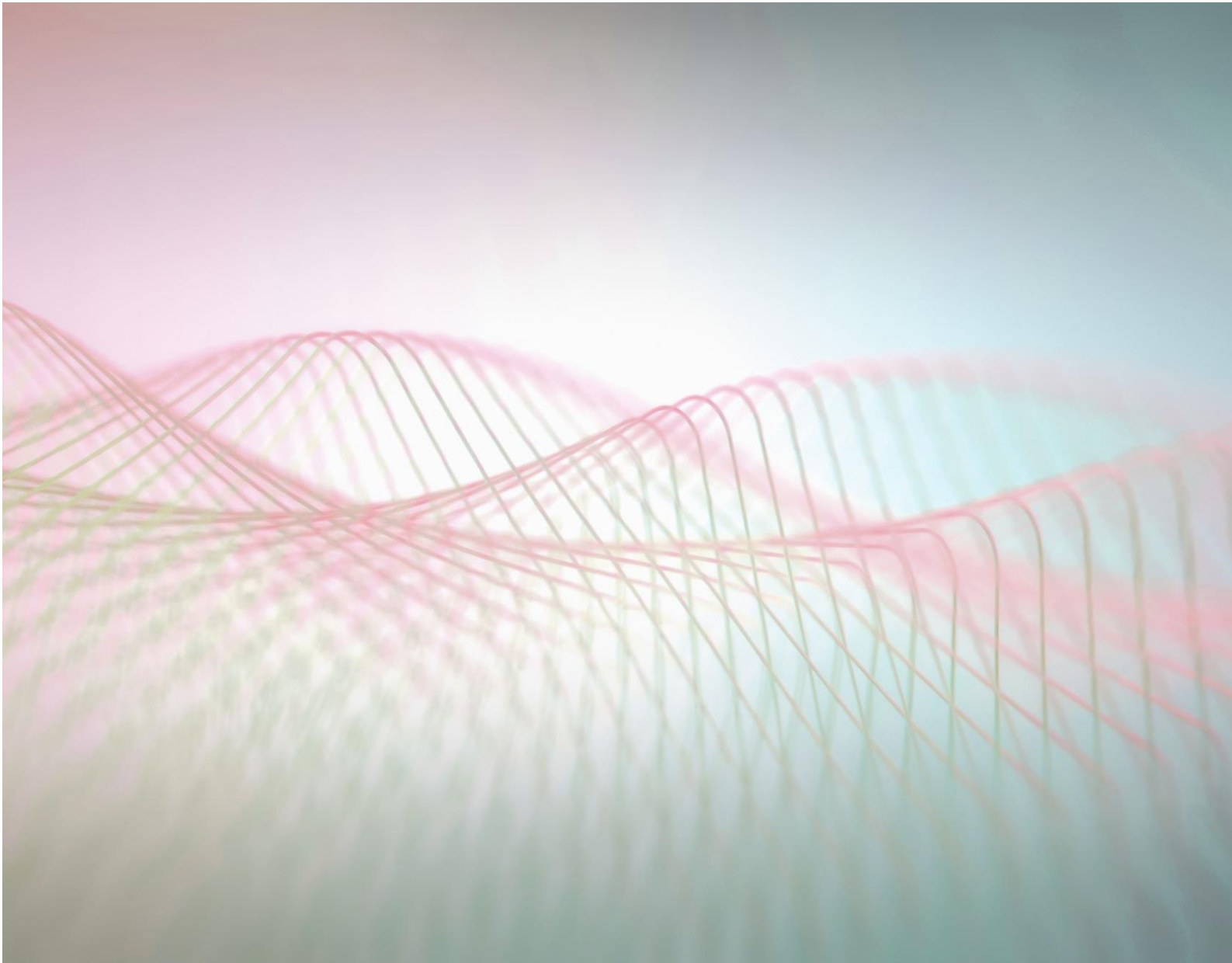
# LOOP INVARIANT CODE MOTION

- Move invariant code before the loop!
  - > What if the loop condition has a side effect?

```
...  
int i = 0;  
while (fn(i)) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
if (fn(i)) {  
    a = b + c;  
    int const a' = a * a;  
    do {  
        d[i] = i + a';  
        ++i;  
    } while (fn(i));  
}...
```



# ADDITIONAL OPTIMIZATION TECHNIQUES

# USING INLINE FUNCTIONS

- Inline function – instead of a function call, copy the code of the function
  - > Manual or automatic
  - > Not identical to macros (where the source code is changed before compilation)
- Advantages
  - > Faster than a function call (function pointers, stack, registers)
  - > The scope of local/global optimization increases
    - Moving invariant code (similar to loops)
    - Register optimization
- Disadvantages
  - > Increased memory costs (duplicate data structures)
  - > Operational cache overflow possible

# REGISTER ALLOCATION

- Register – fast, easy to access, small container
  - > Limited ( $< 32$ )
  - > Would have to load every live variable
  - > When no more space  $\rightarrow$  RAM
  - > Registers are sometimes not independent (e.g. 32 bit register may be used as 2x16 bits)
- What to remove, what to keep in the registers?
  - > NP complete problem (can be reduced to graph coloring, color = register)
  - > Rematerialization instead of storing (e.g. constant integer values)

# ADDITIONAL TECHNIQUES

- Elementary techniques
  - > Eliminate unreachable code
  - > Unfolding and merging loops
  - > Omit index boundary checking
- Platform-dependent (e.g. operating system, CPU architecture) optimization
- Domain-dependent optimization

# TODAY'S AGENDA

I. OPTIMIZATION

II. OBFUSCATION

III. CODE GENERATION

IV. EDITOR SUPPORT



# OBFUSCATION

- Goal: make it harder to...
  - > Understand how the program works
  - > Reverse engineer the program (from machine/IL code)
  - > Analyse the program
- Mostly as a measure of defense
  - > Against security issues
  - > Against theft / copy
- Perfect defense does not exist; can only make reverse engineering harder, not impossible!
- How does it work?
  - > Similar to optimization
  - > Largely dependent on our limitations (time, importance of protection against reverse engineering etc.)

# TECHNIQUES

- Name obfuscation (lexical transformation)
- Data obfuscation (modifying data structures)
- Control flow obfuscation
- Debug information obfuscation



# NAME OBFUSCATION

- Replace meaningful identifiers (classes, methods, variables, functions etc.) with meaningless text
- Restrictions
  - > Names of built-in classes and API-s are fixed
  - > Names of classes to be serialized are fixed
  - > Tricky in case of native access and reflection

# DATA OBFUSCATION

- Change the way data is stored in the memory
- Techniques
  - > Change encoding
  - > Data aggregation (arrays, collections)
  - > Change the role of data (e.g. local vs global)

# EXAMPLE: NAME AND DATA OBFUSCATION

```
function foo( arg1)
{
    var myVar1 = "some string"; //first comment
    var intVar = 24 * 3600; //second comment
    /* here is
       a long multi-line comment . . . */
    document.write( "vars are:" + myVar1 + " " + intVar + " " + arg1) ;
} ;
```



```
function z001c775808( z3833986e2c) { var z0d8bd8ba25=
"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\x67"; var z0ed9bcbcc2=
(0x90b+785-0xc04)* (0x1136+6437-0x1c4b); document.write(
"\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+ z0d8bd8ba25+ "\x20"+
z0ed9bcbcc2+ "\x20"+ z3833986e2c);};
```

# EXAMPLE: NAME AND DATA OBFUSCATION

## ■ Variables, identifiers:

- > foo → z001c775808
- > arg1 → z3833986e2c
- > myvar1 → z0d8bd8ba25
- > intvar → z0ed9bcbcc2

## ■ Integer representation:

- > 20 → (0x90b+785-0xc04)
- > 3600 → (0x1136+6437-0x1c4b)

## ■ Printing:

- > "vars are" → \x76\x61\x72\x73\x20\x61\x72\x65\x3a
- > Space → \x20

```
function foo( arg1)
{
    var myVar1 = "some string";
    //first comment
    var intVar = 24 * 3600;
    //second comment
    /* here is
       a long multi-line
    comment ... */
    document.write( "vars are:" +
myVar1 + " " + intVar + " " +
+ arg1) ;
} ;
```

```
function z001c775808( z3833986e2c) { var z0d8bd8ba25=
"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\x67"; var z0ed9bcbcc2=
(0x90b+785-0xc04)* (0x1136+6437-0x1c4b); document. write(
"\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+ z0d8bd8ba25+ "\x20"+
z0ed9bcbcc2+ "\x20"+ z3833986e2c);};
```

# CONTROL FLOW OBFUSCATION

- Change control flow
- Same result, but harder to understand
- Techniques
  - > Modified control, e.g. inline methods instead of method calls
  - > Change the order of statements
  - > Change computations
    - Insert unreachable code
    - Insert unconditional jumps
    - Split statement blocks to conditional statements
    - Always true/false branches

# EXAMPLE: CONTROL FLOW OBFUSCATION

```
public int CompareTo(Object o)
{
    int n = occurrences - ((WordOccurrence)o).occurrences;
    if (n == 0)
    {
        n = String.Compare(word, ((WordOccurrence)o).word);
    }
    return(n);
}
```



```
public virtual int _a(Object A_0)
{
    int local0; int local1;
    local0 = this.a - (c) A_0.a;
    if (local0 != 0) goto i0;
    goto i1;
    while (true) {
        return local1;
        i0: local1 = local0;}
    i1: local0 = System.String.Compare(this.b, (c) A_0.b); goto i0;
}
```

# DEBUG INFORMATION OBFUSCATION

- Remove debug information
  - > Stack trace
  - > Line numbers
  - > File names, etc.

# TODAY'S AGENDA

I. OPTIMIZATION

II. OBFUSCATION

III. CODE GENERATION

IV. EDITOR SUPPORT





# CODE GENERATION

- Code generation: generate executable code from the inner representation (optimized intermediate code) of the compiler
  - > The target is not necessarily binary (machine) code!
- Based on the nature of the input
  - > 3-address code (Three-Address Code, 3AC)
  - > Syntax trees
  - > etc.
- Based on the nature of the output
  - > Machine code
  - > Intermediate code (e.g. LLVM, IL)
  - > High-level programming language code (transpiler)

# CODE GENERATION – MACHINE CODE

## ■ Code generation

- > Transform code to the operations supported by the target architecture
- > Manage register allocations – RegisterDescriptor + getReg
- > Address administration (address of variables, can change during runtime) – AddressDescriptor

## ■ $x = y \text{ op } z$

- > getReg call to get the address of the result (L)
- > Find the address of Y (AddressDescriptor), then copy: **MOV y L**
- > Find the address of Z (AddressDescriptor), then execute the operation: **OP z L**
- > Result in L, if L is a register, then the descriptor of x must be updated (contained by L)

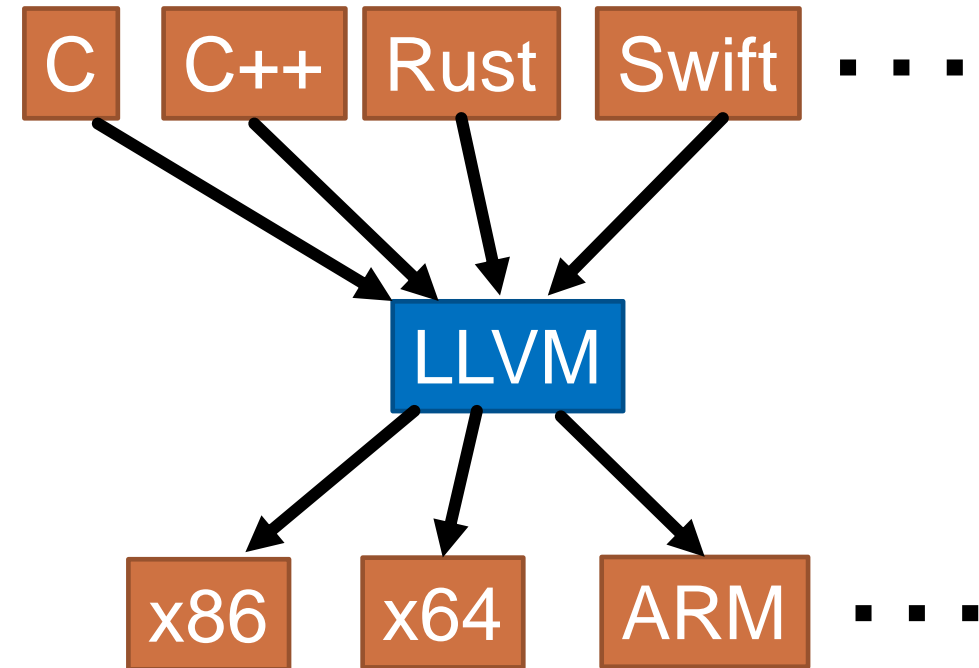
# CODE GENERATION – MACHINE CODE

- The result of the generation can be executed directly
- Advantages
  - > Optimal speed and size can be reached
  - > Fast, can make full use of the nature of the architecture
  - > Anything can be expressed, no language limitations
- Disadvantages
  - > Low-level, difficult to write
  - > Every check (e.g. stack overflow) has to be done manually
  - > Debugging the generated code is difficult

# CODE GENERATION – INTERMEDIATE CODE

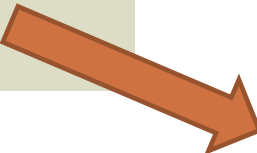
## ■ LLVM

- > Frontend: platform-independent SSA statement list
- > Backend: platform-dependent binary code
- > ~ “readable assembly”
- > Infinite virtual registers
- > Has its own platform-independent typesystem
- > Built-in optimization (e.g. dead code, common subexpression)



# LLVM - EXAMPLE

```
function int factorial(int n){  
    if (n==0) {  
        1  
    } else {  
        n * factorial(n - 1)  
    }  
}
```



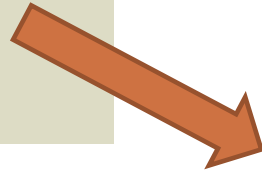
```
define i32 @factorial(i32) {  
  
entry:  
    %eq = icmp eq i32 %0, 0    // n == 0  
    br i1 %eq, label %then, label %else  
  
then:                                ; preds = %entry  
    br label %ifcont  
  
else:                                ; preds = %entry  
    %sub = sub i32 %0, 1        // n - 1  
    %2 = call i32 @factorial(i32 %sub)    // factorial(n-1)  
  
    %mult = mul i32 %0, %2      // n * factorial(n-1)  
    br label %ifcont  
  
ifcont:                              ; preds = %else, %then  
    %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]  
    ret i32 %iftmp  
}
```

# CODE GENERATION – INTERMEDIATE CODE

- Common Intermediate Language (CIL)
  - > Object-oriented (objects, methods, attributes)
  - > Stack-based (instead of registers)
  - > Platform-independent
  - > Capable of storing meta-data
- Run by a Just-in-Time (JIT) compiler
  - > Ahead-of-time compilation is available
  - > Security checks
- Disassembler segítségével magas szintű programnyelvekre is alakítható

# IL - EXAMPLE

```
function int factorial(int n){  
    if (n==0) {  
        return 1  
    } else {  
        n * factorial(n - 1)  
    }  
}
```



```
.method public hidebysig  
instance int32 factorial (int32 n) cil managed {  
    // Method begins at RVA 0x2064  
    // Code size 31 (0x1f)  
    .maxstack 4  
    .locals init ([0] bool,[1] int32)  
  
    // {  
    IL_0000: nop  
  
    // if (n == 0)  
    IL_0001: ldarg.1  
    IL_0002: ldc.i4.0  
    IL_0003: ceq  
    IL_0005: stloc.0  
    IL_0006: ldloc.0  
    IL_0007: brfalse.s IL_000e
```

```
    // return 1;  
    IL_0009: nop  
    IL_000a: ldc.i4.1  
    IL_000b: stloc.1  
  
    // (no C# code)  
    IL_000c: br.s IL_001d  
  
    // return n * factorial(n - 1);  
    IL_000e: nop  
    IL_000f: ldarg.1  
    IL_0010: ldarg.0  
    IL_0011: ldarg.1  
    IL_0012: ldc.i4.1  
    IL_0013: sub  
    IL_0014: call instance int32  
                                     Host::factorial(int32)  
  
    IL_0019: mul  
    IL_001a: stloc.1  
  
    // (no C# code)  
    IL_001b: br.s IL_001d  
    IL_001d: ldloc.1  
    IL_001e: ret  
} // end of method Host::factorial
```

# CODE GENERATION – INTERMEDIATE CODE

## ■ Advantages

- > Easier to generate than machine-code and easier to read for humans
- > Less technical difficulties (e.g. register allocation)
- > Can be very effective, because it is low level
- > Can compile to multiple platforms

## ■ Disadvantages

- > A special compiler is necessary
- > Generation of the target code (intermediate code) is not easy



# CODE GENERATION - TRANSPILER

- Transpiler: compile to a different, typically higher-level target language
  - > Traverse the annotated syntax tree and generate code using templates
  - > Typical solution: every AST element is capable of generating its own target code, hierarchic decomposition
  - > Template-based code generation is possible
  - > Multiple transpilers can be “chained” after each other

```
if (x > 5) then
begin
  while (y < z) do
  begin
    y := x;
  end
end
```



```
if (x > 5)
  while (y < z) {
    y = x;
  }
```

# CODE GENERATION - TRANSPILER

## ■ Advantages

- > Easy to create the code generator
- > Machine code is generated by compiling the code written in the target language
  - The compiler of the target language can be reused
  - Optimization done externally (e.g. Visual Studio, GCC)
  - Security checks (e.g. memory segmentation faults)
- > Easier to verify the semantics of the generated code (C# vs. Assembly)
- > The generated code is easy to integrate with existing applications / tools

## ■ Disadvantages

- > Cannot be optimized as well as machine code
- > The capabilities of the target language restrict the possibilities

## AFTER CODE GENERATION: LINK AND BUILD

- The output of code generation can not always be run on its own: multiple modules ('object' file)
- Link: concatenate object files into a runnable format  
Build: transform linked files into an executable
- Linking – advantages
  - > Instead of one monolithic file, uses multiple smaller (less complex)
  - > Easier error management and incremental compilation
  - > Compiled object files can be reused

# TODAY'S AGENDA

I. OPTIMIZATION

II. OBFUSCATION

III. CODE GENERATION

IV. EDITOR SUPPORT



# TEXTUAL EDITORS

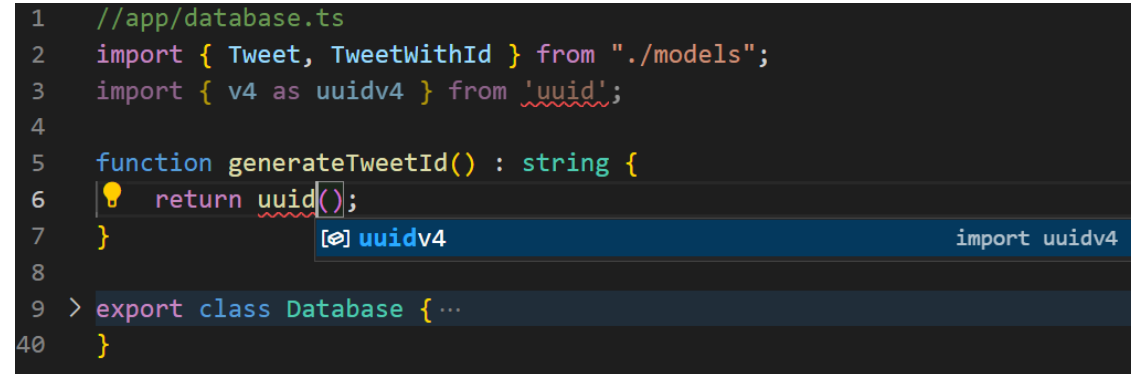
- Textual languages – good tool support
  - Could also use Notepad, but not recommended 😊
  - More advanced text editors exist (e.g. Vim, Emacs, Atom)
    - Integrating these with a custom language can be problematic
  - Integrated development environments (IDE) (e.g. VSCode, Eclipse, Eclipse Theia, IntelliJ IDEA, NetBeans)
    - Many of these are extendable (plugins, extensions, etc.), which makes integration easier
- Integrate the language and the editor
  - In accordance with the steps of compilation (mainly during semantic analysis)



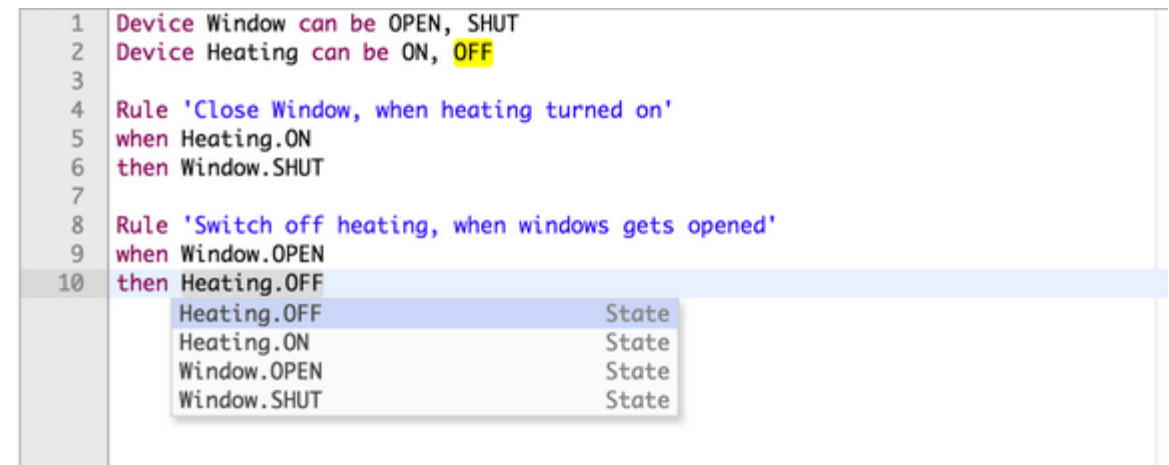
# TEXTUAL EDITORS

- Editor functions
  - Syntax highlighting
  - Code completion
  - Signal errors
  - Support refactoring
  - Folding
  - Quick fixes
  - Etc.

```
1 //app/database.ts
2 import { Tweet, TweetWithId } from "../models";
3 import { v4 as uuidv4 } from 'uuid';
4
5 function generateTweetId() : string {
6   return uuid();
7 }
8
9 > export class Database { ...
40 }
```



```
1 Device Window can be OPEN, SHUT
2 Device Heating can be ON, OFF
3
4 Rule 'Close Window, when heating turned on'
5 when Heating.ON
6 then Window.SHUT
7
8 Rule 'Switch off heating, when windows gets opened'
9 when Window.OPEN
10 then Heating.OFF
```



Heating.OFF	State
Heating.ON	State
Window.OPEN	State
Window.SHUT	State

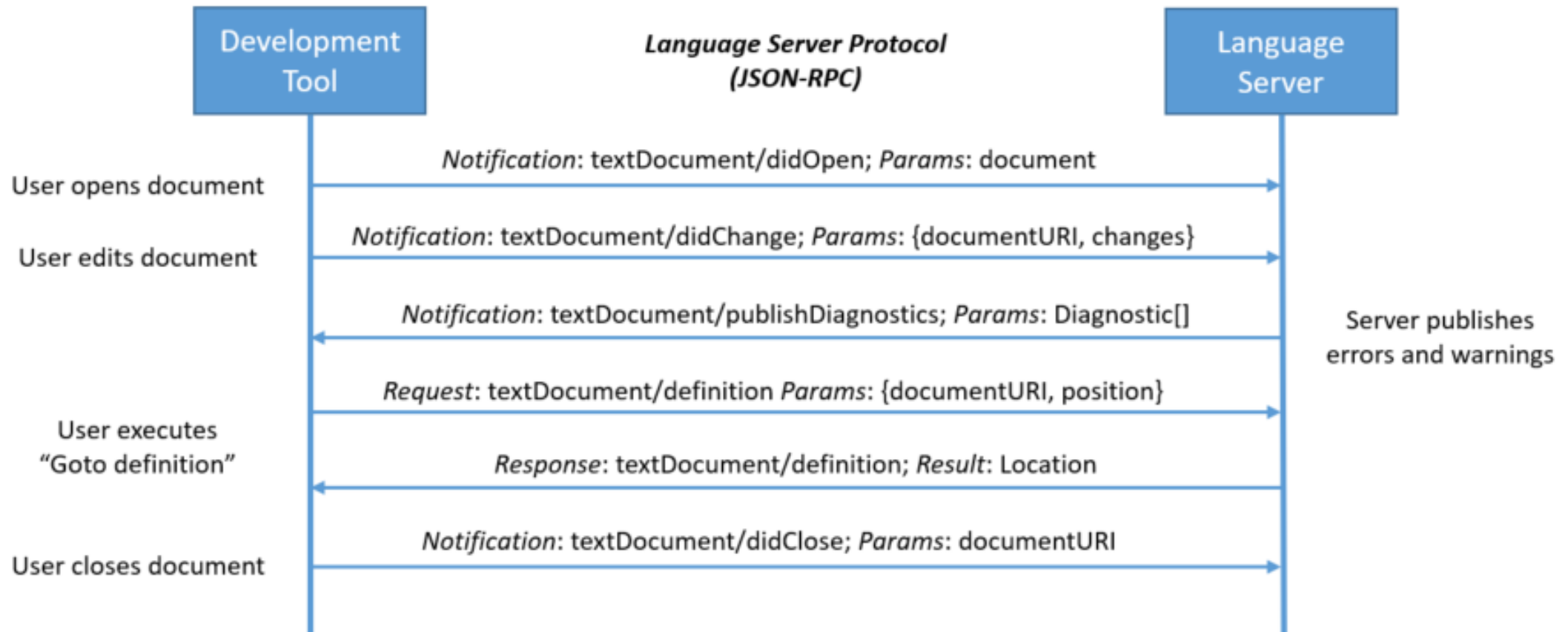
Source:

<https://www.eclipse.org/Xtext/>

# Language Server Protocol (LSP)

- JSON-RPC based protocol
- Communication between textual editors and language servers
- Support for main editor features
  - Syntax highlighting, signal errors, code completion, refactoring etc.
- Why is this useful?
  - Write once, connect with multiple editors
  - Many popular editors support it (e.g. VSCode, Monaco, Eclipse IDE, Eclipse Theia)
  - For many languages, a Language Server implementation already exists
    - <https://langserver.org/>

# Language Server Protocol (LSP)



Source: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>

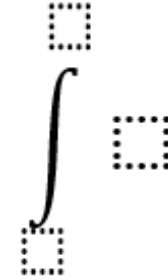


# PROJECTIONAL EDITORS

- Traditional editors
  - Discussed so far
  - Well-tried and widespread
  - Any textual editor can be used (with recommended editor functions)
- Projectional editors
  - The elements of the syntax trees are indirectly visualized
  - *Building syntax trees is not needed!*
  - Special tooling is required
  - E.g. JetBrains MPS, Gentleman

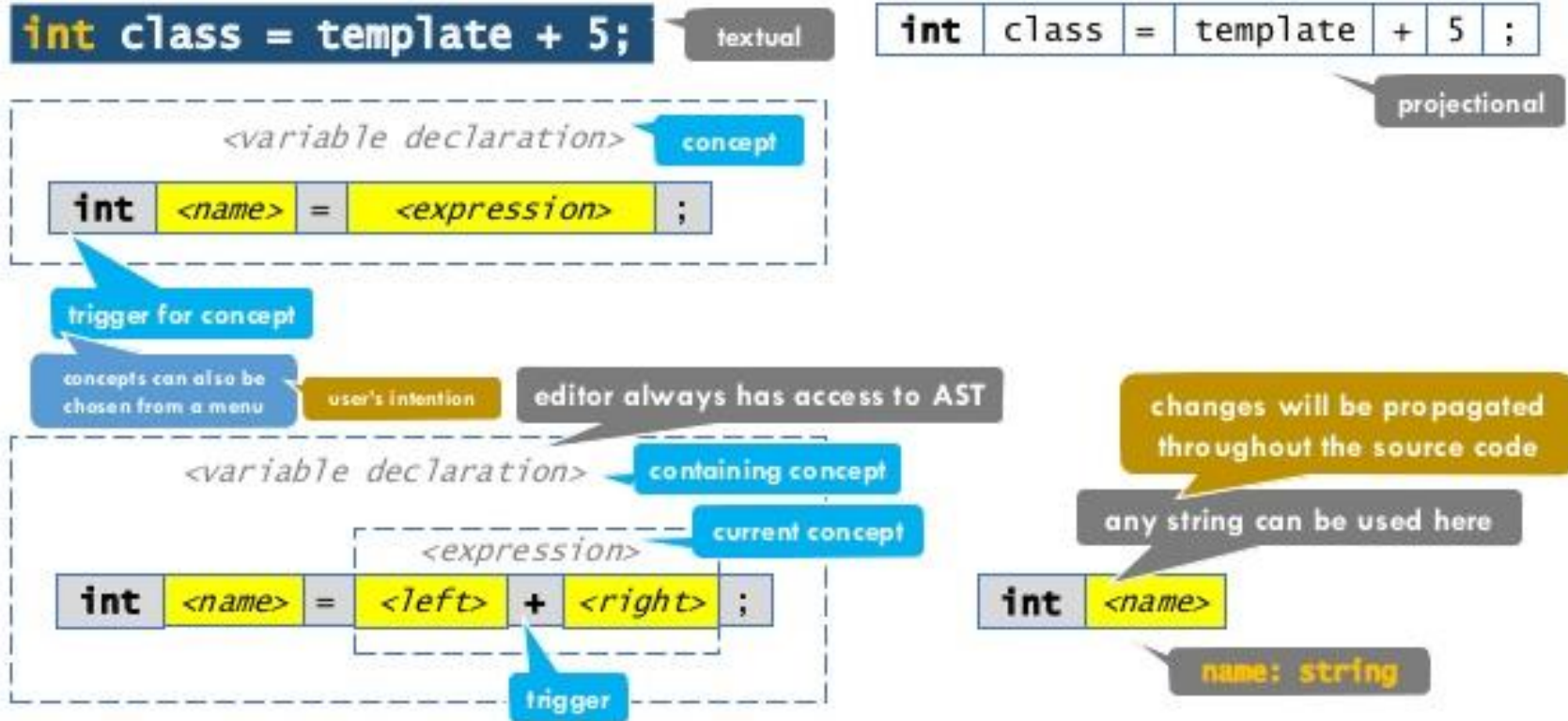
# PROJECTIONAL EDITORS - EXAMPLE

- Semantic model
  - Lower and upper bounds of the integral
  - Integrand
- Concrete expression:  $\int_{-\infty}^{\infty} e^{-x^2} dx$
- Can be mapped directly to the semantic model



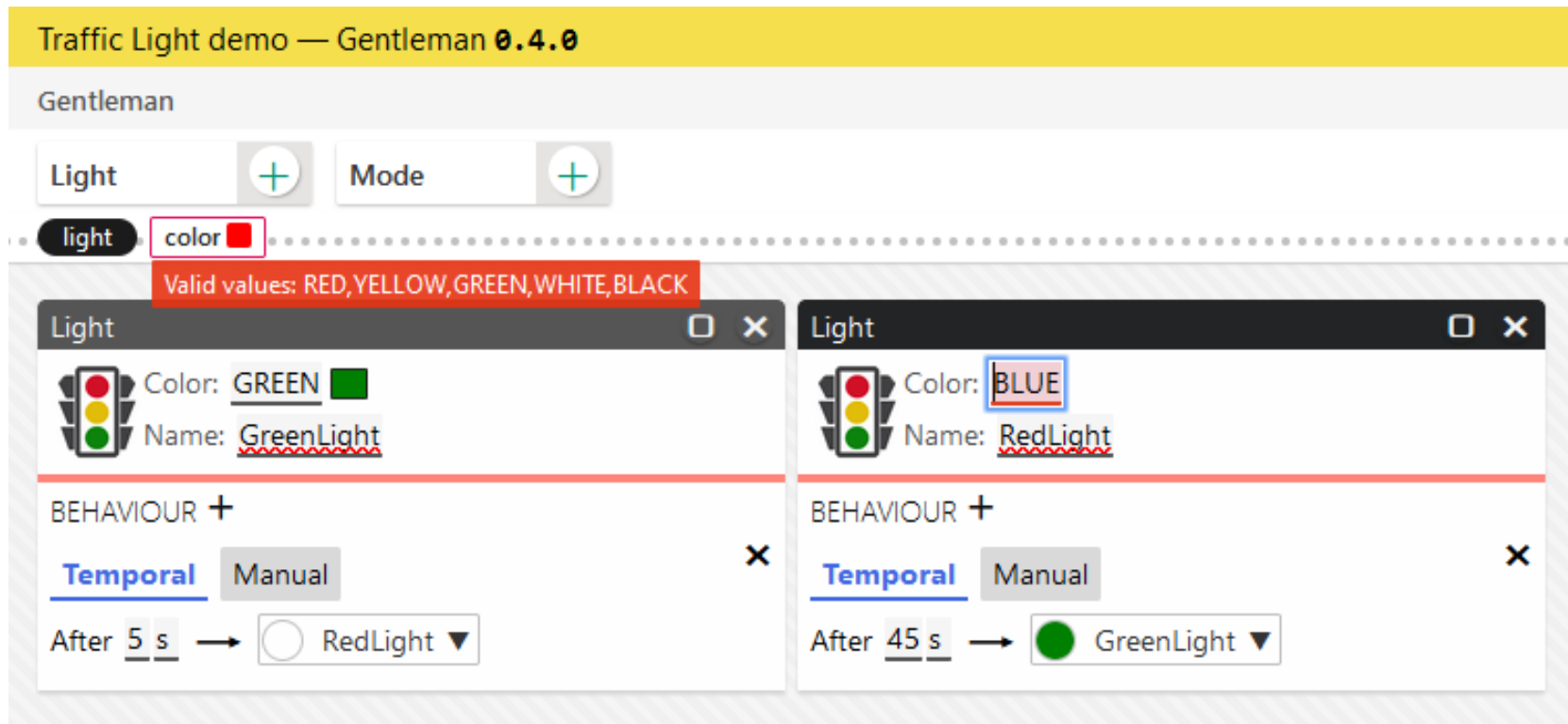
# PROJECTIONAL EDITORS – JETBRAINS MPS

## Parsing vs. projectional editing



Source: Mikhail Barash: Reflections on teaching JetBrains MPS within a university course

# PROJECTIONAL EDITORS – GENTLEMAN



Source: <https://geodes.iro.umontreal.ca/gentleman/demo/traffic-light/index.html>



THANK YOU FOR YOUR ATTENTION!