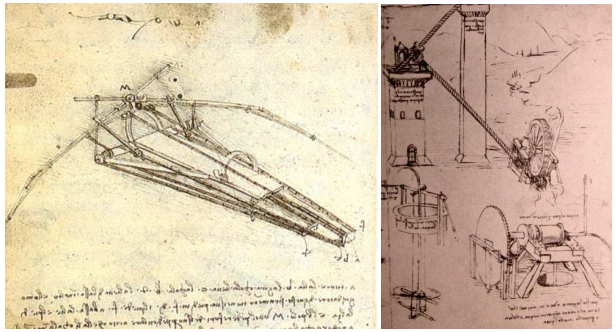
HY540 – Advanced Topics in Programming Language Development



Chapter 6 (two lectures)

Metaprogramming introduction (1/5)

- **Metaprograms –** *general definition*
  - Programs producing other programs
    - usually both in the same language
  - Functions defined in metaprograms, commonly referred as *metafunctions*
    - this distinction may be only conceptual or may be semantically imposed

Metaprogramming introduction (2/5)

- **Metafunctions –** *technical definition*
  - They accept and return program fragments in a form that can be easily manipulated:
    - ☒ source text — *impractical*
    - ☒ intermediate or target code — *low-level*
    - ✓ abstract syntax tree (AST) — *good*
  - Depending on the language, they may be invoked during compilation or execution to:
    - modify or introduce code fragments
    - such fragments become an integral part of the compiled or executing program

Metaprogramming introduction (3/5)

- *More definitions*
  - ***metafunction invocation*** **=** *rewriting* the function invocation expression at a *call site* with the code it actually generates
    - *metafunction f() { gen "return a*x+b;"; }*
    - *function g(a,x,b) { !f(); }* ⇒ *function g(a,x,b) { return a*x+b; }*
    - assume *!f()* to mean *insert f() output in its place*
    - thus rewriting the original code
  - ***metaprogram*** = a program that encompasses invocations of metafunctions
  - If the produced code is an independent module, thus no nee3d to rewrite the call site, the reflection mechanism may suffice
    - need compiler and the loader as library modules

Metaprogramming introduction (4/5)

- *Explanations*
  - Metafunctions may have arguments as well. When an argument is a source code unit it is translated to its respective abstract syntax tree (AST) representation.
    - Let *metafunction AddDesignbyContract(f){…}*
    - Let code *C = 'method f() { do something here }'*
    - Then the call to *AddDesignByContract(C)* produces the source code *method f() {assert pre_f(); do something here assert post_f(); }*
  - Thus, metafunctions may be designed as functions transforming / enriching / filtering units of code
    - the form of code rewriting in this manner is not restricted
    - e.g., could add locking calls to a normal functions for thread enabling or extra diagnostic code for debugging purposes

Metaprogramming introduction (5/5)

- *Special case*
  - Generics or genericity concern a form of type-safe metaprogramming with three important restrictions:
    - arguments to metafunctions must be previously defined *types*
      - e.g. *list[T]* or *list[list[T]]*
    - code generation is restricted to entire classes and functions
      - e.g. *generic class {…}* or *generic function(…){…}*
    - no free code generation is allowed, but the defined generic code is directly copied or invoked at the call site upon compilation
      - e.g. *genericfunction add[T](T x, Ty) { return x+y; }*
  - Because of these severe restrictions we separately refer to this type-safe form of metaprogramming as *generic programming*

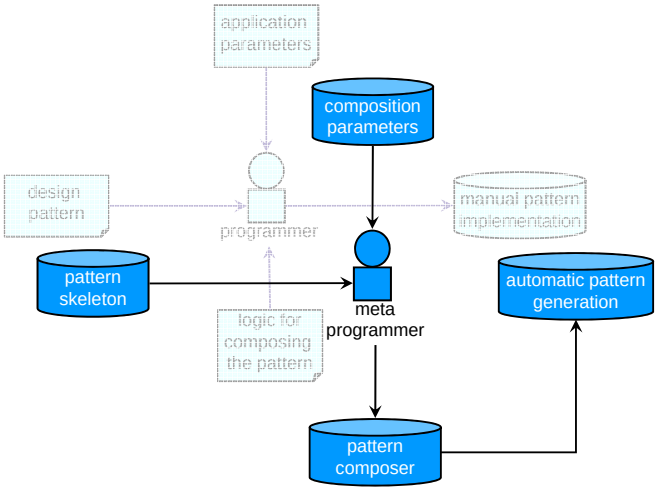
What metaprogramming solves (1/4)

- ***Reusability (1/2)***
  - The ability to reuse depends on how languages enable to express commonly *recurring code* directly into the language as *reusable units*
  - Currently, reuse is mainly instantiated in languages by the following approaches
    - *Procedure-based reuse*
      - functions, generic functions, polymorphic functions
    - *Object-based reuse*
      - classes, generic classes, object protocols

What metaprogramming solves (2/4)

- ***Reusability (2/2)***
  - Today it is acknowledged that reuse is far more critical at the *macroscopic scale*
    - reapplying proven design practices
  - The previous relates to *design patterns*
    - common solutions to recurring software problems
    - the solution cannot be reused as a source fragment
    - but should be manually adapted and applied by reimplementation

What metaprogramming solves (3/4)



What metaprogramming solves (4/4)

- *Example*
  - Assume we wish to add diagnostic messages upon entering and exiting functions automatically

```
function foo(…) {…};
AddDiagnosticChecks(      ←With a metafunction call like this
        foo,
        < printf("Entering 'foo'\n"); >,
        < printf("Exiting 'foo'\n");  >
);
function foo(…) {         ← We could get that
        printf("Entering 'foo'\n");
        …
        printf("Exiting 'foo'\n");
}
```

```
metafunction AddDiagnosticChecks (func, on_enter, on_exit) {
   func.body.AddFirst(on_enter);
   func.body.AddAtAllExitPoints(on_exit);
}
```


Roadmap

- Text units and macros
- ASTs and metafunctions
- Meta compiler architecture
- Multistage languages and staging
- Aspect-oriented programming

Text units and macros (1/4)

- In some languages it is the earliest known technique used as a limited form of metaprogramming
  - inlining text at local context directly
  - some form of text (pre)processing is possible
  - source code is treated as text with no capability to interpret structure (i.e., no AST is visible)
- Some languages tend to practically exaggerate the use of the macro processor for metaprogramming reasons
  - the syndrome is very simple
  - if a language offers only generic programming features
  - then its preprocessor will likely be used to support all cases of code generation
  - where emitted code units need to be linked in a local context

Text units and macros (2/4)

```
#define GEN_ATTRIBUTE(id,type)                          \
          void Set_##id (const type& _) {  id = _; }    \
          const type& Get_##id (void) const { return id; }
class Point3d {
private:
          double x, y, z;
public:
          GEN_ATTRIBUTE(x,double)
          GEN_ATTRIBUTE(y,double)
          GEN_ATTRIBUTE(z,double)
};
```

Common wisdom in C/C++ community: code-generating code is a macro whose parameters represent either partial source-code units, types or names.

- If this specific image of *`Point3d`* class is exactly what you wanted, no generic code could serve your needs
- Clearly, macros and their relevant processing are outside the language constructs (third party tool)
- In fact, it is well defined in the language that macro processing, called preprocessing, is a stage preceding program compilation
- Additionally, macros have no type checking meaning any error will simply appear at the point of use after inlining

Text units and macros (3/4)

- Macro processing continues to be a valuable tool in such languages with the absence of metaprogramming
- And will still be, but it is surprising that the current preprocessor features lay practically in the “stone age”
- Imagine functional-style and interpreter-like features such as (the list is indicative):
  - `#iteration(n,unit),`
  - `#counter`
  - `#break`
  - `#arg(i),` `#numargs`
  - `#condition(cond, ifTrue, ifFalse)`
  - `#eval(unit_with_other_pp_commands)`
- Their implementation is trivial, but the capabilities of the C preprocessor remain so primitive even after two decades of inclusion in the C++ language
  - *although libraries like Boost emulate these features with advanced preprocessor tricks*

Text units and macros (4/4)

- Whatever the macro processor functionality, it tends to be insufficient for metaprogramming
- The reason is that we cannot define code to inspect the internals of code supplied as an argument
  - We may wish to inject some code at specific points of an input source code unit
  - The latter to be possible with text processing requires build an entire parser as part of the meta code
  - Which, besides from being overkill, is likely impossible in the macro language
- Intuitively one would like to have some sort of AST representation to manipulate code either for iteration purposes (*read*) or for editing (*writing*)

ASTs and metafunctions (1/10)

- When source code is supplied as an argument to a metafunction it has to be in a form allowing the meta code perform some reasoning on it
- A suitable form is an AST, in practice it can be very close to a ST
- Normally, AST editing is to be performed by the meta code which is invoked at compile time, so the respective set of AST manipulation library functions is usually linked only with meta programs
- The outcome of a meta program, being a program, will be compiled only after inlining

ASTs and metafunctions (2/10)

- To support meta code the following built-in metafunctions are required at compile time
  - *@syntax('code')* produces and returns the AST of *code*
  - ***@escape(expr)*** preserves *expr* by carrying its value when in *syntax*
  - *@inline(code)* inlines code from *code* being an AST expression
  - ***@run(stmt)*** executes *stmt* during compilation (also ***@define***)
  - *@error(msg)* issues a compilation error

```
//lex
".<"        { return OPEN_QQ; }
">."        { return CLOSE_QQ; }
"@syntax"   { return SYNTAX_TOKEN; }
//yacc
Syntax: SYNTAX_TOKEN '(' OPEN_QQ Stmt CLOSE_QQ ')'  { ... }
Syntax: OPEN_QQ Stmt CLOSE_QQ   { ... }

@inline(
    @inline(
        .< .< printf("hello"); >. >.
    )
);
=>
@inline(.< printf("hello"); >.);
=>
printf("hello");
```


ASTs and metafunctions (3/10)

- As shown, we allow built-in metafunctions to be invoked directly by their name
- Also any metafunction may appear as part of the normal source code. This allows the metacode to also produce extra metacode.
  - We have seen it in the previous example where we had the expression `'@syntax('int y = 20')'` being supplied as argument to ***@syntax*** itself
- In general, every ***@syntax*** expression lifts its source code argument to meta code, while to revert it to normal code one has to explicitly inline it.
  - `@inline(@syntax('code'))` ≡ ***code***
  - `@inline(@syntax('print("hello,world");'))` ≡ ***print("hello,world");***

ASTs and metafunctions (4/10)

- One can generalize the previous as follows:
  - `@inline(N @syntax(N 'code')N)N` ≡ ***code***
    - ******For simplicity many single quotes omitted***
- The meaning is that we would have to compile (inline) once more the outcome of a metafunction if it happens to return the AST of a meta expression
  - Think of it as the general case where *metaprogramming is also applied in implementing the metacode*
- Or think of it as a macro which generates code including macro definitions or preprocessor directives
  - *You would need to explicitly perform an extra preprocessing stage to expand such generated macros*

ASTs and metafunctions (5/10)

- Now we will change to a special syntax for the built-in metafunctions
  - Those are either called *meta tags*, *staging tags* or *quasi quotes*
- While we forbid non-metafunctions be invoked from metafunctions and vice versa
- We use the staging tags of *MetaOcaml*
  - *Meta Ocaml* (Objective *Caml* (*Categorical abstract machine language*)))
  - `.<` *expr* `>.` ≡ *@syntax('code')* ≡ *shift to meta level*
  - *.~ expr* ≡ *@escape(expr)* ≡ *preserve expression*
  - *.! expr* ≡ *@inline(code)* ≡ *compile meta level code*
- Meta tags appearing in a source program are called *meta annotations (staging annotations)*

ASTs and metafunctions (6/10)

```
@power(a, N) {
        if (N == 1)
                return a;
        else
                return .<.~a * .~@power(a, N-1)>.;
}
a = .!@power(.<x>., 4);
a = @power(.<x>., 4); equivalently to previous by implying .!
.!@power(var[x], 4)                                          =>
.!.<var[x] * @power(var[x], 3)>.                             =>
.!.<var[x] * .<var[x] * @power(var[x], 2)>.>.                =>
.!.<var[x] * .<var[x] * .<var[x] * @power(var[x], 1)>.>.>.   =>
.!.<var[x] * .<var[x] * .<var[x] * var[x] >.>.>.             =>
.!.<var[x] * .<var[x] * mul[var[x], var[x]] >.>.             =>
.!.<var[x] * mul[var[x], mul[var[x], var[x]] >.              =>
.!mul[var[x], mul[var[x], mul[var[x], var[x]]]               =>
x*(x*(x*x))                                                  =>
x*x*x*x
```


ASTs and metafunctions (7/10)

```
function power (x,y) { normal implementation (non metafunction) }
@power (x, N) {
        if (not @isconstant(N)) invoke non-optimized version
           return .<power(.~x, .~N)>.;
        else
        if (not @isintegerconst(N))
           @error("Non integer constant supplied to 'power'");
        else generate inline code for evaluation (like 'loop unrolling')
        if (N is constant value 1)
            return .~x;
        else
            return .<.~x * .~@power(x, N-1)>.;
}
.!@power(.<x>., .<y>.);                  =>
.!call[power, args[var[x], var[y]]]      => equivalent
power(x,y)
```

- Metafunctions may be also used to perform (actually to program) some compile-time optimizations that cannot be normally done by optimizers.
- For instance, in this example the optimization applied depends on the semantics of the *power* function, something that cannot be known by an optimizer.

ASTs and metafunctions (8/10)

- In some cases optimization-specific metacode may be written in languages with genericity and some degree of pattern matching, like C++ templates

```
template
<class Tbase, class Texponent> struct power {
    double operator()(Tbase x, Texponent y) const
        { return pow(x,y); }
};

template <class Tbase>
struct power<Tbase, unsigned int> {
    double operator()(Tbase x, unsigned int y)
        { for (Tbase r = x; --y; r *= x); return r; }
};
```

Via partial template specialization = *compile-time type-pattern matching method of the language*

- But the language was not designed for full manipulation of ASTs at compile-time
  - for example, can't distinguish the compile-time const-value type (e.g. *const unsigned int N*) from the const type of a runtime value (*const unsigned int*)

ASTs and metafunctions (9/10)

- For practical reasons we introduce two extra meta functions normally not met in languages, thus not needed *per se* for metaprogramming
  - *.#expr* ≡ *unparse a meta expression (AST→text)*
  - *.@string_const* ≡ *parse a compile-time string constant to AST*
    - the *string_const* may represent any valid expression of the language, not only viable source code
- Their presence allows
  - extrapolate the source code outcome of a metaprogram
    - via .# meta tag - for metacode debugging
  - use string literals as code segments inside metaprograms
    - via .@ meta tag - for code assembly (think of it like macros)

Intermezzo

$$.!.@.\#.\langle expr \rangle. \equiv expr$$

$\underbrace{\quad}_{AST \leftarrow expr\ (lift)}$

$\underbrace{\quad}_{Text \leftarrow AST\ (unparse)}$

$\underbrace{\quad}_{AST \leftarrow Text\ (parse)}$

$\underbrace{\quad}_{execute\ AST\ (translate)}$

ASTs and metafunctions (10/10)

```
@function ClassPrefix (id, heritage) {
        return "class " + id + heritage + "{";
}
@function ClassSuffix (id) {
        return   id + "(const " + id + "&);"        +
                 id + "(void);"                     +
                 "virtual ~" + id + "();"           +
                 "};"
}
@function AddField (type, id) {
        return   "private:" + type + " " + id + ";"                          +
                 "public: const " + type + "& Get_" + id + "(void) const"    +
                 "{ return " + id + ";}"                                     +
                 "public: void Set_" + id + "(const" type + "& _)"           +
                 "{" + id + "=_;}";
}
@(PointClassCode = ClassPrefix("Point", "")        +
                   AddField("int", "x")            +
                   AddField("int", "y")            +
                   ClassSuffix("Point"));  Evaluate a (meta) expression at compile time
.!.@PointClassCode;  Notice that PointClassCode is a metaprogram variable, not a program variable
```

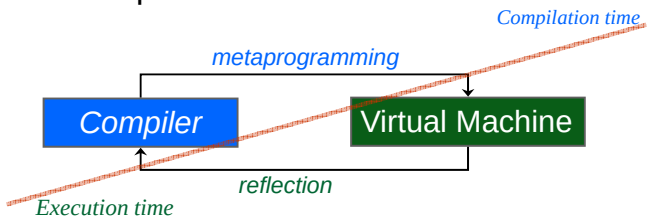
- The .@ and .# meta tags allow powerful text-code combination at compile-time in a way superior to typical macro systems.
- Additionally, the @ compile-time call operator is added to evaluate meta expressions in general.

Meta compiler architecture (1/7)

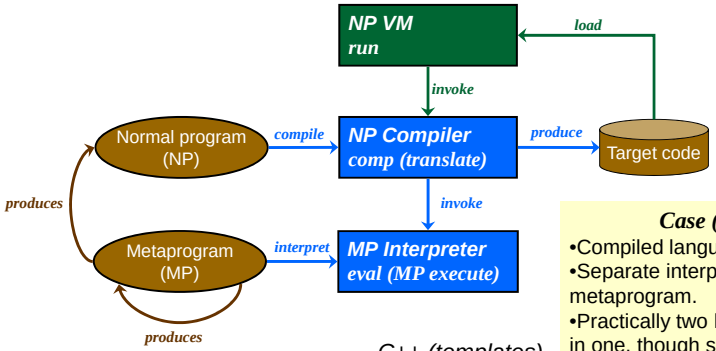
- In reflection we have seen that to support on-the-fly compilation of source code
  - the compiler should be made an integral part of the language runtime system (VM) implementation
- In meta programming to support execution of source code during compilation
  - the language runtime (VM) should be made an integral part of the compiler implementation

Meta compiler architecture (2/7)

- The previous introduces a sort of symmetry and can be seen as completeness in terms of the code manipulation features of the language
- However it introduces the issue of non-termination since the metaprogram may either hang or take a lot of time to complete



Meta compiler architecture (3/7)



Case (I)
- Compiled language.
- Separate interpreter for the metaprogram.
- Practically two languages in one, though separate.
- The VM is never invoked during compilation (no invocation loop).

C++ (templates), excluding the reflection stuff

Meta compiler architecture (4/7)

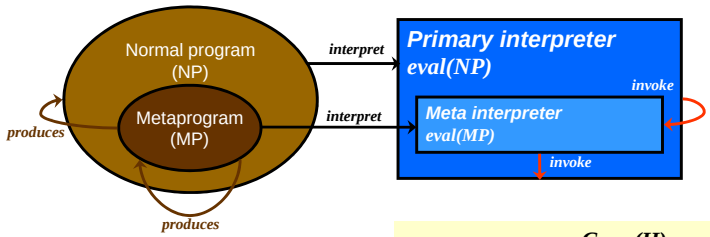
- C++ template example
  - Template definitions correspond to metafunction definitions
  - Template instantiations correspond to metafunction invocations

```
template<typename T, int N>   //declaration of template Array with params T, N
class Array {
   T values[N];
};

using IntArray10 = Array<int, 10>;   //instantiating Array with specific types
➔ class IntArray10 {                 //and values (e.g. int, 10) generates the
    int values[10];                  //normal code on the left
  };                                 //Template (MP) produces normal code (NP)

template<int N>
using DoubleArray = Array<double, N>;//instantiating Array with a
➔ template<int N>                    //parameterized type or value (e.g. N)
   class DoubleArray {               //generates the template code on the left
      double values[N];              //Template (MP) produces template (MP)
   };
```

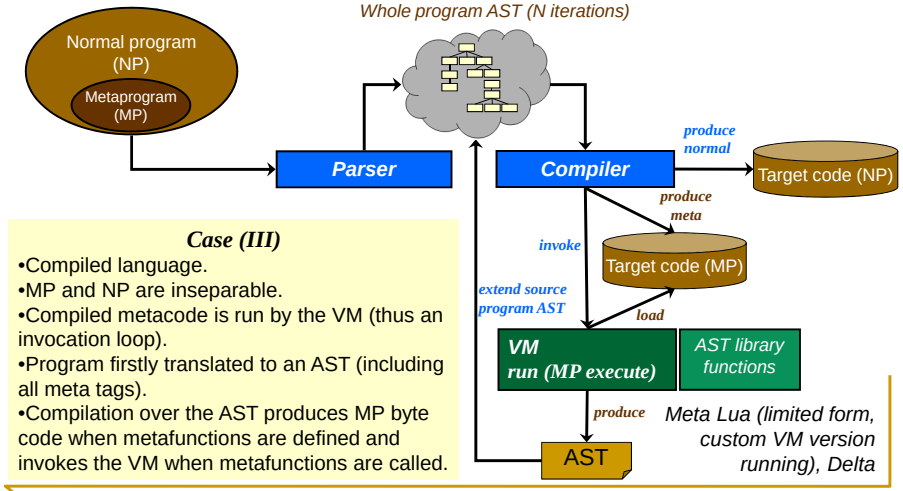

Meta compiler architecture (5/7)



Case (II)
- Interpreted language.
- MP interpreter just a part of the NP interpreter.
- Metaprogramming extensions seem inseparable from the main language.
- There is an invocation loop amongst the two interpreters.
- Remember that generating text source code and invoking *eval* is partially metaprogramming.

MetaOCaml, ML, PERL, all functional languages with MP extensions

Meta compiler architecture (6/7)



Case (III)
- Compiled language.
- MP and NP are inseparable.
- Compiled metacode is run by the VM (thus an invocation loop).
- Program firstly translated to an AST (including all meta tags).
- Compilation over the AST produces MP byte code when metafunctions are defined and invokes the VM when metafunctions are called.

Meta Lua (limited form, custom VM version running), Delta

Meta compiler architecture (7/7)

- Delta metaprogramming example
  - Metacode is specified through staging tags
    - *execute* - ***&code***
    - *inline* - ***!code***
  - The form *&function* corresponds to a metafunction definition
  - Calls within inline tags are metafunction invocations

```
&function func_generator(name, args , body) {    //metafunction definition
  return << function ~name (~args) { ~body; } >>;//that creates a function
}                                                 //based on the given input
//metafunction invocation generates normal code: MP produces NP
!(func_generator(<<id>>, <<x>>, <<return x;>>)); ➔function id(x){return x;}

&function metacode_generator(code) {  //metafunction definition that turns
  return <<&~code;>>;                 //<<stmt>> into <<&stmt>>, causing stmt
}                                     //to be executed at compile time
//metafunction invocation generates meta code: MP produces MP
!(metacode_generator(<<print("meta-code")>>)); ➔ &print("meta-code");
```


Multistage languages and staging (1/4)

- The presence of meta annotations in a program imply that it has a metaprogram which needs to be executed to generate the actual program
- In this sense, metaprograms can be seen as program generators, although it is common that metacode is mixed with normal program code, meaning there may be no isolated continuous-source metaprogram
- The execution of a metaprogram is a *compilation stage* that precedes the compilation of its outcome
- In general, if the output of metacode execution encompasses meta tags then an extra execution stage is always needed to produce further output

Multistage languages and staging (2/4)

- Multistage is a language enabling metacode to produce metacode (i.e. with staging annotations) and offering an operator for compile-time invocation of metacode
- Staging as such is a common technique for program generation beyond metaprogramming
  - parser generators prescribe compilation of grammar rules to parser code and then compilation of the produced code to machine code
  - since two distinct languages and tools are involved, we have multistage generation but not a multistage language
- *Terminology*
  - Staged program:
    - Conventional program + staging annotations
  - Stage metaprogram or just metaprogram
    - Program which outputs source code in a stage

Multistage languages and staging (3/4)

- Delta metaprogramming example revisited
  - There is actually no syntactic or semantic distinction between functions and metafunctions
  - *Execute* (***&code***) and *inline* (***!code***) staging tags define the boundaries between different stages and introduce stage nesting

```
&function generate(code, stage_nesting){       //stage 1 function definition
  local result = code;
  for (local i = 0; i < stage_nesting; ++i)
    result = <<&~result;>>;                    //turns <<stmt>> into <<&stmt>>
  return result;
}
//stage 1 function call generates stage 0 (i.e. normal) code: MP produces NP
!(generate(<<print("Normal code")>>, 0)); ➔ print("Normal code");
//stage 1 function call generates stage 1 code: MP produces MP
!(generate(<<print("Meta-code")>>, 1)); ➔ &print("Meta-code");
//stage 1 function call generates stage 2 code: MP produces higher order MP
!(generate(<<print("Meta-meta-code")>>, 2)); ➔ &&print("Meta-meta-code");
```

Multistage languages and staging (4/4)

- Apart from the evident challenges for writing *$n^{>2}$-stage metaprograms*, tool support is also demanding
- Both metacode and generated code require
  - Error reporting
  - Source-level debugging
  - Source editing
  - Source browsing
- Metaprograms may also require a full scale build system, with build flags and dependencies

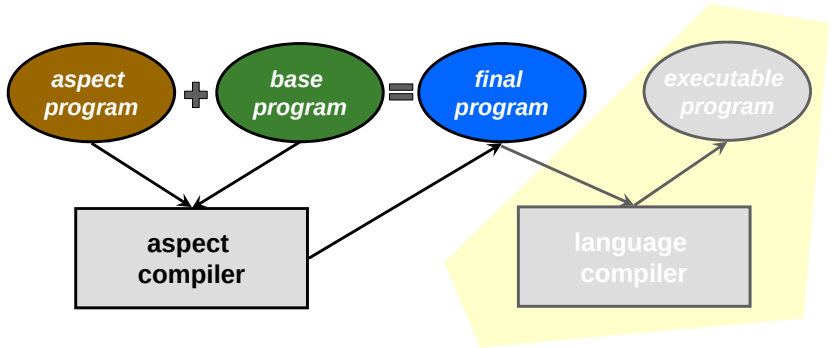
Aspect-oriented programming (AOP) (1/8)

- *The idea*
  - *Question:* What is the actual programming problem that AOP aims to solve?
  - *Answer:* Need to globally perform update actions introducing concerns applying to multiple points at the source code that would mandate deep refactoring to be handled as a new abstraction
  - *Example:* You need to introduce diagnostic logging for the invocation of specific methods of specific classes
  - *Solution*: Describe a logging aspect which defines the classes and methods to match and the logging statements to inject
  - *Avoids*: To manually introduce logging invocations or introduce something like a *Loggable* abstraction (superclass), especially if logging is a transient requirement

Aspect-oriented programming (AOP) (2/8)

- *Methodologically*
  - it is a way to globally apply well-defined transformations on a program using some sort of code pattern matching (query)
  - for example, *in <every method> matching <this criterion> add <this code snippet> at <this method point>*
- *Theoretically*
  - it allows to make programming statements of the form: ***in program P, whenever condition C arises, perform action A***
  - such statements form an *aspect program* while the program to be transformed is called the *base program*
- *Technically*
  - It is a generation technique with a single stage where the *aspect compiler* transforms a *base program* according to the definitions of an *aspect program*
- Aspect J, Aspect C++, Aspect Lua

Aspect-oriented programming (AOP) (3/8)



•The base program need not be in a source code format but in some compiled form (like byte code). In this case the final program is ready for execution.

Aspect-oriented programming (AOP) (4/8)

- The following concerns arise when designing an AOP system supporting statements of the form ***in program P, whenever condition C arises, perform action A***
  - ***Quantification*** What kinds of *C* conditions (matching criteria) can we specify
  - ***Interface*** What is the interface of the transformation actions *A* (how do they interact with base programs and each other)
  - ***Weaving*** How will the system arrange to intermix the execution of the base actions (statements / code) of program ***P*** with the actions ***A***

Aspect-oriented programming (AOP) (5/8)

- Lets study the *quantification* characteristic of an aspect language
  - Over what we can quantify (i.e. set conditions or matching criteria)?
  - Broadly, we may quantify either on the *static structure* of the system (*source conditions*) or over its *dynamic behavior* (*runtime conditions*)
- ***Static quantification***
  - ***Black box***: over the public interface of components
  - ***White box***: over the parsed code structure of components
- ***Dynamic quantification***
  - Over runtime conditions and events (exceptions, invocation, history patterns)

Aspect-oriented programming (AOP) (6/8)

- *Terminology*
  - *aspect*
    - the base program *transformation specifications*
  - *advice*
    - the extra *behavior added* to the base program by an aspect
  - *pointcut*
    - the quantification (*query / conditions / matching criteria*)
  - *join points*
    - points of *code that will match* a pointcut
  - *concern*
    - *the design* concept reflected by an advice

Aspect-oriented programming (AOP) (7/8)

- Logging example in AspectJ

```
aspect Logging {                                  //aspect definition
 pointcut method() : execution(* *(..)); //pointcut: execution of all methods
 before() : method() {//before advice: code to execute before method execution
   System.out.println("Entering " + thisJoinPoint.getSignature().toString());
 } //thisJoinPoint is an object with information about the matched method
 after() : method() {//after advice: code to execute after method execution
   System.out.println("Leaving " + thisJoinPoint.getSignature().toString());
 }
}
public class Test {
 public void f() { System.out.println("Inside Test.f()"); } //joinpoint match:
 ➔ public void f() {                                       //transforms the
     System.out.println("Entering Test.f()");              //initial code as
     System.out.println("Inside Test.f()");                //shown on the left
     System.out.println("Leaving Test.f()");
   }
}
```

Aspect-oriented programming (AOP) (8/8)

- When comparing metaprogramming to AOP it is clear that the two have different origins
  - metaprogramming upgrades programming to a higher-order design activity
    - defining metafunctions accepting as parameters program units and producing as output subprograms
  - AOP programming turns disciplined extensions to an program transformation specification activity
    - defining when and how extensions are to be applied
- *Metaprogramming can be applied for implementing aspects with static white box quantification and virtually any form of program transformation*