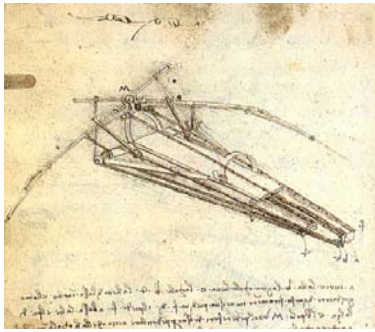
HY540 – Advanced Topics in Programming Language Development




Chapter 6 (two lectures)

Meta programming (1/3)

- *Definitions*
  - ***Metafunction***: a function that can produce source code at compile-time *in the context* of its invocation (*or a function producing other functions*)
    - Let *metafunction f() { gen "return a*x+b;"; }*
    - Then *function g(a,x,b) { f(); }* ≡ *function g(a,x,b) { return a*x+b; }*
  - ***Metaprogram***: a program that encompasses metafunctions
  - One may view a metaprogram as a program that is capable to generate other programs
  - If the produced code is an simply an independent module not linked to some local context then the language reflection mechanism may well suffice
    - *having the compiler and the loader as library functions*

Meta programming (2/3)

- *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).
    - 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 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

Meta programming (3/3)

- *Specialization*
  - Generics or genericity concern a form of type-safe metaprogramming with three important restrictions:
    - arguments to metafunction 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 define 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*

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 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

Text units and macros (3/4)

- Macro processing continues to be a valuable tool in such languages with the absence of meta-programming
- An will still be, but it is surprising that the 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)*
- Their implementation is trivial, but the capabilities of the C preprocessor remain so primitive even after two decades of inclusion in the C++ language

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 library functions is linked only to meta programs
- The operation through which the outcome of a meta program, being a program, is normally compiled is called *run or comp* and is the normal static compilation
  - we may ignore the operator if clear that the outcome is normal code that needs to be compiled

ASTs and metafunctions (2/10)

- To support meta code the following built-in metafunctions are required at compile time
  - *@syntax('code')* produces the AST for *code*
  - *@meta(expr)* preserves *expr* as a meta expression
  - *@comp(code)* compiles meta code *code* *(optional)*
  - *@error(msg)* issues a compilation error
- For example, here is an *identity* metafunction

```
@identity(ast) { @comp(ast); }  we define a new metafunction
      #define identity(x) x
@identity(@syntax('int x = 10;'));
      @comp AST for 'int x = 10;'
@identity(@identity(@syntax('@syntax('int y = 20')')));
      identity(@comp AST for '@syntax('int y = 20')
      @comp AST for 'int y = 20'
```


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 compile it.
  - `@comp(@syntax('`*code*`'))` ≡ *code*
  - `@comp(@syntax('`*print("hello,world");*`'))` ≡ *print("hello,world");*

ASTs and metafunctions (4/10)

- One can generalize the previous as follows:
  - **@comp(ᴺ @syntax(ᴺ '*code*')ᴺ)ᴺ** ≡ *code*
    - ****For simplicity many single quotes omitted***
- The meaning is that we would have to compile 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* (*O*bjective *Caml* (*C*ategorical *a*bstract *m*achine *l*anguage)))
  - **.<** *expr* **>.** ≡ *@syntax('code')* ≡ *shift to meta level*
  - **.~** *expr* ≡ *@meta(expr)* ≡ *preserve meta expression*
  - .! *expr* ≡ *@comp(code)* ≡ *compile meta level code*
- Meta tags appearing in a source program are called *meta annotations (staging annotations)*

ASTs and metafunctions (6/10)

```
@power(x, N) {
        if (N == 1)
                return .~x;
        else
                return .<.~x * .~@power(x, 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 unrollin
        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
  - **.#***var* ≡ *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

.!.@.#.<*expr***>.** ≡ *expr*

AST← expr (lift)

Text ← AST (unparse)

AST ← Text (parse)

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
```

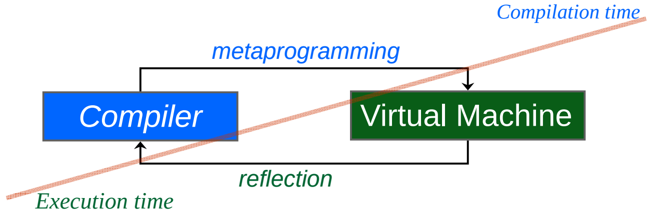
- The *.@* and *.#* meta tags allow powerful text-code combination at compile-time in a way superior to typical macro systems.
- Additionally, the *@* compile-time call operator is added to evaluate meta expressions in general.

Meta compiler architecture (1/5)

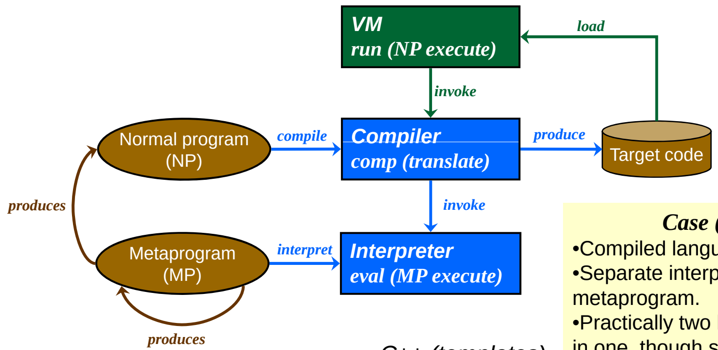
- In reflection we have seen that to support on-the-fly compilation of source code
  - the compiler should be made an integral part of the language runtime system (VM) implementation
- In meta programming to support execution of source code during compilation
  - the language runtime (VM) should be made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

Meta compiler architecture (2/5)

- 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 (3/5)

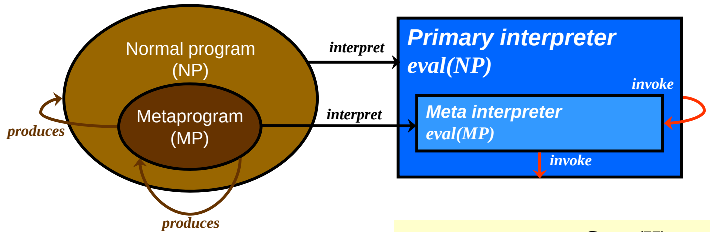


C++ (templates), excluding the reflection stuff

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).

Meta compiler architecture (4/5)

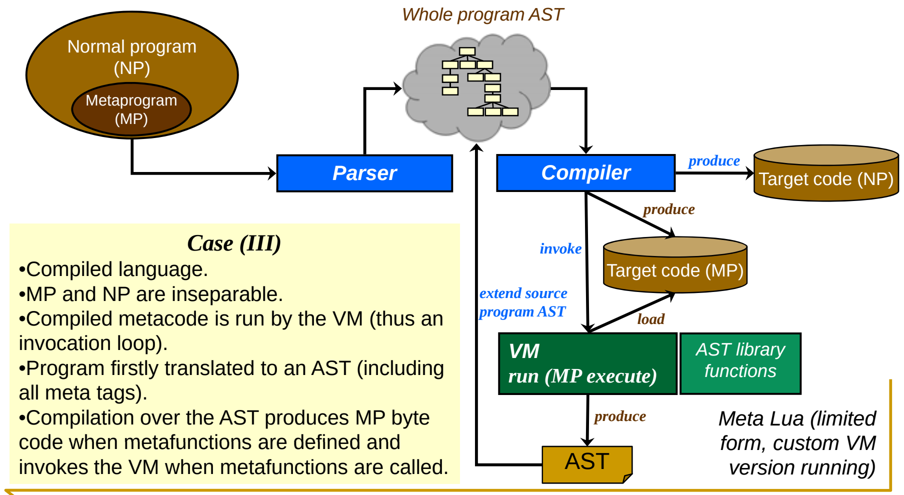


OCaml, ML, PERL, all functional languages with MP extensions

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.

Meta compiler architecture (5/5)



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.

Multistage languages and staging (1/3)

- The presence of meta annotations in a program imply that it is 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 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 (2/3)

- 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
- As we will review at the end, apart from the evident challenges for writing *$n^{>2}$-stage metaprograms*, tool support is also demanding

Multistage languages and staging (3/3)

- *Staged program*
  - Conventional program + staging annotations
- *Meta program*
  - Conventional program + meta annotations

Aspect-oriented programming (AOP) (1/7)

- *Methodologically*
  - it is a way to globally apply well-defined transformations on a program using some sort of code pattern matching (query)
  - for example, *add to <every method> matching <this criterion> <this code snippet> at <this 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 transformed program 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*

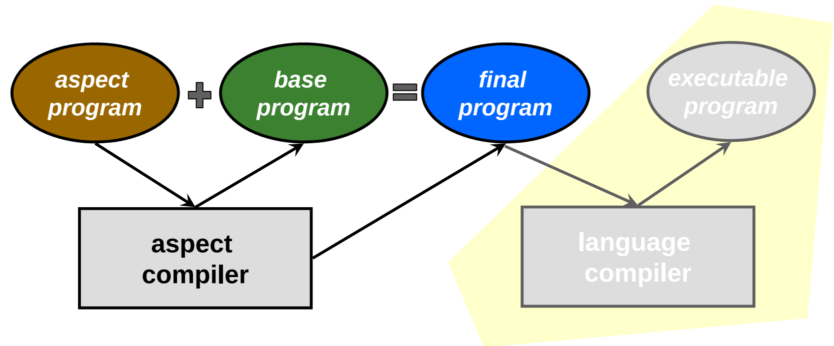
Aspect-oriented programming (AOP) (2/7)

- 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) (3/7)

- *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) (4/7)



•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) (5/7)

- Lets study the characteristics of an aspect language starting from *quantification*
  - 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) (6/7)

- *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) (7/7)

- 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 an form transformation*