11. Program Transformation

Oscar Nierstrasz
Roadmap

- Program Transformation
- Refactoring
- Aspect-Oriented Programming
Links

> **Program Transformation:**
  — http://swerl.tudelft.nl/bin/view/Pt
  — http://www.program-transformation.org/

> **Spoofax/Stratego:**
  — http://www.metaborg.org/

> **TXL:**
  — http://www.txl.ca/

> **Refactoring:**
  — http://recoder.sourceforge.net/wiki/
  — http://www.refactory.com/RefactoringBrowser/

> **AOP:**
  — http://www.eclipse.org/aspectj/
Roadmap

> Program Transformation
  — Introduction
  — Stratego/XT
  — TXL
> Refactoring
> Aspect-Oriented Programming

Thanks to Eelco Visser and Martin Bravenboer for their kind permission to reuse and adapt selected material from their Program Transformation course. http://swerv.tudelft.nl/bin/view/Pt
What is “program transformation”?  

> **Program Transformation** is the process of transforming one program to another.

> Near synonyms:
  — Metaprogramming
  — Generative programming
  — Program synthesis
  — Program refinement
  — Program calculation
Metaprogramming refers more generally to programs that manipulate other programs.

Generative programming refers to the generation of programs, for example, by composing templates in C++.

Program synthesis concerns the generation of programs from high-level specifications.

Program refinement refers to the stepwise transformation of high-level programs to lower-level executable ones.

Program calculation is the derivation of programs by manipulating formulas.
Applications of program transformation

> Translation
  - Migration
  - *Synthesis*
    - Refinement
    - Compilation
  - *Reverse Engineering*
    - Decompilation
    - Architecture Extraction
    - Visualization
  - *Program Analysis*
    - Control flow
    - Data flow
• **Migration**: transforming code from an old language to a new one.
• **Synthesis**: generating executable programs from higher-level specifications.
• **Reverse Engineering**: extracting higher-level representations from low-level code.
• **Program Analysis**: extracting information from code in order to reason about its properties.
function fact(n : int) : int =
    if n < 1 then 1
    else (n * fact(n - 1))

Tiger

```
fact:  subu $sp, $sp, 20
        sw  $fp, 8($sp)
        addiu $fp, $sp, 20
        sw  $s2, -8($fp)
        sw  $ra, -4($fp)
        sw  $a0, 0($fp)
        move $s2, $a1
        li   $t0, 1
        bge  $s2, $t0, c_0
        li   $v0, 1
        b    d_0

    c_0:  lw   $a0, ($fp)
          li   $t0, 1
          subu $a1, $s2, $t0
          jal  fact_a_0
          mul  $v0, $s2, $v0

    d_0:  lw   $s2, -8($fp)
          lw   $ra, -4($fp)
          lw   $fp, 8($sp)
          addiu $sp, $sp, 20
          jr    $ra
```

MIPS

Tiger is an experiment in using program transformation to compile a high-level programming language (Tiger) to assembly code.

See: http://strategoxt.org/Tiger/WebHome
Translation — migration from procedural to OO

Tiger

```tiger
type tree = {key: int, children: treelist}

type treelist = {hd: tree, tl: treelist}

function treeSize(t : tree) : int =
    if t = nil then 0 else 1 + listSize(t.children)

function listSize(ts : treelist) =
    if ts = nil then 0 else listSize(t.tl)
```

Java

```java
class Tree {
    Int key;
    TreeList children;
    public Int size() {
        return 1 + children.size
    }
}

class TreeList { ... }
```

Rephrasing — desugaring regular expressions

Exp := Id
| Id "(" {Exp ","}*)"
| Exp "+" Exp
| ...

⇒

Exp := Id
| Id "(" Exps ")"
| Exp "+" Exp
| ...

Exps :=
| Expp

Expp := Exp
| Expp "," Exp

“Syntactic sugar” refers to syntax that is convenient for the programmer, but is not strictly needed as the host language already supports a more verbose way of doing the same thing. Autoboxing in Java is an example of syntactic sugar, since:

```java
Integer n = 1;
```

is automatically rewritten to:

```java
Integer n = new Integer(1);
```

The example in the previous slide shows how Extended BNF (EBNF) with a Kleene closure operator (*) can be desugared by program transformation to an equivalent BNF without the extension.
Rephrasing — partial evaluation

function power(x : int, n : int) : int =
    if n = 0 then 1
    else if even(n) then square(power(x, n/2))
    else (x * power(x, n - 1))

Tiger

\[ n = 5 \]

function power5(x : int) : int =
    x * square(square(square(x)))

Partial evaluation is a technique to rewrite programs if some of their parameters are known in advance. In this example, the second parameter \( n \) to the function \( \text{power}(\ ) \) is known, so we can generate an equivalent function that only takes the first argument, by partially evaluating the code of the original function.

If \( n=5 \), then the original body can be rewritten by partially evaluating it to:

\[
x \times \text{power}(x, 4)
\]

Another step allows us to partially evaluate \( \text{power}(x, 4) \) yielding:

\[
x \times \text{square}(\text{power}(x, 2))
\]

A final partial evaluation step yields:

\[
x \times \text{square}(\text{square}(x))
\]
Transformation pipeline

http://losser.st-lab.cs.uu.nl/~mbravenb/PT05-Infrastructure.pdf
This general scheme applies to Stratego, TXL and various other systems. Transformation systems and languages may support or automate different parts of this pipeline.

If the source language is fixed, then a fixed parser and pretty-printer may be used.

If the source and target languages are arbitrary, then there should be support to specify grammars and automatically generate parsers and pretty-printers.
Roadmap

> Program Transformation
  — Introduction
  — Stratego/XT
  — TXL

> Refactoring

> Aspect-Oriented Programming
Stratego/XT (aka Spoofax Language Workbench)

> **Stratego**
  - A language for specifying program transformations
    - term rewriting rules
    - programmable rewriting strategies
    - pattern-matching against syntax of object language
    - context-sensitive transformations

> **XT**
  - A collection of transformation tools
    - parser and pretty printer generators
    - grammar engineering tools

http://strategoxt.org/
The parser and a basic pretty-printer 100% generated.
Language-specific support for transformations are generated.
Parsing

Rules translate terms to terms

Stratego parses any context-free language using Scannerless Generalized LR Parsing

```module Exp
exports
  context-free start-symbols Exp
sorts Id IntConst Exp

lexical syntax
  [\ \t\n]  -> LAYOUT
  [a-zA-Z]+  -> Id
  [0-9]+     -> IntConst

context-free syntax
  Id        -> Exp {cons("Var")}
  IntConst  -> Exp {cons("Int")}
  "("  Exp  ")" -> Exp {bracket}
  Exp "*"  Exp -> Exp {left, cons("Mul")}
  Exp "/"  Exp -> Exp {left, cons("Div")}
  Exp "%"  Exp -> Exp {left, cons("Mod")}
  Exp "+"  Exp -> Exp {left, cons("Plus")}
  Exp "-"  Exp -> Exp {left, cons("Minus")}

context-free priorities
  {left:
    Exp "*"  Exp -> Exp
    Exp "/"  Exp -> Exp
    Exp "%"  Exp -> Exp
  }
  > {left:
    Exp "+"  Exp -> Exp
    Exp "-"  Exp -> Exp
  }
```
This example shows a simple expression language, with lexical rules for tokens, a simple (ambiguous) grammar for the syntax, and priority rules for the operators to aid in disambiguation. The grammar rules also contain actions to produce syntax tree terms. Generalized LR (GLR) parsing essentially does a parallel, breadth-first LR parse to handle ambiguity.

https://en.wikipedia.org/wiki/GLR_parser

See the Makefile for the steps needed to run the example.

git://scg.unibe.ch/lectures-cc-examples
   subfolder: cc-Stratego

Caveat: the workbench is rather complex and not so easy to install and use.
Testing

testsuite Exp
topsort Exp

test egl parse
"1 + 2 * (3 + 4) * 3 - 1" ->
Minus(
    Plus(
        Int("1")
    , Mul(
        Mul(Int("2"), Plus(Int("3"), Int("4"))))
    , Int("3")
    )
), Int("1")
)
This file specifies a test that the given input string, when parsed, will result in the term that follows.
Running tests

pack-sdf -i Exp.sdf -o Exp.def
  including ./Exp.sdf

sdf2table -i Exp.def -o Exp.tbl -m Exp
SdfChecker: error: Main module not defined
  --- Main

parse-unit -i Exp.testsuite -p Exp.tbl

executing testsuite Exp with 1 tests

* OK   : test 1 (eg1 parse)

results testsuite Exp
successes : 1
failures  : 0
We need to perform several steps to run the tests. First the SDF modules (including any imported ones) are “packed” into a single definition. Next, the definitions are analyzed and a parse table is generated. Finally, the tests are run.
Interpretation example

```plaintext
module ExpEval

imports libstratego-lib
imports Exp

rules
  convert : Int(x) -> <string-to-int>(x)
  eval : Plus(m,n) -> <add>(m,n)
  eval : Minus(m,n) -> <subt>(m,n)
  eval : Mul(m,n) -> <mul>(m,n)
  eval : Div(m,n) -> <div>(m,n)
  eval : Mod(m,n) -> <mod>(m,n)

strategies
  main = io-wrap(innermost(convert <+ eval))
```

Stratego separates the specification of rules (transformations) from strategies (traversals). In principle, both are reusable.
In this example we specify a separate set of rules to transform syntactic terms of the parse tree to evaluated expressions. In this case the transformation rules use built-in arithmetic functions.

In general, term rewriting can be performed using a variety of strategies (top-down, bottom-up, etc.). Stratego separates the definition of the rules from the strategies (unlike Prolog, in which rules are strictly applied in the order they appear).

For this example, arithmetic functions can only be applied to fully evaluated expressions (i.e., numbers), hence the strategy to use must work from the leaves inwards (“innermost”). Furthermore, we must convert integers before applying operations, so we specify (convert <+ eval).
A *strategy* determines how a set of rewrite rules will be used to traverse and transform a term.

- innermost
- top down
- bottom up
- repeat
- …
Running the transformation

```
sdf2rtg -i Exp.def -o Exp.rtg -m Exp
SdfChecker: error: Main module not defined
--- Main

rtg2sig -i Exp.rtg -o Exp.str
strc -i ExpEval.str -la stratego-lib

[ strc | info ] Compiling 'ExpEval.str'
[ strc | info ] Front-end succeeded : [user/system] = [0.56s/0.05s]
[ strc | info ] Optimization succeeded -O 2 : [user/system] = [0.00s/0.00s]
[ strc | info ] Back-end succeeded : [user/system] = [0.16s/0.01s]
gcc -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -Wno-unused-label -Wno-unused-variable -Wno-unused-function -Wno-unused-parameter -DSIZEOF_VOID_P=4 -DSIZEOF_LONG=4 -DSIZEOF_INT=4 -c ExpEval.c -fno-common -DPIC -o .libs/ExpEval.o
gcc -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -Wno-unused-label -Wno-unused-variable -Wno-unused-function -Wno-unused-parameter -DSIZEOF_VOID_P=4 -DSIZEOF_LONG=4 -DSIZEOF_INT=4 -c ExpEval.c -o ExpEval.o >/dev/null 2>&1
[ strc | info ] C compilation succeeded : [user/system] = [0.31s/0.36s]
[ strc | info ] Compilation succeeded : [user/system] = [1.03s/0.42s]

sglri -p Exp.tbl -i ultimate-question.txt | ./ExpEval
```

Generate regular tree grammar

Generate signature

Compile to C

Parse and transform
Roadmap

> Program Transformation
  — Introduction
  — Stratego/XT
  — TXL
> Refactoring
> Aspect-Oriented Programming
The TXL paradigm: *parse, transform, unpars*
TXL was originally designed as a desugaring tool for syntactic extensions to the teaching language Turing (originally, TXL = “Turing eXtender Language”). Now it is more a general-purpose source to source transformation language.

See also:

https://en.wikipedia.org/wiki/Turing_(programming_language)
https://www.txl.ca
TXL programs

Base grammar

defines tokens and non-terminals

Grammar overrides

extend and modify types from grammar

Transformation rules

rooted set of rules and functions
TXL assumes that there is a host programming language as the target, whose syntax is specified by a base grammar, and a set of “grammar overrides” specifying extensions to the base grammar. TXL then applies a set of transformation rules that will “desugar” the extensions, yielding a valid program adhering to just the base grammar.
% Part I. Syntax specification

define program
   [expression]
end define

define expression
   [expression] + [term]
   |   [expression] - [term]
   |   [term]
end define

define term
   [term] * [primary]
   |   [term] / [primary]
   |   [primary]
end define

define primary
   [number]
   |   ( [expression] )
end define

% Part 2. Transformation rules

rule main
   replace [expression]
       E [expression]
   construct NewE [expression]
       E [resolveAddition]
       [resolveSubtraction]
       [resolveMultiplication]
       [resolveDivision]
       [resolveBracketedExpressions]
       where not
       NewE [= E]
   by
       NewE
end rule

rule resolveAddition
   replace [expression]
       N1 [number] + N2 [number]
   by
       N1 [+ N2]
end rule

... 

rule resolveBracketedExpressions
   replace [primary]
       ( N [number] )
   by
       N
end rule
This example specifies a grammar for an expression language (the same one we saw before), and rules to transform expressions by evaluating them.

TXL reverses the usual BNF convention and puts non-terminals in square brackets while interpreting everything else (except special chars) as terminals.

The default lexical scanner can be modified, but is usually fine for first experiments.

See:

   git://scg.unibe.ch/lectures-cc-examples
   subfolder: cc-TXL
Running the example

File: Ultimate.Question

1 + 2 * (3 + 4) * 3 - 1

txl Ultimate.Question
TXL v10.5d (1.7.08) (c)1988-2008 Queen's University at Kingston
Compiling Question.Txl ...
Parsing Ultimate.Question ...
Transforming ...
42
Example: TIL — a tiny imperative language

```til
// Find all factors of a given input number
var n;
write "Input n please";
read n;
write "The factors of n are";
var f;
f := 2;
while n != 1 do
    while (n / f) * f = n do
        write f;
        n := n / f;
    end
    f := f + 1;
end
```

File: factors.til

http://www.program-transformation.org/Sts/TILChairmarks
This toy language is used for various transformation examples.

See: http://www.program-transformation.org/Sts/TILChairmarks
% Keywords of TIL
keys
  var if then else while
do for read write
end keys

% Compound tokens
compounds
  := !=
end compounds

% Commenting convention
comments
  //
end comments

define program
  [statement*]
end define

define statement
  [declaration]
  | [assignment_statement]
  | [if_statement]
  | [while_statement]
  | [for_statement]
  | [read_statement]
  | [write_statement]
end define

% Untyped variables
define declaration
  'var [id] ;
end define

define assignment_statement
  [id] := [expression] ;
end define

define if_statement
  'if [expression] 'then
  |[statement*]
  |[opt else_statement]
  'end
end define

...
The [NL], [IN] and [EX] annotations tell the pretty-printer where to insert newlines, and where to indent and dedent code.
include "TIL.Grm"
function main
    match [program]
        _ [program]
    end function

var n;
write "Input n please";
read n;
write "The factors of n are";
var f;
f := 2;
while n != 1 do
    while (n / f) * f = n do
        write f;
        n := n / f;
    end
    f := f + 1;
end
Generating statistics

include "TIL.Grm"

function main
    replace [program]
        Program [program]

        % Count each kind of statement we're interested in
        % by extracting all of each kind from the program

    construct Statements [statement*]
        _ [^[ Program]
    construct StatementCount [number]
        _ [length Statements] [putp "Total: %"]

    construct Declarations [declaration*]
        _ [^[ Program]
    construct DeclarationsCount [number]
        _ [length Declarations] [putp "Declarations: %"]
    ...
    by
        % nothing
end function
Tracing

```plaintext
include "TIL.Grm"
...
redefine statement
  ...  
  | [traced_statement]
end redefine

define traced_statement
  [statement] [attr 'TRACED]
end define

rule main
replace [repeat statement]
  S [statement]
  Rest [repeat statement]
...
  by
    'write QuotedS;  'TRACED
    S                 'TRACED
    Rest
end rule
...
```

File: TILtrace.Txl

```plaintext
write "Trace: var n;"
var n;
write "Trace: write \"Input n please\";"
write "Input n please"
write "Trace: read n;"
read n;
...
```
This transformation replaces every statement by a “quoted” version of the statement (i.e., that prints the text of the statement preceded by “Trace: ”), followed by the original statement. Executing the program will then produce a printout of every statement just before it is executed.
## TXL vs Stratego

<table>
<thead>
<tr>
<th>Stratego</th>
<th>TXL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scannerless GLR parsing</td>
<td>Agile parsing (top-down + bottom-up)</td>
</tr>
<tr>
<td>Reusable, generic traversal strategies</td>
<td>Fixed traversals</td>
</tr>
<tr>
<td>Separates rewrite rules from traversal strategies</td>
<td>Traversals part of rewrite rules</td>
</tr>
</tbody>
</table>
Commercial systems

“The DMS Software Reengineering Toolkit is a set of tools for automating customized source program analysis, modification or translation or generation of software systems, containing arbitrary mixtures of languages.”

http://www.semdesigns.com/Products/DMS/DMSToolkit.html
Roadmap

> Program Transformation

> **Refactoring**
  
  — Refactoring Engine and Code Critics
  
  — Eclipse refactoring plugins

> Aspect-Oriented Programming
What is Refactoring?

> The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*.

Rename Method — manual steps

> Do it yourself approach:
   — Check that no method with the new name already exists in any subclass or superclass.
   — Browse all the implementers (method definitions)
   — Browse all the senders (method invocations)
   — Edit and rename all implementers
   — Edit and rename all senders
   — Remove all implementers
   — Test

> Automated refactoring is better!
Rename Method

> Rename Method (method, new name)

> Preconditions
   — No method with the new name already exists in any subclass or superclass.
   — No methods with same signature as method outside the inheritance hierarchy of method

> PostConditions
   — method has new name
   — relevant methods in the inheritance hierarchy have new name
   — invocations of changed method are updated to new name

> Other Considerations
   — Statically/Dynamically Typed Languages ⇒ Scope of the renaming
The Refactoring Browser
### Typical Refactorings

<table>
<thead>
<tr>
<th>Class Refactorings</th>
<th>Method Refactorings</th>
<th>Attribute Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (sub)class to hierarchy</td>
<td>add method to class</td>
<td>add variable to class</td>
</tr>
<tr>
<td>rename class</td>
<td>rename method</td>
<td>rename variable</td>
</tr>
<tr>
<td>remove class</td>
<td>remove method</td>
<td>remove variable</td>
</tr>
<tr>
<td>push method down</td>
<td>push method down</td>
<td>push variable down</td>
</tr>
<tr>
<td>push method up</td>
<td>pull variable up</td>
<td></td>
</tr>
<tr>
<td>add parameter to method</td>
<td>create accessors</td>
<td></td>
</tr>
<tr>
<td>move method to component</td>
<td>abstract variable</td>
<td></td>
</tr>
<tr>
<td>extract code in new method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


QualityAssistant — search for common errors
QualityAssistant is a tool integrated into the Pharo IDE. It uses syntactic rules to detect violations of quality rules, and can also propose fixes with the help of transformation rules.
Refactoring Engine — matching trees

NB: All metavariables start with `~`

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~</code></td>
<td>recurse</td>
</tr>
<tr>
<td><code>@</code></td>
<td>list</td>
</tr>
<tr>
<td><code>.</code></td>
<td>statement</td>
</tr>
<tr>
<td><code>#</code></td>
<td>literal</td>
</tr>
</tbody>
</table>

```
`~`@object halt
recursively match send of halt

`@`.Statements
match list of statements

Class `@message:~@args`
match all sends to Class```
Rewrite rules

initialize

super initialize.

self rewriteRule

replace: '``@object = nil' with: '``@object isNil';
replace: '``@object == nil' with: '``@object isNil';
replace: '``@object =~ nil' with: '``@object notNil';
replace: '``@object ~= nil' with: '``@object notNil'
Rewrite rules help a developer to easily specify a pattern that has to be matched and another pattern that should be used for replacement.
Complicated matching rules

```
initialize
  super initialize.
  self matcher
  matchesAnyOf: #(
    'i to: 'j do: [:e | @temps | @Stmts. 'x := 'x + 1. 'Stmts2]
    'i to: 'j by: 'k do: [:e | @temps | @Stmts. 'x := 'x + 'k. 'Stmts2]
    'i to: 'j do: [:e | @temps | @Stmts. 'x := 'x - 1. 'Stmts2]
    'i to: 'j by: 'k do: [:e | @temps | @Stmts. 'x := 'x - 'k. 'Stmts2]
  )
  do: [ :node :answer | node ]
```
Pattern-matching rules are powerful, but may be hard to maintain when they get more complex.
MatchTool

Pattern code

```
@i to: \@j do: [ ] :e |
  | `@temps |
    \@.Stmts.
  x := x + 1.
`@.Stmts2]
```

Test code

```
generateSequence
  1 to: 10 do: [ :x | 5 to: 7 do: [ :y |  
z |  z := x * i + y.  j := j + 1. ]  
  i := i + 1 ]
```
MatchTool provides an advanced way to work with the matching syntax. A developer can edit pattern code in the top left corner and it will be highlighted accordingly. The bottom left pane contains test code against which the pattern code from the top pane will be matched. The list in the middle contains all the matches (in this case 2). The right list contains a map specifying what each metavariable matched.

The example contains one of the match expressions from the “Complicated matching rules” slide.
Rewrite Tool

RewriteRuleBuilder

myDictionary := Dictionary new
at: 1 put: 'a';
at: 2 put: 'b';
yourself.

myDictionary
at: 3
ifAbsent:
    temp1 := 'c'.
temp2 := 'd'.
temp3 := temp1, temp2.
myDictionary at: 3 put: temp3.

Transformation Rule

Generate rule
Undo
Redo
Browse rules
While the MatchTool provides an interface to experiment with matching, the Rewrite Tool allows one to experiment while defining transformations.

From top left in a counter clockwise direction:

1) the target code of the transformation
2) matching code
3) replacement code
4) replacement result
Transformation is about AST

Old syntax:

Smalltalk ui icons smallPlayIcon

New syntax:

#smallPlayIcon asIcon
At some point developers decided to simplify the API to retrieve icons. It should be easy to transform code with matching syntax.
Transformation is about AST

Old syntax:

Smalltalk ui icons `icon

New syntax:

`icon asIcon
Transformation is about AST

Old syntax:

```
Smalltalk ui icons `icon
```

New syntax:

```
`icon asIcon
```

```
In theory you just have to take a “word” that comes after Smalltalk ui icons and append it with asIcon. Also you have to prepend it with # which is not a common use case for the rewrite syntax.
Transformation is about AST

Old syntax:

Smalltalk ui icons `icon

New syntax:

#`icon asIcon

message

variable (literal)
The problem is more complicated, because `icon in the first expression is a meta message sent while in the second one it is a meta variable. This means that the transformation cannot happen automatically.
Transformation is about AST

```ruby
initialize
  super initialize.
  self
    replace: 'Smalltalk ui icons `iconName`
    by: [ :node :matchMap |
        | literal |
        literal := RBLiteralNode value: '#', (matchMap at: `iconName`).
        RBMessageNode receiver: (literal) selector: 'asIcon' ]
```
Because matching engine works on AST we can obtain the matched result, take matched message selector, create a symbol literal out of it, and use it as a receiver of an `asIcon` massage.
Roadmap

> Program Transformation
> **Refactoring**
>   — Refactoring Engine and Code Critics
>   — *Eclipse* refactoring plugins
> > Aspect-Oriented Programming
A workbench action delegate

When the workbench action proxy is triggered by the user, it delegates to an instance of this class.

```java
package astexampleplugin.actions;
...
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

class ChangeAction implements IWorkbenchWindowActionDelegate {
...
    public void run(IAction action) {
        for (ICompilationUnit cu : this.classes) {
            try {
                ...
                parser.setSource(cu);
                ...
                CompilationUnit ast = (CompilationUnit)parser.createAST(null);
                ...
                StackVisitor visitor = new StackVisitor(ast.getAST());
                ast.accept(visitor);
            } catch ...
        }
    }
}
```

A field renaming visitor

```java
package astexampleplugin.ast;
...
import org.eclipse.jdt.core.dom.ASTVisitor;

public class StackVisitor extends ASTVisitor {
    private static final String PREFIX = "_";
    ...
    public boolean visit(FieldDeclaration field){
        ...
    }

    public boolean visit(FieldAccess fieldAccess){
        String oldName = fieldAccess.getName().toString();
        String newName = this.fields.get( oldName );
        if(newName == null){
            newName = PREFIX + oldName;
            this.fields.put( oldName , newName );
        }
        fieldAccess.setName( this.ast.newSimpleName( newName ) );
        return true;
    }
}
```

The visitor simply implements the `visit` method for field declarations and accesses, and prepends an underscore.
Renaming fields
Roadmap

- Program Transformation
- Refactoring
- Aspect-Oriented Programming
Problem: cross-cutting concerns

Certain features (like logging, persistence and security), cannot usually be encapsulated as classes. They *cross-cut* code of the system.
Aspect-Oriented Programming

AOP improves modularity by supporting the separation of cross-cutting concerns.

An **aspect** packages cross-cutting concerns

A **pointcut** specifies a set of **join points** in the target system to be affected

**Weaving** is the process of applying the aspect to the target system
package tjp;

public class Demo {
    static Demo d;
    public static void main(String[] args){
        new Demo().go();
    }
    void go(){
        d = new Demo();
        d.foo(1,d);
        System.out.println(d.bar(new Integer(3)));
    }
    void foo(int i, Object o){
        System.out.println("Demo.foo(" + i + ", " + o + ")\n");
    }
    String bar (Integer j){
        System.out.println("Demo.bar(" + j + ")\n");
        return "Demo.bar(" + j + ")";
    }
}

A logging aspect

aspect GetInfo {
  pointcut goCut(): cflow(this(Demo) && execution(void go()));
  pointcut demoExecs(): within(Demo) && execution(* *(.));
  Object around(): demoExecs() && !execution(* go()) && goCut() {
    ...
  }
  ...
}

Intercept execution within control flow of Demo.go()

Identify all methods within Demo

Wrap all methods except Demo.go()
These *pointcuts* define *join points* in the target program where to add code.

Each pointcut is associated with a predicate that specifies either static or dynamic conditions. A `cflow` predicate is dynamic. The `goCut()` pointcut intercepts the control flow of `Demo.go()`.

The `demoExecs()` pointcut intercepts all methods within the `Demo` class. Notice the use of the pattern “* * *( .. )”, which specifies any method with any return value and any number of arguments.

Finally, the *around* advice specifies the code to wrap the matching join points. In this case the pointcut specifies that we want to match all methods within `Demo` except `Demo.go()`.
aspect GetInfo {
  ...
  Object around(): demoExecs() && !execution(* go()) && goCut() {
    println("Intercepted message: "+
      thisJoinPointStaticPart.getSignature().getName());
    println("in class: "+
      thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
    printParameters(thisJoinPoint);
    println("Running original method: \n");
    Object result = proceed();
    println(" result: " + result );
    return result;
  }
 ...
}
Here are the details of the around advice. Note the analogy with the use of `super` in single-inheritance OO languages. We are effectively “overriding” each matched method with the given code. The invocation of `proceed()` indicates where the original method is to be invoked (i.e., wrapped), in exactly the same way we use a `super` invocation to wrap an overridden method in a subclass.

Note how `Demo.foo()` and `Demo.bar()` are logged but not `Demo.go()`.

(No apologies for the stupid toy example.)
Making classes visitable with aspects

public class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil l) {}  

    public void visit(Cons l) {
        sum = sum + l.head;
        l.tail.accept(this);
    }
}

public static void main(String[] args) {
    List l = new Cons(5, new Cons(4, new Cons(3, new Nil())));
    SumVisitor sv = new SumVisitor();
    l.accept(sv);
    System.out.println("Sum = " + sv.sum);
}

public interface Visitor {
    void visit(Nil l);
    void visit(Cons l);
}

public class List {}
public class Nil implements List {}
public class Cons implements List {
    int head;
    List tail;
    Cons(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
}

We want to write this

But we are stuck with this ...
In this example we would like to be able to “visit” an existing composite structure, but we cannot since the structure is not “visitable”. We also do not own the code, so we cannot simply modify it to make it visitable.
package ajvisit;

public aspect Visible { 
    public void List.accept(Visitor v) { }

    public void Nil.accept(Visitor v) {
        v.visit(this);
    }

    public void Cons.accept(Visitor v) {
        v.visit(this);
    }
}
This problem is easily solved by defining an aspect `Visitable` that defines the missing accept methods.

Here we are using aspects not to define any “before”, “after”, or “around” advice, but simply to add new methods.

This is also known as a “class extension”, or as a form of “monkey patching”.

See:

https://en.wikipedia.org/wiki/Monkey_patch
With aspects, who needs visitors?

```java
public class SumList {
    public static void main(String[] args) {
        List l = new Cons(5, new Cons(4, new Cons(3, new Nil())));
        System.out.println("Sum = "+l.sum());
    }
}
```

```java
public aspect Summable {
    public int List.sum() {
        return 0;
    }
    public int Nil.sum() {
        return 0;
    }
    public int Cons.sum() {
        return head + tail.sum();
    }
}
```
Don't forget that the Visitor pattern arose from a need to be able to add new algorithms to work with data structures after the fact. But AOP also offers a way to adapt classes after the fact, so perhaps we don't need visitors at all.

In fact, what we can do instead is to simply add methods directly to our List structure that do what our visitor is doing, i.e., compute sums directly.
What you should know!

- What are typical program transformations?
- What is the typical architecture of a PT system?
- What is the role of term rewriting in PT systems?
- How does TXL differ from Stratego/XT?
- How does the Refactoring Engine use metavariables to encode rewrite rules?
- Why can’t aspects be encapsulated as classes?
- What is the difference between a pointcut and a join point?
Can you answer these questions?

✎ How does program transformation differ from metaprogramming?
✎ In what way is optimization a form of PT?
✎ What special care should be taken when pretty-printing a transformed program?
✎ How would you encode typical refactorings like “push method up” using a PT system like TXL?
✎ How could you use a PT system to implement AOP?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:
- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[More information](http://creativecommons.org/licenses/by-sa/4.0/)