# Lecture 13: Code Generation and Optimization

**Eelco Visser**

**CS4200 Compiler Construction**
**TU Delft**
**December 2019**

TU Delft

# Code Generation and Optimization

## Code Generation

- Compilation schemas
- Correctness
- Mechanics

## Optimization of generated code

- Peephole optimization
- Tail recursion elimination

# Compilation Schemas

# Compilation Schemas

## How do language constructs translate to target code?

## Compilation schema

- Source language pattern
- Target language pattern
- Assuming translation for the pattern variables
- Additional constraints on source and/or target patterns

## Exploration

- Before constructing code generation rules
- Manually translate small fragments to understand schemas

## Examples

- Some compilation schemas for Tiger constructs

# Compilation Schema Schema

```
[[ c e1 … en ]]
```

```
ins
[[ e1 ]]
ins
…
ins
[[ en ]]
ins
```

Translation of language construct c with sub-expressions e1 … en

Combines translation of sub-expressions with instruction pattern

# Tiger Compilation Schemas: Arithmetic Expressions

```
[[ x ]]
; type of x is int
; x is n-th variable
```

```
iload n
```

```
[[ e1 + e2 ]]
```

```
[[ e1 ]]
[[ e2 ]]
iadd
```

```
[[ e1 * e2 ]]
```

```
[[ e1 ]]
[[ e2 ]]
imul
```

# Tiger Compilation Schemas: Control-Flow

```
[[
  (
    e1;
    e2;
    ...
    en
  )
]]
```

```
[[ e1 ]]
[[ e2 ]]
…
[[ en ]]
```

To do: Pop elements from stack!

Sequential composition

# Tiger Compilation Schemas: Control-Flow

```
[[ if e1 then e2 else e3 ]]
```

```
  [[ e1 ]]
ifeq else
  [[ e2 ]]
goto end
else:
  [[ e3 ]]
end:
```

Jump labels should be unique

# Tiger Compilation Schemas: Control-Flow

```
[[ while e1 do e2 ]]
```

⇒

```
goto check
body:
  [[ e2 ]]
check:
  [[ e1 ]]
ifne body
```

# Tiger Compilation Schemas: Control-Flow

```
[[ while e1 do e2 ]]
```

```
check:
  [[ e1 ]]
ifeq end
  [[ e2 ]]
goto check
end:
```

# Tiger Compilation Schemas: Function Definition and Call

```
[[ function f(n: int, …): int = e ]]
```

```
.method public static f(I…)I
  [[ e ]]
  ireturn
.end method
```

```
[[ f(e,…) ]]
```

```
[[ e ]]
...
invokestatic Program/f(I…)I
```

# Homework: More Tiger Schemas

## Compilation schemas for other Tiger constructs

- Let bindings with local variables
- For loop with bound iteration variable
- Break statement in for/while loop
- Array types, creation and access
- Record types, creation and access

# Homework: Compiling Nested Functions

```
let
  function power(x: int, n: int): int =
    let
      function pow(n: int): int =
        if n = 0 then 1 else x * pow(n - 1)
     in pow(n)
 in
   power(3, 4)
```

Tiger has nested function definitions

Design a compilation schema for compiling such functions to JVM byte code

# Homework: Compiling Nested Functions

```
let
  function power(x: int, n: int): int =
    let
      var p : int := 1
      function pow(n: int): int =
        if n = 0 then p
        else (
          p := x * p;
          pow(n - 1)
        )
     in pow(n)
 in
   power(3, 4)
```

That can also deal with programs like this one

# Correctness of Code Generation

# When is a code generator correct?

# Correctness of Code Generation

## Target code should be

- Syntactically correct
- Type correct
- Other well-formedness criteria
- Ensures that generated code compiles / runs on target platform

## Is that sufficient?

## Target code should preserve

- Names: has the same interface
- Types: computed values have same type
- Semantics: compute same values
- Ensures that generated code has the same semantics

# Guaranteeing Correctness of Generated Code

## Testing

- Test suite of source language code, apply compiler

## Verification

- Of generated code (post-translation validation)
- Of code generator

## Type checking

- Building verification into type checker of the meta language

## For all correctness criteria

- Different techniques apply to different criteria

# Byte Code Well-Formedness

Java Virtual Machine Specification 11

TU Delft

# JVM: Format Checking

## Format checking

- Magic number
- Predefined attributes have proper length
- Not truncated or have extra bytes
- Satisfy constant pool constraints
- Field and method references have valid names, classes, descriptors

# JVM: Constraint on Class File

## Static Constraints

- Only valid instructions
- First instruction at index 0
- Index of next instruction = index + length
- Target of jump is opcode within method
- …
- The index operand of each iload, fload, aload, istore, fstore, astore, iinc, and ret instruction must be a non-negative integer no greater than max_locals - 1.
- …

# JVM: Constraint on Class File

## Structural Constraints

- Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variable array, regardless of the execution path that leads to its invocation.
- Operand stack cannot grow to a depth greater than max_stack
- No more values can be popped from stack than it contains
- No local variable can be accessed before it is assigned a variable
- …

# JVM: Verification of class Files

## Class file verification guarantees

- No operand stack overflows or underflows
- All local variable uses and stores are valid
- Arguments of all JVM instructions are of valid types

## Specification

- Using Prolog predicates

# JVM Verification Summary

## Many non-trivial constraints on class files

## JVM verifies all class files

- Compiler is not a trusted component

## Compiler should generate correct class files

- Does not say anything about correctness of compiler
- Compiler generating byte code not necessarily semantics preserving

# Code Generation Mechanics

# Code Generation Mechanics

## Code generation

- Input: AST of source language program
  - with name and type annotations
- Output: machine instructions

## Mechanics

- What techniques are available to define translation?
- What are the advantages and disadvantages of these techniques?
- To what extent do these techniques help with verification?

# Code Generation by String Manipulation

# Printing Strings as Side Effect

```
to-jbc = ?Nil()   ; <printstring> "aconst_null\n"
to-jbc = ?NoVal() ; <printstring> "nop\n"
to-jbc = ?Seq(es) ; <list-loop(to-jbc)> es

to-jbc =
   ?Int(i);
   <printstring> "ldc ";
   <printstring> i;
   <printstring> "\n"

to-jbc = ?Bop(op, e1, e2) ; <to-jbc> e1 ; <to-jbc> e2 ; <to-jbc> op

to-jbc = ?PLUS()  ; <printstring> "iadd\n"
to-jbc = ?MINUS() ; <printstring> "isub\n"
to-jbc = ?MUL()   ; <printstring> "imul\n"
to-jbc = ?DIV()   ; <printstring> "idiv\n"
```

# String Concatenation

```
to-jbc: Nil()   -> "aconst_null\n"
to-jbc: NoVal() -> "nop\n"
to-jbc: Seq(es) -> <concat-strings> <map(to-jbc)> es

to-jbc: Int(i) -> <concat-strings> ["ldc ", i, "\n"]

to-jbc: Bop(op, e1, e2) -> <concat-strings> [ <to-jbc> e1,
                                              <to-jbc> e2,
                                              <to-jbc> op ]


to-jbc: PLUS()  -> "iadd\n"
to-jbc: MINUS() -> "isub\n"
to-jbc: MUL()   -> "imul\n"
to-jbc: DIV()   -> "idiv\n"
```

# String Interpolation

```
to-jbc: Nil()   -> $[aconst_null]
to-jbc: NoVal() -> $[nop]
to-jbc: Seq(es) -> <map-to-jbc> es

map-to-jbc: [] -> $[]
map-to-jbc: [h|t] ->
   $[[<to-jbc> h]
     [<map-to-jbc> t]]

to-jbc: Int(i) -> $[ldc [i]]
to-jbc: Bop(op, e1, e2) ->
 $[[<to-jbc> e1]
   [<to-jbc> e2]
   [<to-jbc> op]]

to-jbc: PLUS()  -> $[iadd]
to-jbc: MINUS() -> $[isub]
to-jbc: MUL()   -> $[imul]
to-jbc: DIV()   -> $[idiv]
```

# Summary: Code Generation by String Manipulation

## Printing strings

- Generated code depends on order of traversal of the AST
- Explicit layout (whitespace) management
- Verbose quotation and anti-quotation
- Escaping meta-variables
- Easy to make syntax errors
- Output needs to be parsed for further processing

## String concatenation

- Makes generation order independent

## String interpolation (templates)

- Makes quotation and anti-quotation more concise
- Layout (whitespace) from template layout

# Correctness of String-Based Code Generators

## All bets are off

- Only guarantee is that you get some text
- String interpolation may help with producing readable code
- Very easy to make even trivial syntactic errors

## Verification

- Use target code checker for verification
- No input independent guarantees

# Code Generation by Term Transformation

# Code Generation by Transformation

## AST to AST translation

- input: source language AST
- output: target language AST

## Defined using term rewrite rules

- Recognise AST pattern for language construct
- Recursively translate sub-terms
- Compose results with target code schema for language construct

## Intermediate representation (IR)

# Code Generation by Transformation: Example

```
to-jbc: Nil()   -> [ ACONST_NULL() ]
to-jbc: NoVal() -> [ NOP() ]
to-jbc: Seq(es) -> <mapconcat(to-jbc)> es

to-jbc: Int(i)    -> [ LDC(Int(i)) ]
to-jbc: String(s) -> [ LDC(String(s)) ]

to-jbc: Bop(op, e1, e2) -> <mapconcat(to-jbc)> [ e1, e2, op ]

to-jbc: PLUS()  -> [ IADD() ]
to-jbc: MINUS() -> [ ISUB() ]
to-jbc: MUL()   -> [ IMUL() ]
to-jbc: DIV()   -> [ IDIV() ]

to-jbc: Assign(lhs, e) -> <concat> [ <to-jbc> e, <lhs-to-jbc> lhs ]

to-jbc:     Var(x) -> [ ILOAD(x) ]  where <type-of> Var(x) => INT()
to-jbc:     Var(x) -> [ ALOAD(x) ]  where <type-of> Var(x) => STRING()
lhs-to-jbc: Var(x) -> [ ISTORE(x) ] where <type-of> Var(x) => INT()
lhs-to-jbc: Var(x) -> [ ASTORE(x) ] where <type-of> Var(x) => STRING()
```

```
to-jbc : Exp -> List(Instruction)
```

# Code Generation by Transformation: Example

```
to-jbc:
   IfThenElse(e1, e2, e3) -> <concat>  [ <to-jbc> e1
                            , [ IFEQ(LabelRef(else)) ]
                            , <to-jbc> e2
                            , [ GOTO(LabelRef(end)), Label(else) ]
                            , <to-jbc> e3
                            , [ Label(end) ]
                            ]
   where <newname> "else" => else
   where <newname> "end"  => end

to-jbc:
   While(e1, e2) -> <concat> [ [ GOTO(LabelRef(check)), Label(body) ]
                      , <to-jbc> e2
                      , [ Label(check) ]
                      , <to-jbc> e1
                      , [ IFNE(LabelRef(body)) ]
                      ]
   where <newname> "test" => check
   where <newname> "body" => body
```

## Compiler component composition

- AST output can be consumed by compatible AST transformations

## Example compilation pipeline

- Parse source language text => source language AST
- Desugar => source language AST
- Type-check => annotated source language AST
- Translate => target language AST
- Optimize => target language AST
- Pretty-print => target language text

## Easy to extend with new components

# Guaranteeing Syntactically Correct Target Code

TUDelft

# Syntactically Correct Target Code

## Property: Syntactically correct target code

- Guarantee that generated code parses

## Type correct AST = syntactically correct code

- AST types represent syntactic categories
  - `Plus: Exp * Exp -> Exp`
- Type check translation patterns

## Language support

- Any programming language with a static type system
- And support for algebraic data types

## Note: lexical syntax

# Type Checking Transformation Rules

```
module Tiger-Condensed
signature
  constructors
    Var         : Id -> Var
    String      : StrConst -> Exp
    Seq         : List(Exp) -> Exp
    Call        : Var * List(Exp) -> Exp
    Plus        : Exp * Exp -> Exp
    Minus       : Exp * Exp -> Exp
    Assign      : Var * Exp -> Exp
    If          : Exp * Exp * Exp -> Exp
    Let         : List(Dec) * List(Exp) -> Exp
    VarDec      : Id * TypeAn * Exp -> Dec
    FunctionDec : List(FunDec) -> Dec
    FunDec      : Id * List(FArg) * TypeAn * Exp -> FunDec
    FArg        : Id * TypeAn -> FArg
    NoTp        : TypeAn
    Tp          : TypeId -> TypeAn
```

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Type checking terms in rules guarantees
*syntactic* correctness of generated code

# Guaranteeing Syntactically Correct Target Code in Stratego?

## :-(

## Stratego

- Only checks arities of constructor applications, not types
- Transformation rules could be checked by the compiler
- Generic traversals make traditional type checking impossible

## Research

- A static analysis for Stratego that guarantees syntactic correctness

## Workaround

- Meta-programming with concrete object syntax

This paper defines a generic technique for embedding the concrete syntax of an object language into a meta-programming language.

Applied to Stratego as meta-language and Tiger as object language.

Combines two advantages
- guarantee syntactic correctness of match and build patterns
- make rules more readable

https://doi.org/10.1007/3-540-45821-2_19

# Meta-programming with Concrete Object Syntax

Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. http://www.cs.uu.nl/~visser, visser@acm.org

**Abstract.** Meta programs manipulate structured representations, i.e., abstract syntax trees, of programs. The conceptual distance between the concrete syntax meta-programmers use to reason about programs and the notation for abstract syntax manipulation provided by general purpose (meta-) programming languages is too great for many applications. In this paper it is shown how the syntax definition formalism SDF can be employed to fit *any* meta-programming language with concrete syntax notation for composing and analyzing object programs. As a case study, the addition of concrete syntax to the program transformation language Stratego is presented. The approach is then generalized to arbitrary meta-languages.

## 1 Introduction

Meta-programs analyze, generate, and transform object programs. In this process object programs are structured data. It is common practice to use abstract syntax trees rather than the textual representation of programs [10]. Abstract syntax trees are represented using the data structuring facilities of the meta-language: records (structs) in imperative languages (C), objects in object-oriented languages (C++, Java), algebraic data types in functional languages (ML, Haskell), and terms in term rewriting systems (Stratego).

Such representations allow the full capabilities of the meta-language to be applied in the implementation of meta-programs. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching (e.g., ML, Haskell) it is easy to compose and decompose abstract syntax trees. For meta-programs such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages.

However, there are many applications of meta-programming in which the use of abstract syntax is not satisfactory since the conceptual distance between the

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 299-315, 2002.

# Concrete Object Syntax

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Abstract syntax transformation

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    [[ function f(xs) = e ]] ->
    [[ function f(xs) = (enterfun(s); e; exitfun(s)) ]]
    where !f => s
  TraceFunction :
    [[ function f(xs) : tid = e ]] ->
    [[ function f(xs) : tid =
         (enterfun(s);
          let var x : tid := nil in x := e; exitfun(s); x end) ]]
    where new => x ; !f => s
  IntroducePrinters :
    e -> [[ let var ind := 0
                function enterfun(name : string) = (
                  ind := +(ind, 1);
                  for i := 2 to ind do print(" ");
                  print(name); print(" entry\\n"))
                function exitfun(name : string) = (
                  for i := 2 to ind do print(" ");
                  ind := -(ind, 1);
                  print(name); print(" exit\\n"))
            in e end ]]
```

Concrete syntax transformation

# Implementing Concrete Object Syntax

```
module StrategoTiger
imports
  Tiger Tiger-Sugar Tiger-Variables Tiger-Congruences
imports
  Stratego [ Id => StrategoId
         Var => StrategoVar
             StrChar => StrategoStrChar ]
exports
  context-free syntax
    "[[" Dec       "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" FunDec    "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" Exp       "]]"  -> Term        {cons("ToTerm"),prefer}
    "~"  Term            -> Exp         {cons("FromTerm"),prefer}
    "~*" Term            -> {Exp ","}+  {cons("FromTerm")}
    "~*" Term            -> {Exp ";"}+  {cons("FromTerm")}
    "~"  Term            -> Id          {cons("FromTerm")}
    "~*" Term            -> {FArg ","}+ {cons("FromTerm")}
```

Embedding of object language into meta language

# From Concrete Syntax to Abstract Syntax

```
[[ x := let ds in ~* es end ]] -> [[ let ds in x := (~* es) end ]]
```

parse

```
Rule(ToTerm(Assign(Var(meta-var("x")),
                   Let(meta-var("ds"),FromTerm(Var("es"))))),
     ToTerm(Let(meta-var("ds"),
                [Assign(Var(meta-var("x")),
                        Seq(FromTerm(Var("es"))))])))
```

Mixed AST

explode

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

Pure AST

pretty-print

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

# Meta Explode

```
module meta-explode
imports lib Stratego
strategies
  meta-explode =
    alltd(?ToTerm(<trm-explode>) + ?ToStrategy(<str-explode>))

  trm-explode =
    TrmMetaVar <+ TrmStr <+ TrmFromTerm <+ TrmFromStr <+ TrmAnno
    <+ TrmConc <+ TrmNil <+ TrmCons <+ TrmOp

  TrmOp       : op#(ts) -> Op(op, <map(trm-explode)> ts)

  TrmMetaVar  : meta-var(x) -> Var(x)
  TrmStr      = is-string; !Str(<id>)
  TrmFromTerm = ?FromTerm(<meta-explode>)
  TrmFromStr  = ?FromStrategy(<meta-explode>)
  TrmAnno     = Anno(trm-explode, meta-explode)
  TrmNil      : [] -> Op("Nil", [])
  TrmCons     : [x | xs] -> Op("Cons",[<trm-explode>x, <trm-explode>xs])
  TrmConc     : Conc(ts1,ts2) ->
                <foldr(!<trm-explode> ts2,
                       !Op("Cons", [<Fst>, <Snd>]), trm-explode)> ts1
```

Find term embedding

Explode it

How do you type check that?

The concrete syntax embedding techniques is not specific to Stratego as meta-language. This paper shows how to use it to embed DSLs into Java.

```
ATerm x = id |[ propertyChangeListeners ]| ;

ATerm stm = bstm |[ {
    if(x == null) return;
    PropertyChangeEvent event =
      new PropertyChangeEvent(this, f, v1, v1);
    for(int c=0; c < x.size(); c++) {
      ((...)x.elementAt(c)).propertyChange(event);
    }
  }
]| ;
```

https://doi.org/10.1145/1035292.1029007

# Concrete Syntax for Objects

## Domain-Specific Language Embedding and Assimilation without Restrictions

Martin Bravenboer
Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands
martin@cs.uu.nl

Eelco Visser
Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands
visser@acm.org

### ABSTRACT

Application programmer's interfaces give access to domain knowledge encapsulated in class libraries without providing the appropriate notation for expressing domain composition. Since object-oriented languages are designed for extensibility and reuse, the language constructs are often sufficient for expressing domain abstractions at the semantic level. However, they do not provide the right abstractions at the syntactic level. In this paper we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level. The method is supported by proven and available technology, i.e. the syntax definition formalism SDF and the program transformation language and toolset Stratego/XT. We illustrate the method with applications in three domains: code generation, XML generation, and user-interface construction.

### Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.3 [**Programming Languages**]: Processors

**General Terms:** Languages, Design

**Keywords:** MetaBorg, Stratego, SDF, Embedded Languages, Syntax Extension, Extensible Syntax, Domain-Specific Languages, Rewriting, Meta Programming, Concrete Object Syntax



### 1. INTRODUCTION

Class libraries encapsulate knowledge about the domain for which the library is written. The application programmer's interface to a library is the means for programmers to access that knowledge. However, the generic language of method invocation provided by object-oriented languages does often not provide the right notation for expressing domain-specific composition. General purpose languages, particularly object-oriented languages, are designed for extensibility and reuse. That is, language concepts such as objects, interfaces, inheritance, and polymorphism support the construction of class hierarchies with reusable implementations that can easily be extended with variants. Thus, OO languages provide the flexibility to develop and evolve APIs according to growing insight into a domain.

Although these facilities are often sufficient for expressing domain abstractions at the semantic level, they do not provide the right abstractions at the syntactic level. This is obvious when considering the domain of arithmetic or logical operations. Most modern languages provide infix operators using the well known notation from mathematics. Programmers complain when they have to program in a language where arithmetic operations are made available in the same syntax as other procedures. Consider writing `e1 + e2` as `add(e1, e2)` or even `x := e1; x.add(e2)`. However, when programming in other domains such as code generation, document processing, or graphical user-interface construction, programmers are forced to express their designs using the generic notation of method invocation rather than a more appropriate domain notation. Thus programmers have to write code such as

```
JPanel panel =
  new JPanel(new BorderLayout(12,12));
panel.setBorder(
  BorderFactory.createEmptyBorder(15,15,15,15));
```

in order to construct a user-interface, rather than using a more compositional syntax reflecting the nice hierarchical structure of user-interface components in the Swing library. Building in syntactic support for such domains in a general purpose language is not feasible, however, because of the different speeds at which languages and domain abstractions develop. A language should strive for stability, while libraries can be more volatile.

In this paper we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding*

This paper generalizes the concrete syntax techniques to all sorts of host and guest languages, with an application to preventing injection attacks.

Injection attacks are caused by unhygienic construction of code through which user input can be turned into executable code.

doi:10.1016/j.scico.2009.05.004

Science of Computer Programming 75 (2010) 473–495

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Preventing injection attacks with syntax embeddings[☆]

Martin Bravenboer[a,*], Eelco Dolstra[b], Eelco Visser[b]

[a] *Department of Computer Science, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA 01003, USA*

[b] *Department of Software Technology, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands*

A R T I C L E   I N F O

*Article history:*
Received 7 March 2008
Received in revised form 18 May 2009
Accepted 21 May 2009
Available online 31 May 2009

*Keywords:*
Injection attacks
Security
Syntax embedding
Program generation
Program transformation
Concrete object syntax

A B S T R A C T

Software written in one language often needs to construct sentences in another language, such as SQL queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, the concatenation of constants and client-supplied strings. A client can then supply specially crafted input that causes the constructed sentence to be interpreted in an unintended way, leading to an *injection attack*. We describe a more natural style of programming that yields code that is impervious to injections *by construction*. Our approach embeds the grammars of the *guest languages* (e.g. SQL) into that of the *host language* (e.g. Java) and automatically generates code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. This approach is generic, meaning that it can be applied with relative ease to any combination of context-free host and guest languages.



## 1. Introduction

In this paper we propose using *syntax embedding* to prevent injection vulnerabilities in a language-independent way. Injections form a very common class of security vulnerabilities [22]. Software written in one language often needs to construct sentences in another language, such as SQL, XQuery, or XPath queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, whereby constant and client-supplied strings are concatenated to form the sentence. Consider for example the following piece of server-side Java code that authenticates a remote HTTP user against a database, where getParam() returns a string supplied by the user, for instance through a form field:

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

On testing, this code may appear to work correctly, but it is vulnerable to a very common security flaw. For instance, if the user specifies as the password the string `' OR 'x' = 'x`, then the constructed SQL query will be

```
SELECT id FROM users WHERE name = '...' AND password = '' OR 'x' = 'x'
```

☆ An earlier version appeared in GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering.

* Corresponding author.
*E-mail addresses:* martin.bravenboer@acm.org (M. Bravenboer), e.dolstra@tudelft.nl (E. Dolstra), visser@acm.org (E. Visser).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.05.004

# Hygienic

```
$username = $_GET['username'];
$q = "SELECT * FROM users WHERE username = '" . $username . "'";
executeSQL($q);
```

*SQL in PHP: SQL injection vulnerability*

```
String e = "/users[@name='" + name + "' and " +
                  "@password='" + password + "']";
factory.newXPath().evaluate(e, doc);
```

*XPath in Java: XPath injection vulnerability*

```
$searchfilter = "(cn=" . $username . ")";
$search = ldap_search($connection, $directory, $searchfilter);
```

*LDAP in PHP: LDAP injection vulnerability*

```
$command = "svn cat \"file name\" -r" . $rev;
system($command);
```

*Shell calls in PHP: command injection vulnerability*

```
String topic = getParam("topic");
String query = "SELECT body FROM comments WHERE topic = '" + topic + "'";
ResultSet results = executeQuery(query);
foreach (String body : results)
  println("<tr><td>" + body + "</td></tr>");
```

*XML and SQL in Java: XSS vulnerability*

```
$username = $_GET['username'];
$q = <| SELECT * FROM users WHERE username = ${$username} |>;
executeSQL($q->toString());
```

*SQL in PHP*

```
XPath e = {- /users[@name=${name} and @password=${password}] -};
factory.newXPath().evaluate(e.toString(), doc);
```

*XPath in Java*

```
$searchfilter = (| (cn=$($username)) |);
$search = ldap_search($connection, $directory, $searchfilter->toString());
```

*LDAP in PHP*

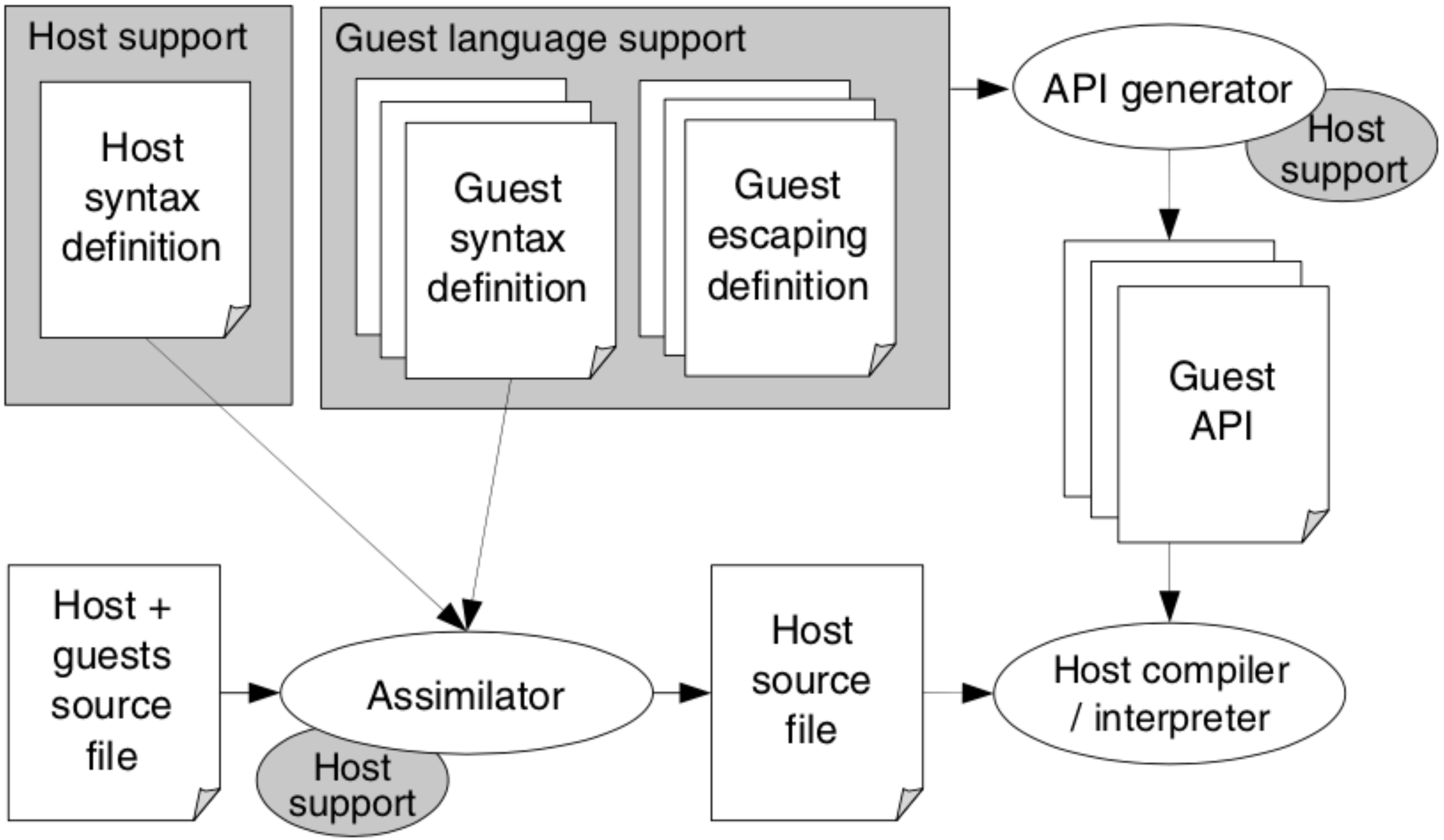
```
$command = <| svn cat "file name" -r${$rev} |>;
system($command->toString());
```

*Shell calls in PHP*

```
String topic = getParam("topic");
SQL query = <| SELECT body FROM comments WHERE topic = ${topic} |>;
ResultSet results = executeQuery(query.toString());
foreach (String body : results)
  println(<tr><td>${body}</td></tr>.toString());
```

*XML and SQL in Java*

# A Generic Architecture



# Hygienic Transformations

# Hygienic Transformations

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Does new variable in TraceProcedure not capture variables in e?

# Guaranteeing Hygiene

## Guarantee that variables are not captured

- Which variables?

## Object language name analysis for transformation rules

- E.g. apply Tiger constraint rules to patterns in rules

## Existing approaches

- Hygienic macros in Scheme/Racket
- Higher-order abstract syntax
- Nominal abstract syntax

## Research

- Hygienic transformations for more complex binding patterns

# Guaranteeing Type Correct Target Code

TU Delft

# Guaranteeing Type Correct Code

## Property: Type correct target code

- Guarantee that generated code type checks

## Intrinsically-typed ASTs

- Encode type system in algebraic signature
- Including binding structure
- Language support: Generalized ADTs

## Research

- Advanced type systems & binding patterns

# Semantics Preservation

# Interface Preservation

**Generate code has same interface as source code**

# Type Preservation

## Generated code produces values with the same type

## Intrinsically-typed interpreters for imperative languages

- POPL18 paper
- Verify that interpreters are type preserving
- Including non-lexical binding patterns

## Research

- how to do this for other transformations?

# Dynamic Semantics Preservation

## Semantics preservation

- Generated code has the same behaviour as the source program

## CompCert

- Certified C compiler
- Defines operational semantics of source language (most of C) and all intermediate languages
- Mechanically verify that translations between IR preserve behaviour
  - For all possible programs
- Or: verify that generated output has same behaviour as input
  - For programs that compiler is applied to

# Optimizing (Virtual) Machine Code

# Optimizations

## Reasons

- code overhead
- execution overhead

## Inlining

- replace calls by body of the procedure
- source code level

## Tail recursion

- replace recursive calls by loops or jumps
- source or machine code level

# Peephole Optimization

TUDelft

# Code Generation

```
function fac0(n0: int): int =
   if
      n0 = 0
   then
      1
   else
      n0 * fac0(n0 - 1)
```

```
.method public static fac0(I)I

              iload 1
              ldc 0
              if_icmpeq label0
              ldc 0
              goto label1
   label0:    ldc 1
   label1:    ifeq else0
              ldc 1
              goto end0
   else0:     iload 1
              iload 1
              ldc 1
              isub
              invokestatic
                 Exp/fac0(I)I
              imul
   end0:      ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

           iload 1
           ldc 0
           if_icmpeq label0
           ldc 0
           goto label1
   label0: ldc 1
   label1: ifeq else0
           ldc 1
           goto end0
   else0:  iload 1
           iload 1
           ldc 1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
   end0:   ireturn
.end method
```

```
.method public static fac0(I)I

           iload_1
           ifne else0

           iconst_1
           ireturn

   else0:  iload_1
           dup
           iconst_1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
           ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ldc 0
            if_icmpeq label0
            ldc 0
            goto label1
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifeq label0

            ldc 0
            goto label1
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            ldc 0
            goto label1
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            ldc 0
            ifeq else0
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            ldc 0
            ifeq else0
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            goto else0

   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            goto else0
   label0:  ldc 1
   label1:  ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            goto else0
   label0:  ldc 1
            ifeq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

           iload 1
           ifeq label0
           goto else0
   label0: ldc 1
           ifeq else0
           ldc 1
           goto end0
   else0:  iload 1
           iload 1
           ldc 1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
   end0:   ireturn
.end method
```

```
.method public static fac0(I)I

           iload 1
           ifeq label0
           goto else0
   label0: ldc 1


           goto end0
   else0:  iload 1
           iload 1
           ldc 1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
   end0:   ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifeq label0
            goto else0
   label0:  ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifneq else0

   label0:  ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifneq else0
   label0:  ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            goto end0
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            ireturn
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            ireturn
   else0:   iload 1
            iload 1
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            ireturn
   else0:   iload 1
            dup
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

            iload 1
            ifneq else0
            ldc 1
            ireturn
   else0:   iload 1
            dup
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

```
.method public static fac0(I)I

            iload_1
            ifneq else0
            ldc 1
            ireturn
   else0:   iload_1
            dup
            ldc 1
            isub
            invokestatic
               Exp/fac0(I)I
            imul
   end0:    ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

             iload_1
             ifneq else0
             ldc 1
             ireturn
   else0:    iload_1
             dup
             ldc 1
             isub
             invokestatic
                Exp/fac0(I)I
             imul
   end0:     ireturn
.end method
```

```
.method public static fac0(I)I

             iload_1
             ifneq else0
             iconst_1
             ireturn
   else0:    iload_1
             dup
             iconst_1
             isub
             invokestatic
                Exp/fac0(I)I
             imul
   end0:     ireturn
.end method
```

# Optimization

```
.method public static fac0(I)I

           iload 1
           ldc 0
           if_icmpeq label0
           ldc 0
           goto label1
   label0: ldc 1
   label1: ifeq else0
           ldc 1
           goto end0
   else0:  iload 1
           iload 1
           ldc 1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
   end0:   ireturn
.end method
```

```
.method public static fac0(I)I

           iload_1
           ifneq else0
           iconst_1
           ireturn
   else0:  iload_1
           dup
           iconst_1
           isub
           invokestatic
              Exp/fac0(I)I
           imul
           ireturn
.end method
```

# Implementing Peephole Optimizations

# Tail Recursion Elimination

TUDelft

# Tail Recursion Elimination

## Recursive function

- creates a new stack frame for each invocation

## Tail recursion

- recursive call is last action

## Tail recursion elimination

- If function is tail recursive, reuse stack frame for recursive call

# Example: Non-Tail Recursive Function

```
function fac(n : Int) =
  if n = 1
  then
    1
  else
    n * fac(n - 1)
```

```
.class public Exp

   .method public static fac(I)I

            iload 1
            ifne else

            iconst_1
            ireturn

      else: iload 1
            dup
            iconst_1
            isub
            invokestatic Exp/fac(I)I
            imul
            ireturn
   .end method
```

# Example: Make Tail Recursive

```
function fac(n : Int) =
  if n = 1
  then
    1
  else
    n * fac(n - 1)
```

```
function fac(n : Int) =
  fac2(n, 1)

function fac2(n : Int, acc: Int) =
  if n = 1
  then
    acc
  else
    fac(n - 1, n * acc)
```

Use an accumulator argument to build return value

# Example: Make Tail Recursive (in Byte Code)

```
.class public Exp

  .method public static fac(I)I

          iload 1
          ifne else

          iconst_1
          ireturn

    else: iload 1
          dup
          iconst_1
          isub
          invokestatic Exp/fac(I)I
          imul
          ireturn
  .end method
```

```
.class public Exp

  .method public static fac(II)I

          iload 1
          ifne else

          iload 2
          ireturn

    else: iload 1
          iconst_1
          isub
          iload 1
          iload 2
          imul
          invokestatic Exp/fac(II)I
          ireturn
  .end method
```

# Example: Tail Recursion Elimination

```
.class public Exp

   .method public static fac(II)I

               iload 1
               ifne else

               iload 2
               ireturn

         else: iload 1
               iconst_1
               isub
               iload 1
               iload 2
               imul
               invokestatic Exp/fac(II)I
               ireturn
   .end method
```

```
.class public Exp

   .method public static fac(II)I

         strt: iload 1
               ifne else

               iload 2
               ireturn

         else: iload 1
               iconst_1
               isub
               iload 1
               iload 2
               imul
               istore 2
               istore 1
               goto strt
  .end method
```

# Next: Memory Management