# Considerations on Porting Perl to the Java Virtual Machine

University of Cincinnati
ECECS Department
Master's Thesis Defense
22 January 2001

Bradley M. Kuhn

# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

# Introduction

- Java Virtual Machine
- Why Port to the JVM?
- Porting Challenges

# Introduction

- Java Virtual Machine
- Why Port to the JVM?
- Porting Challenges

# Java Virtual Machine

- Well Defined Architecture
- Class File Interface
- Bytecode Verification

# Java Virtual Machine

- Well Defined Architecture
- Class File Interface
- Bytecode Verification

# JVM Class File Interface

- superclass and interfaces
- constant pools
- methods and signatures
- class and instance variables
- code segment

# Java Virtual Machine

- Well Defined Architecture
- Class File Interface
- Bytecode Verification

# Bytecode Verification

- Reasons:
  - efficiency
  - security
- Problems:
  - impede valid use
  - compiler must be aware of restrictions

# Introduction

- Java Virtual Machine
- Why Port to the JVM?
- Porting Challenges

# Why port to JVM?

## Embedded JVMs (Software)

- Web Browsers
- Jemacs - JVM as universal interface
- PDA Architectures
- Software JVMs breath life into old hardware

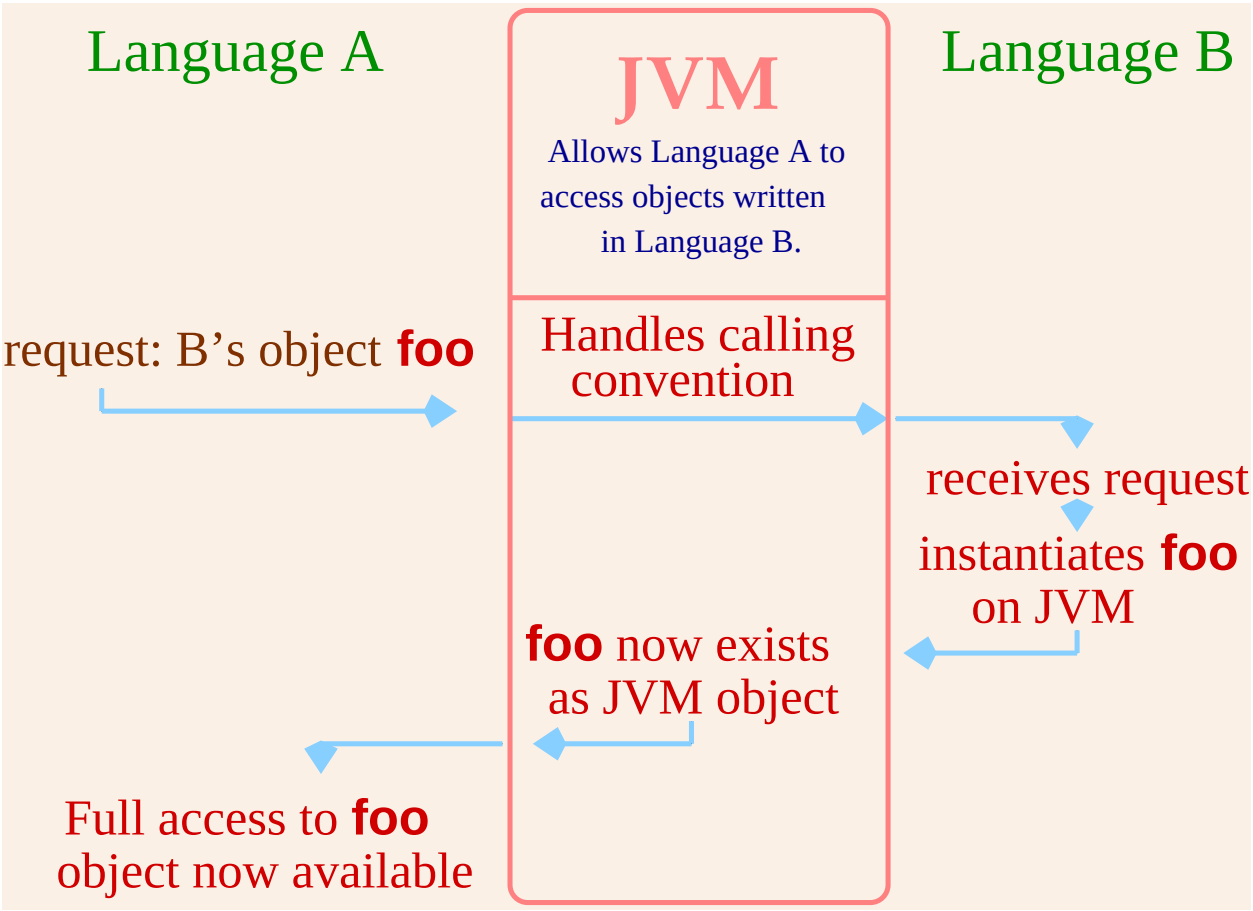
# Why port to JVM?

## Embedded JVMs (Hardware)

- JVM on a chip
- Active focus of research and industry
- More languages make hardware more useful
- JVM is really "just another" platform

# Why port to JVM?

## Interoperability



# Introduction

- Java Virtual Machine
- Why Port to the JVM?
- Porting Challenges

# General Challenges

- Code must pass verification
- Language must map to JVM class file
- Class file must be "Object-Aware"

# Perl Specific Challenges

- Perl is hard to parse
  - "Six people understand the parser"
- No formal specification (yet)
- uncommon and rich native data types

# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

# Possible Approaches

- Java Native Interface (JNI)
- Survey of Approaches
- Which Approach for Perl?

# Possible Approaches

- Java Native Interface (JNI)
- Survey of Approaches
- Which Approach for Perl?

# Java Native Interface (JNI)

- C Interface to JVM
- Not a "native C port"
- Used by JPL
- Different Problem Space

# Possible Approaches

- Java Native Interface (JNI)
- Survey of Approaches
- Which Approach for Perl?

# Interpreter in Java?

Advantages

- Straight-forward approach
- Java code always runs on JVM
- "eval" is native

Disadvantages

- Everything is interpreted.  Maybe twice!

# Source-to-Source?

Advantages

- No need to worry about JVM details

Disadvantages

- Java source is not as versatile as JVM
- Java must be syntacticly correct
- tried and failed for Perl

# Direct Compilation?

Advantages

- Leverage existing front-end
- Avoid code divergence

Disadvantages

- Existing compiler’s IR might not match JVM
- Might treat JVM as glorified assembler

# Map Language Idioms

## Advantages

- Seamless, easy-to-understand port
- Code is straightforward

## Disadvantages

- Still needs some JVM code generation
- Works only if data types match perfectly

# Possible Approaches

- Java Native Interface (JNI)
- Survey of Approaches
- Which Approach for Perl?

# Which Approach For Perl?

- Direct Compilation (Retarget)
  - Leverage perl front-end
  - IR Mismatch
  - JVM restrictions
- New Approach
  - Map to existing Kawa IR
  - Still leverage front-end

# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

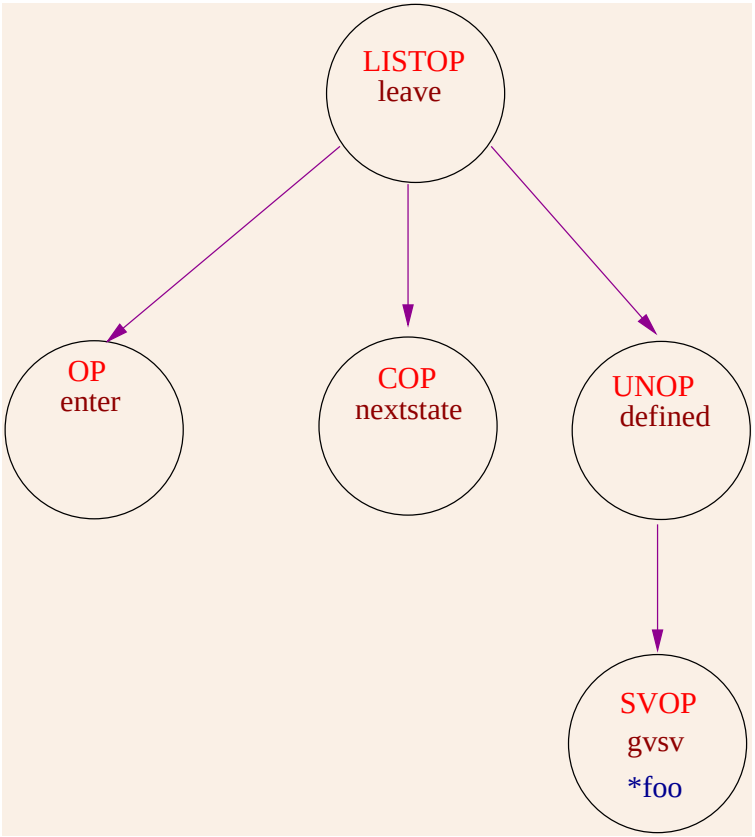
# perl Internals

- Front-end
  - parses Perl
  - compiles to intermediate representation (IR)
- Back-end, aka "Perl Virtual Machine" (PVM)
  - PVM has an implementation of Perl’s native data types
  - PVM "walks" the IR, executes code, using native data types

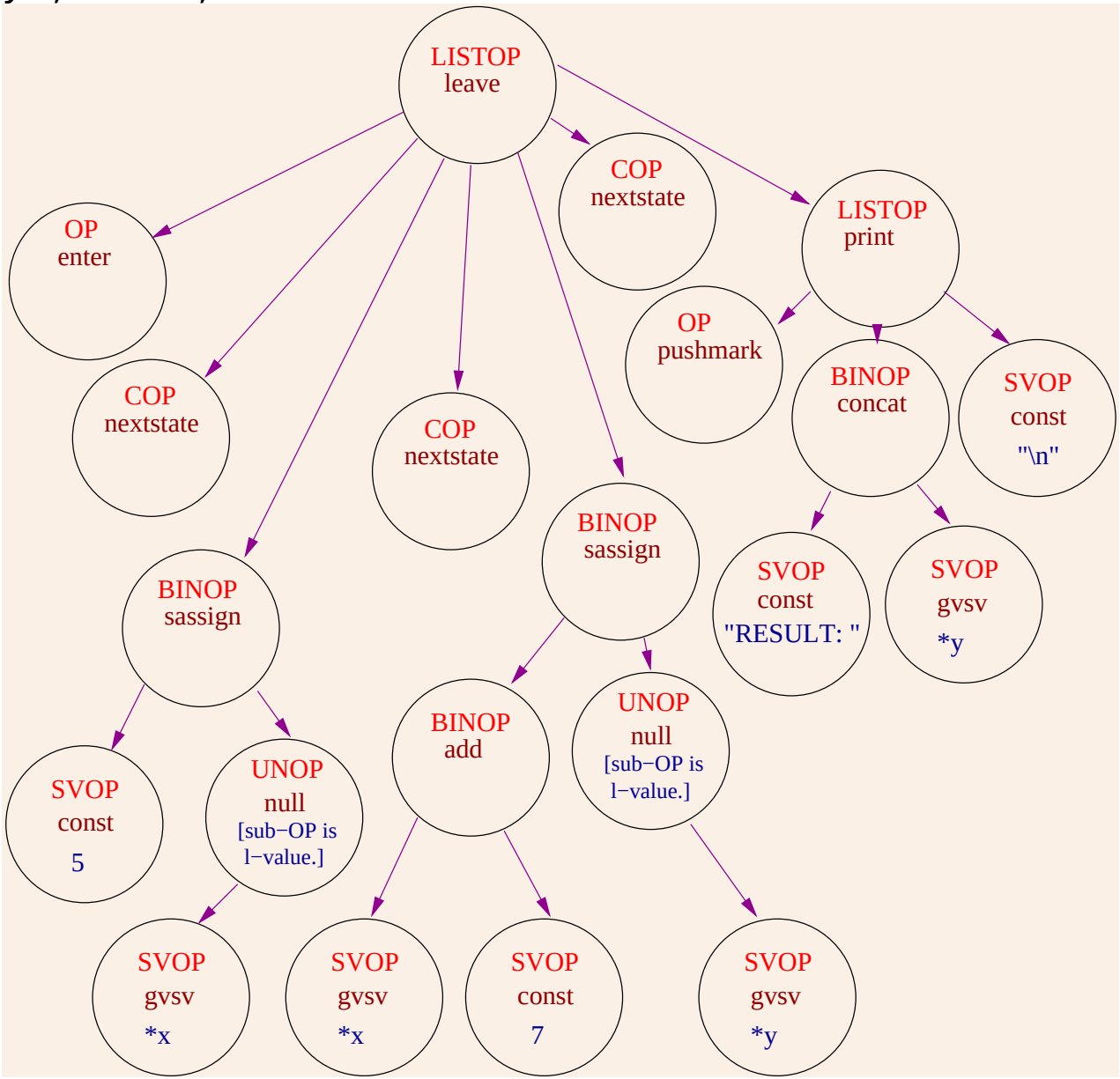
# A Simple Example

The program: `defined $foo;`

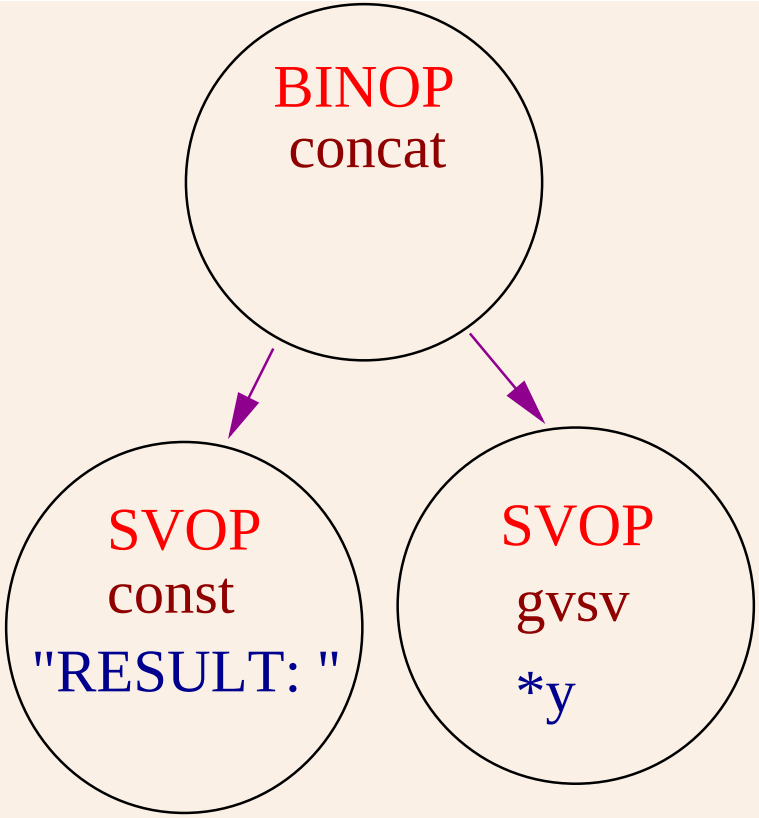
Generates the tree:



```
$x = 5;
$y = $x + 7;
print "RESULT: $y", "\n";
```



# Stack Before concat




STACK

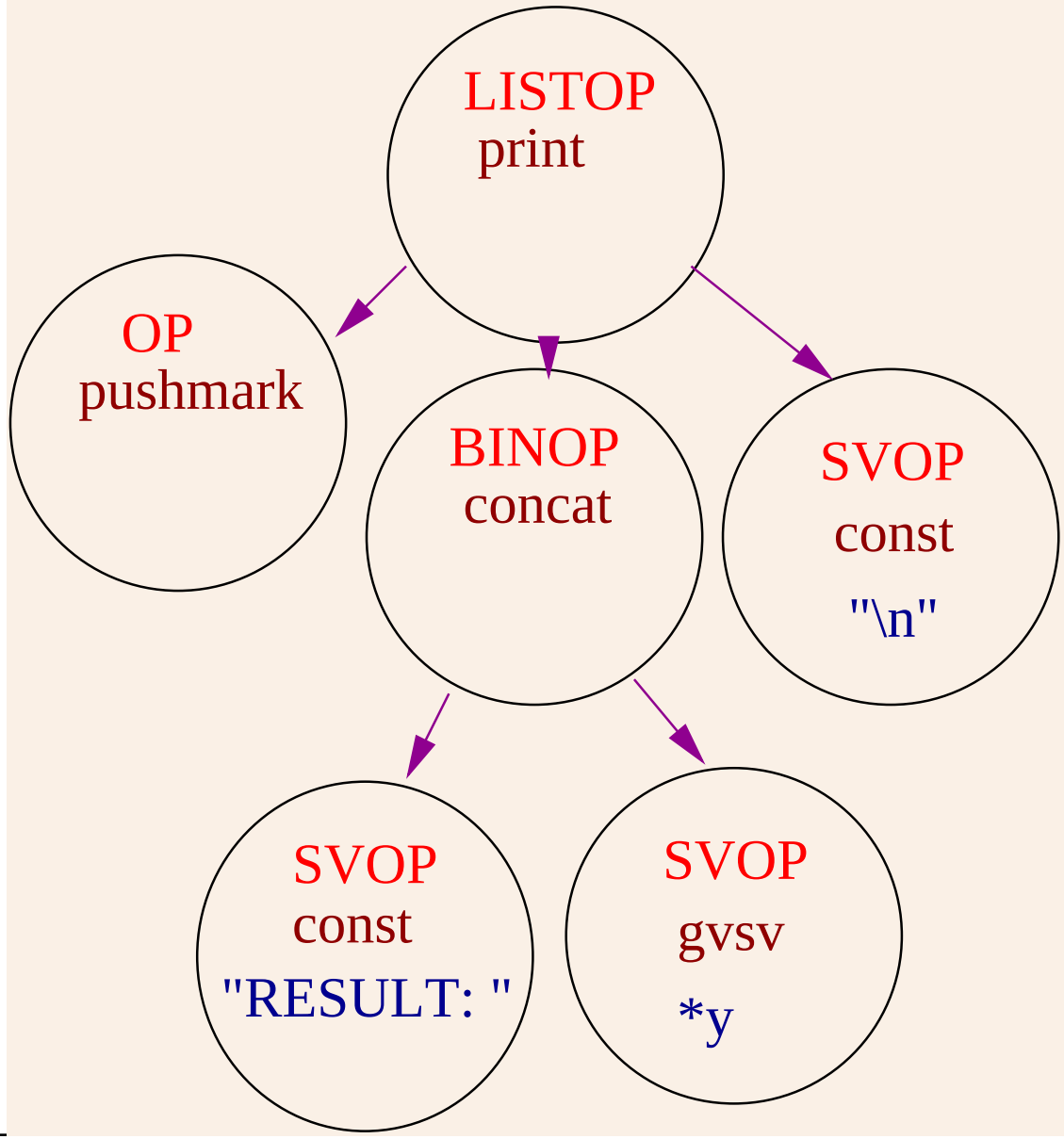
(before concat)

*

"RESULT: "

12

# Stack After concat




STACK

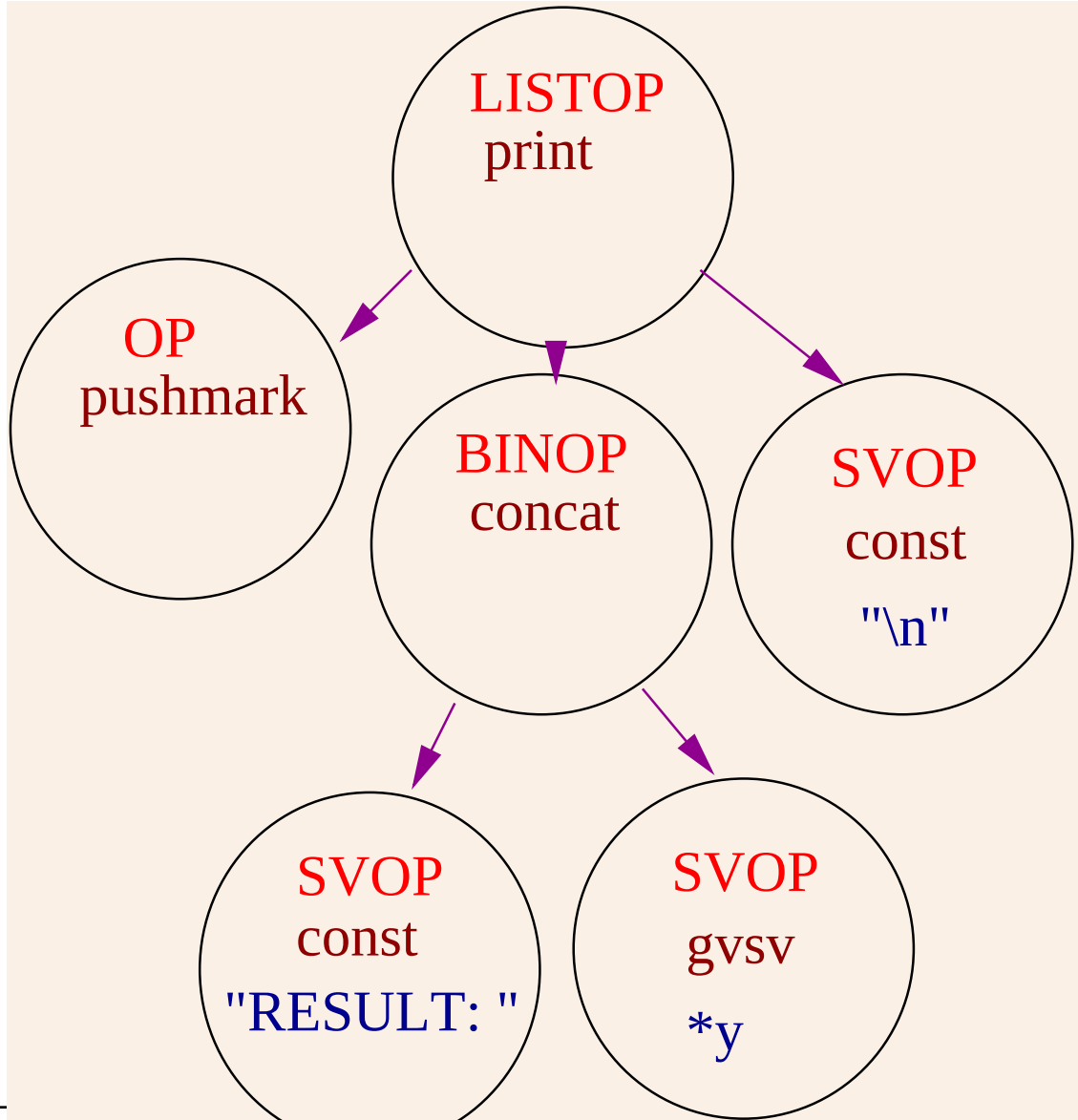
(before print)

*

"RESULT: 12"

"\n"

# Stack After print




STACK

(after print)

SV_YES

# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

# Direct Compilation

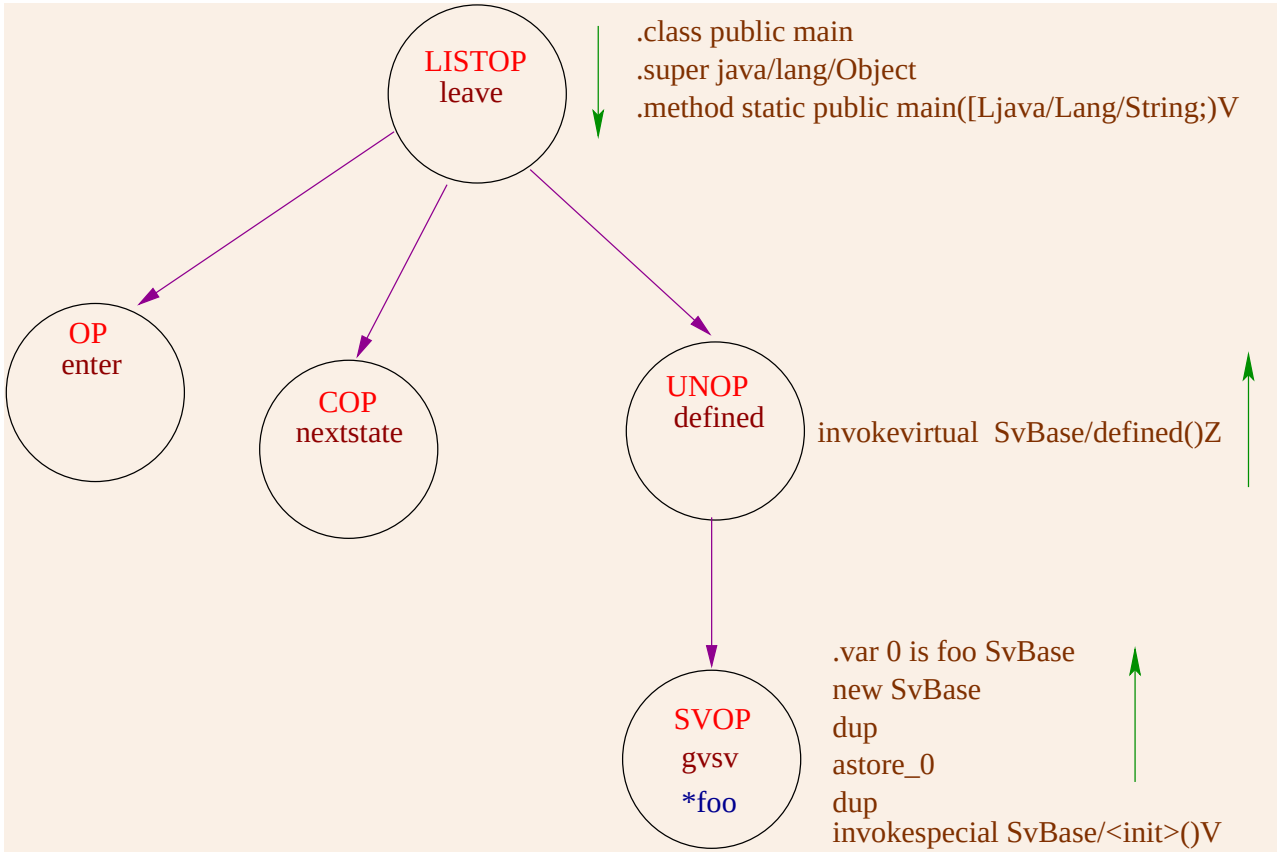
Leverage From perl:

- B:: controls the "walk"; examining IR
- emit JVM assembler while "walking" the tree

JVM assembler:

- use Jasmin syntax
- wrapper object for other assemblers

# Code Emission



# Emitted Source

Jasmin source

```
.class public main
.super java/lang/Object
.method static public main([Ljava/lang/String;)V
.var 0 is foo LSvBase
new SvBase
dup
astore_0
dup
invokespecial SvBase/<init>()V
invokevirtual SvBase/defined()Z
```

Java equivalent (didactic purpose only)

```
class main {
  static public void main(String argv[]) {
    SvBase foo = new SvBase();   foo.defined();
  }
}
```

# Using Other Classes?

- Perl types must exist natively on JVM
- Implement PVM objects in Java
  - previous example: SvBase matches "SvNULL"
- Can cause trouble...

# Failure of Direct Compilation

- reimplement PVM in Java!
- meticulous code generation
- Short-cuts lead to unverifable code

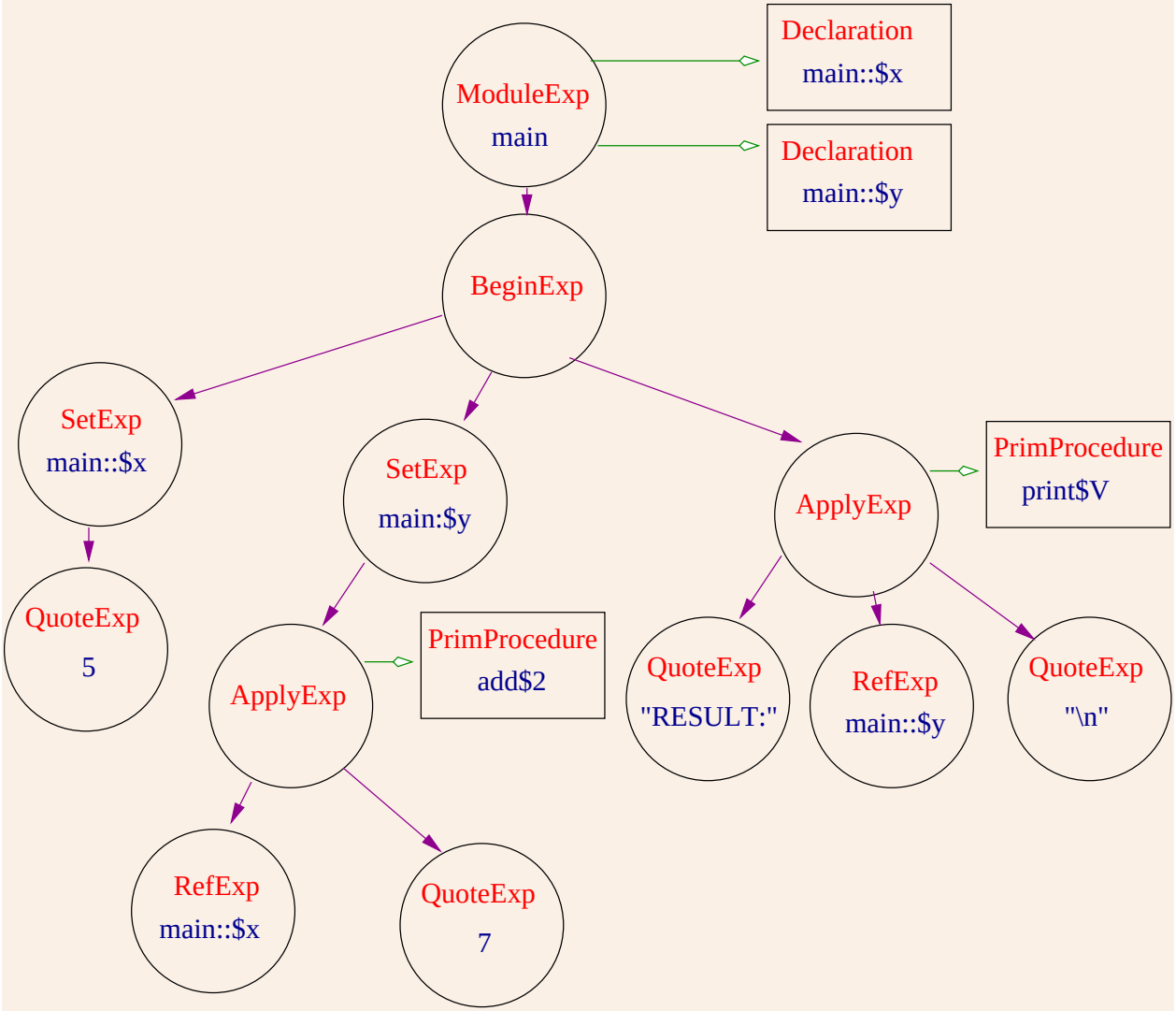
# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

# What Kawa Provides

- A new layer of abstraction
- Higher-level IR for JVM
- Promise of true language interoperability

```
$x = 5;
$y = $x + 7;
print "RESULT: $y", "\n";
```



# Overview

- Introduction
- Possible Approaches
- Internals of perl
- Direct Compilation
- Kawa Integration
- Conclusions / Future Work

# Conclusions

- perl’s front-end can be leveraged, with care
- Additional layer of abstraction needed
- Kawa and JVM together show promise
- Kawa/JVM can compete with .NET

# Future Work

- Use Kawa to finish Perl port to JVM
- Kawa/JVM optimizations: other languages?
- Experiments in Language Interoperability

Questions, Grilling, Tar-and-feather-ing...